1988

Service Execution in a Distributed Environment (Thesis)

Craig E. Wills

Report Number:
88-769

https://docs.lib.purdue.edu/cstech/658

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
SERVICE EXECUTION IN A DISTRIBUTED ENVIRONMENT

Craig E. Wills

CSD-TR-769
May 1988
SERVICE EXECUTION IN A DISTRIBUTED ENVIRONMENT

A Thesis
Submitted to the Faculty

of

Purdue University

by

Craig E. Wills

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

May 1988
ACKNOWLEDGMENTS

I would like to thank my major professor, Tim Korb, for his continued support and guidance throughout this endeavor as he helped me keep a perspective on my work through the good times and bad. I would also like to thank Doug Comer for planting the seed of a research idea many summers ago and then helping me with its growth as a member of my thesis committee. I would also like to thank the other members of my committee, Piyush Mehrotra and Ryan Stansifer, for their time, suggestions, and encouragement. Many systems and staff people also helped me along the way. Special thanks to Ralph, Chris, Paul, Dan, Jim, Raj, and John.

A few people deserve special thanks for their help. Thanks to my parents and family for supporting me in all of my pursuits. Thanks to Cristina for being my first close friend at Purdue and for your constant encouragement and belief in me. Thanks to Bala for being my friend and colleague these past few years and for the numerous chats we had along the way. Thanks to my friend Sara for helping me reach higher and see farther. Finally, thanks to John, Thomas, and Susan for your friendship and making our house a home as we all dealt with the rigors of pursuing a Ph.D.

This work was supported in part by grants from the National Science Foundation (MCS-8219178), Sun Microsystems Incorporated, and Digital Equipment Corporation.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>GLOSSARY</td>
<td>ix</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xiv</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The TILDE Project</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Statement of Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.3 The Thesis</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Outline of Dissertation</td>
<td>4</td>
</tr>
<tr>
<td>2. ACTION EXECUTION MODEL</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Approach</td>
<td>6</td>
</tr>
<tr>
<td>2.3 The Initiation Component</td>
<td>7</td>
</tr>
<tr>
<td>2.4 The Binding Components</td>
<td>7</td>
</tr>
<tr>
<td>2.4.1 Resolution</td>
<td>8</td>
</tr>
<tr>
<td>2.4.2 Protocol and Mapping</td>
<td>9</td>
</tr>
<tr>
<td>2.4.3 Location and Selection</td>
<td>10</td>
</tr>
<tr>
<td>2.5 The Invocation Component</td>
<td>10</td>
</tr>
<tr>
<td>2.5.1 Authentication</td>
<td>11</td>
</tr>
<tr>
<td>2.5.2 Execution Environment</td>
<td>11</td>
</tr>
<tr>
<td>2.6 Analysis of the Model</td>
<td>12</td>
</tr>
<tr>
<td>2.6.1 Component Binding Times</td>
<td>12</td>
</tr>
<tr>
<td>2.6.2 Component Binding Location</td>
<td>13</td>
</tr>
<tr>
<td>2.6.3 Transparency and the Model Components</td>
<td>14</td>
</tr>
<tr>
<td>2.6.4 Orthogonality and Completeness of the Model Components</td>
<td>14</td>
</tr>
</tbody>
</table>
### 3. APPLICATION OF THE MODEL TO EXISTING WORK

3.1 Command Execution
   - 3.1.1 UNIX
   - 3.1.2 Locus
   - 3.1.3 Other Command Execution Work
   - 3.1.4 Summary

3.2 Network Services

3.3 Remote Procedure Call

3.4 Object-Oriented Systems

3.5 Component Specific Work
   - 3.5.1 Initiation
   - 3.5.2 Resolution and Naming
   - 3.5.3 Location and Selection
   - 3.5.4 Summary

### 3.6 Summary

### 4. A NEW APPROACH

4.1 Introduction

4.2 Definition of the Service Execution Mechanism
   - 4.2.1 Initial Ordering
   - 4.2.2 Additional Ordering
   - 4.2.3 Summary

4.3 Overview of the Service Execution Mechanism

4.4 Service Initiation

4.5 Service Naming
   - 4.5.1 Naming Styles
   - 4.5.2 The Service Name Space
   - 4.5.3 Service Equivalence
   - 4.5.4 Metaservices
   - 4.5.5 Extending the Service Name Space

4.6 Service Resolution
   - 4.6.1 Treatment of Executable Files
   - 4.6.2 Name Resolution Conflicts
   - 4.6.3 Using Service Information for Resolution

4.7 Service Location and Selection
   - 4.7.1 The Service Selection Problem
   - 4.7.2 Our Approach to Service Selection

4.8 Choosing an Invocation Protocol
   - 4.8.1 Execution Environment
   - 4.8.2 Authentication
   - 4.8.3 Summary

4.9 Summary
5. LOCATING DISTRIBUTED INFORMATION

5.1 Introduction

5.2 Management Techniques for Distributed Information
5.2.1 Direct Access
5.2.2 Multicast
5.2.3 Local Store
5.2.4 Network Master
5.2.5 Variations
5.2.6 Summary

5.3 Evaluation Criteria

5.4 Application of Information Management Techniques

5.5 Locating Service Information
5.5.1 Service Information Management Techniques
5.5.2 Evaluation of the Techniques
5.5.3 Summary

5.6 Locating Server Status Information
5.6.1 Server Status Information Management Techniques
5.6.2 Evaluation of the Techniques
5.6.3 Summary

5.7 Summary

6. SERVICE EXECUTION MECHANISM PROTOTYPES

6.1 Introduction

6.2 The Proto-A Prototype
6.2.1 Implementation Environment
6.2.2 Prototype Organization and Management of Information
6.2.3 Service Registration
6.2.4 Service Execution Interface and Components
6.2.5 Service Invocation
6.2.6 Experience with Proto-A
6.2.7 Improvements on Proto-A

6.3 The Proto-B Prototype
6.3.1 Implementation Environment
6.3.2 Prototype Organization and Management of Information
6.3.3 Service Registration
6.3.4 Service Execution Interface and Components
6.3.5 Service Invocation
6.3.6 Experience with Proto-B
6.3.7 Improvements on Proto-B

6.4 Summary
7. CONCLUSIONS .................................................. 88

7.1 Review of Goals and Motivation ................................ 88
7.2 Results and Evaluation .......................................... 88
  7.2.1 Action Execution Model ................................. 88
  7.2.2 Service Execution Mechanism ......................... 89
  7.2.3 Service Naming and Resolution ....................... 89
  7.2.4 Locating Service Information ......................... 90
  7.2.5 Service Selection ...................................... 91
  7.2.6 Service Invocation ................................... 91
7.3 Future Directions ............................................ 92
7.4 Summary .................................................. 93

BIBLIOGRAPHY .................................................... 95

VITA ............................................................. 102
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Rate of Packets Handled for Each Machine</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Rate of Packets Handled on the Network</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>Rate of CPU Time Use for Each Machine</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>Response Time to Retrieve Information</td>
<td>60</td>
</tr>
<tr>
<td>5.5</td>
<td>Storage Required for Each Machine</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>Timeliness of the Information</td>
<td>61</td>
</tr>
<tr>
<td>5.7</td>
<td>Classification of UNIX Processes as Services</td>
<td>64</td>
</tr>
<tr>
<td>5.8</td>
<td>Access Rate per User for Service and Server Status Information</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Service Related Services</td>
<td>82</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 A TILDE Computing Engine</td>
<td>2</td>
</tr>
<tr>
<td>2.1 Components and Binding Times of the Model</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Action Execution Model for UNIX</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Action Execution Model for Locus</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Action Execution Model for Network Services</td>
<td>21</td>
</tr>
<tr>
<td>3.4 Action Execution Model for Birrell and Nelson's RPC</td>
<td>22</td>
</tr>
<tr>
<td>3.5 Action Execution Model for Eden</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Execution of a Service</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Examples of Metaservices</td>
<td>40</td>
</tr>
<tr>
<td>4.3 Examples of Service Resolution</td>
<td>42</td>
</tr>
<tr>
<td>4.4 Ideal Model of Service Invocation</td>
<td>48</td>
</tr>
<tr>
<td>4.5 Realistic Model of Service Invocation</td>
<td>49</td>
</tr>
<tr>
<td>4.6 Action Execution Model for the Service Execution Mechanism</td>
<td>51</td>
</tr>
<tr>
<td>5.1 Frequency of Process Invocation During an Average Day</td>
<td>64</td>
</tr>
<tr>
<td>5.2 Criteria for Service Information Management Techniques</td>
<td>67</td>
</tr>
<tr>
<td>5.3 Frequency of Load Updates for Fixed SLD Values</td>
<td>70</td>
</tr>
<tr>
<td>5.4 Criteria for Server Status Information Management Techniques</td>
<td>72</td>
</tr>
<tr>
<td>6.1 Implementation Environment for Prototype Proto-B</td>
<td>79</td>
</tr>
<tr>
<td>6.2 Two Pipeline Methods for ( A: serv_1</td>
<td>B: serv_2</td>
</tr>
</tbody>
</table>
GLOSSARY

The terminology used and defined in this dissertation is reproduced below for convenient reference.

absolute name: a name that maps to a unique action. Also called a complete or unique name.

access transparency: a system in which all actions are initiated in the same manner whether invoked on the client or a server machine.

action: a named computation that can be specified for execution.

action attribute: first attribute of a service name that describes what the service does.

action name: a string of symbols, usually characters, used to refer to a specific computation.

action name space: set of action names that follow the same naming convention.

active object: an object, such as a process, that performs work and manipulates the passive objects.

alias: an alternate name for an object allowing multiple names to refer to the same object.

attribute: a descriptive name for an object. Attributes may be explicitly named with a keyword, and the value of the attribute may be of a particular type.

authentication: the act of determining if the initiator, such as a user, of an action has permission to invoke the action.

authority attribute: second attribute of a service name that explicitly indicates the person who provides a service and vouches for its function.

autonomous distributed environment: a set of machines where each machine has control over its own resources rather than being in a pool of processors.

capabilities: a set of access rights for an object and the permitted actions on the object.

class attribute: third attribute of a service name that is used to distinguish a set of services that share a particular property.

client machine: executes actions for users interacting with the machine and requests actions to be invoked by server machines on behalf of the user.
**command**: an action invoked for execution by creating a new process and loading the image for the process with the contents of a binary file.

**complete**: as related to the components of the action execution model it means that taken together the components make all decisions necessary to execute an action. As related to attributes, it refers to an attribute that is associated with all objects it describes.

**component**: a step of the action execution model needed to execute an action.

**computation**: the work done by a process.

**computing engine**: a cluster of heterogeneous machines loosely coupled with high-speed local area networks.

**context**: an interpretation of a set of names.

**current working directory**: a single context that is commonly used to resolve file names in a hierarchical file system.

**cycle server**: a server process resident on server machines. Knows how to invoke a service given a service name by mapping the service to its underlying name.

**decision service**: makes decisions about where and what services to initiate based on service specific criteria.

**directed resolution**: successive contexts are applied to a partial name until a single absolute name is found.

**early binding time**: a decision made at session initialization or action specification time, which means the decision is made by the initiator of an action.

**execution environment**: the set of bindings between names and values used by an action during execution.

**execution**: the task of translating an action name into the invocation of the computation needed to carry out the action.

**flat name space**: an object is named by a single system-wide unique identifier that may be a number or a character name.

**global binding**: a binding between a name and a value that is shared between all actions in a system.

**heterogeneous distributed environment**: a set of machines that run different native operating systems.

**hierarchical name space**: a structured set of levels, where each level is an attribute, and a name is a set of attributes separated by delimiters.
homogeneous distributed environment: a set of machines that each run the same native operating system, although the processor type may be different between the machines.

inclusive resolution: all contexts are applied to a partial name and the result is the set of all valid absolute names that are found.

inheritance: in terms of the execution environment it means that the execution environment of the process invoking the action is established for the process carrying out the invoked action.

invocation protocol: a set of conventions for communicating with an existing process or creating a new process to carry out the computation defined by an action.

late binding time: a decision made at action execution time, which means the decision is made dynamically after the action has been initiated and execution has begun.

local binding: a binding between a name and a value that is specific to an invoked action and initialized at invocation time.

location transparency: a system in which the names for actions are independent of where the action is invoked.

master machines: intermediate machines in a network that either know all information in the network or cooperate to share all information among themselves.

metaservice: a special kind of service that, instead of containing a protocol and underlying name to invoke a service directly, contains "pointers" to other services. Execution of a metaservice causes one of the other services to be invoked.

name resolution mechanism: for an action translates a partial name into a complete name in the action name space.

negative information: information about the unavailability or negative status of a resource.

network transparency: a system in which an invoked action executes in the same manner regardless of where it is invoked.

ordering relation: is denoted by the symbol "<" and is defined between two components if one component must always precede the other.

orthogonal: as related to the components of the action execution model it means that each component is conceptually independent of the other components.

parameter: a binding between a name and a value where the names are known to the invoked action and the values are bound by the invoker.

partial name: a non-absolute name within a name space.
passive object: an object that has worked performed on it, such as a file, device, or mailbox.

pipeline: a series of actions where the output data of one action is used as the input of the next action.

protocol transparency: a system in which the names for actions are independent of how the action is invoked.

search path: used in conjunction with directed resolution to produce an absolute name from a partial name and a set of contexts.

server machine: invokes actions based on requests from client machines.

server process: a nontransient process used to control specific resources within a distributed environment.

service: a user-level action that is offered by one or more machines where the identification of the service is separated from its execution.

service execution mechanism: performs the steps necessary to execute a service.

service instance: an instantiation of a service offered by one or more machines in the computing engine.

service registration: the act of instantiating an instance of a service on a particular machine.

service server: a server process resident on each machine in the computing engine.

Knows about the services available on its machine, and controls what services are offered to users on the local and remote machines in the computing engine.

service set: an ordered set of attributes that is used to resolve a service name when a partial service name (one with only an action attribute) is given.

shell script: a set of shell commands and services grouped together into a single executable file that is available as a service.

significant load difference: (SLD) occurs when the current sampled load of a machine and the last reported load differ by more than a certain value.

specification: a mapping between a condition and an action name to execute when the condition is true.

static binding time: a decision made at system generation time, which means the decision is the same for every action that is executed within the system.

system interface: executes an action by translating an action name into an invoked action.
underlying name: the name needed by an invocation protocol to invoke an action.

user interface: allows the user to form specifications and initiate actions for execution.
ABSTRACT

Wills, Craig E. PhD, Purdue University, May, 1988. Service Execution in a Distributed Environment. Major Professor: John T. Korb.

The research presented in this dissertation is specifically concerned with understanding the issues and problems of transparently executing computations in a computing engine, a distributed environment of machines loosely coupled with high-speed local area networks. The central idea of our approach is that actions available to the user in a distributed system should be treated as services, where a service is a user-level action that is offered by one or more machines. Our approach is to separate the identification of a service from its execution. This abstraction allows the details of performing a service to be hidden from the user, and allows the user to specify what services he would like to use and not be concerned with where or how the services are invoked.

We model the essential steps necessary to execute a computation in a distributed environment in terms of a set of orthogonal components that each perform a specific task. For the model we denote a named computation as an action and term the model the action execution model. Using the action execution model we have a common representation for studying how actions, such as commands, network services, and remote procedure calls, are executed in a variety of distributed environments.

We also apply the action execution model to a set of design principles derived from our service abstraction to develop a framework for the design of a service execution mechanism. Our service execution mechanism provides the user consistent and uniform access to computations from multiple machines in a distributed environment. The design addresses a number of issues concerning how services are named; how service information is located in the computing engine to determine which services are available and which machines offer these services; how to select a machine from among multiple machines providing a service; and how to invoke the service on the selected machine.

We have implemented a prototype of the service execution mechanism on a network of Sun workstations, which provide local services and invoke remote services on behalf of users, and VAX machines running the UNIX operating system augmented with Sun's Network File System. Our experience with the prototype indicates that the service mechanism adds a relatively small overhead for locally invoked services while providing transparent access to all services whether local or remote. We conclude that the separation of what a service does from where and how it is invoked is the right approach for transparent execution of computations in a distributed, especially heterogeneous, environment.
1. INTRODUCTION

As computing environments grow from a single machine to a distributed environment of machines connected by networks, the capability to access resources on remote machines in a transparent manner is desirable. Transparency means that how the resource is accessed and where it is located is not part of the resource name and therefore does not need to be known by the user of the resource. The research presented in this dissertation is specifically concerned with understanding the issues and problems of transparently executing computations in a multi-machine architecture, which we call a computing engine. A computing engine is a cluster of heterogeneous machines loosely coupled with high-speed local area networks. A computing engine was the environment for research of the TILDE project [CKTM84], which served as the starting point for our research. More specifically, our work is an outgrowth of the DASH project, which looked at how the user accessed the computing engine [Kor84].

1.1 The TILDE Project

The TILDE project (Transparent Integrated Local and Distributed Environment) was concerned with problems in the development of distributed computing environments. Figure 1.1 represents the structure of a TILDE computing engine, which consists of many types of nodes communicating through an interconnection network. In the computing engine, workstations, on the behalf of users, execute computations available from execution servers, retrieve files available from storage servers, and contact other special servers such as time and weather servers. The interconnection network supports both point-to-point communication between any two nodes as well as broadcast communication in the computing engine. The system is dynamic and allows machines to be added or deleted without affecting the overall operation of the computing engine. For our work we assume that all machines within a computing engine are under the control of a centralized authority that arbitrates the naming of machines and user accounts.

The model used for performing computations in the TILDE system is that all computations are performed by processes, much like the UNIX operating system [RT74]. Processes are used to access system resources, communicate with other processes, and spawn child processes for execution of additional tasks. In addition, nontransient processes, servers, are used to control specific resources within the distributed environment.

The research described in this dissertation began as part of the DASH (Dynamic Access to Shared Hosts) portion of the TILDE project, which was concerned with the problems faced by users interacting with a TILDE environment through workstations. Our work has extended beyond the lifetime of the project, but retained the TILDE model that users will use workstations to access computations in a distributed environment. Users
request computations to be performed on their behalf by interacting with an agent on the workstation such as a shell. The computation requested by the user may be invoked by the shell itself, by spawning a new process on the workstation to perform the computation, or by contacting a server process on an execution server to request the computation be performed.

1.2 Statement of Problem

An important goal of the TILDE project was to incorporate transparency in order to present the user with an integrated interface to a distributed network of heterogeneous resources. Work on transparency proceeded in two directions: transparent file access through the development of a per-process naming environment called Tilde file naming \cite{CM86b,Dro86}; and the problem addressed by our work on how to provide transparent access to computations in a distributed environment.

This problem is a key issue in the use of a distributed environment, especially in a computing engine where machines are heterogeneous and each machine is autonomous. We define heterogeneous to mean that machines within the environment run different native operating systems. In a homogeneous environment all machines run the same native operating system, although the processor type may be different between the machines. We define autonomous to mean that each machine has control over its own resources rather than being in a pool of processors. Because of autonomy all machines of the same type may not provide the same computations or may limit services to particular users; because of heterogeneity the same computation may not be named or invoked the same way on all machines. However, the user of a distributed system would like to specify a computation for execution without worrying about the details of how or where the computation is done.
Much related work on the execution of computations in a distributed environment has been done. Many systems have extended the idea of command execution, where a new process is created to perform each computation, from a single-machine to distributed environment. UNIX is such a single-machine system, while Locus [BP84, WPE+83] and the V-System [Che88, TLC85, VDG86] are examples of systems that have extended the command execution model to a homogeneous, distributed system.

Other work has used the mechanism of remote procedure call (RPC) to extend the simple concept of a procedure call to a distributed environment. When compiled into a program, an RPC allows access to computations on remote machines. Birrell and Nelson [BN84] implemented one of the first RPC mechanisms for the Cedar programming environment. In contrast to a process-oriented model, systems such as EDEN [ABLN85] and ARGUS [LS83] have extended the object-oriented model introduced in Smalltalk [GR83] to a distributed environment. In these systems each entity is an object located on some machine in the distributed environment that has a set of procedures associated with it that can be invoked to operate on the object.

1.3 The Thesis

Our central thesis is this:

The actions available to the user in a distributed system should be treated as services, where a service is a user-level action that is offered by one or more machines. Our approach is to separate the identification of a service from its execution. This abstraction allows the details of performing a service to be hidden from the user, and allows the user to specify what services he would like to use and not be concerned with where or how the services are invoked.

As a first step we identify the essential components necessary to execute a computation in a distributed environment. The resultant model expresses the underlying structure for executing a computation in terms of a set of orthogonal components that each perform a specific task. For the model we denote a named computation as an action and term the model the action execution model. Using the action execution model we have a common representation for studying how actions, such as commands, network services, and remote procedure calls, are executed in a variety of distributed environments.

In addition, we apply the execution model to principles that follow from our thesis statement to develop a framework for the design of a service execution mechanism. Our service execution mechanism provides the user consistent and uniform access to computations from multiple machines in a distributed environment. Using the service mechanism, the user has access to new services as they become available without learning any new syntax or protocols to do so. Any new machines added to the computing engine are reflected to the user in only two ways: (1) as an increase in the number of available services; or (2) faster response for old services that are now available on more or faster machines.

A service execution mechanism design that provides the user a transparent interface to network services must address the following issues:
• Naming. How should services be named? Does service naming allow the user to specify partial names and if so, how are these partial names resolved to complete names?

• Location. How is service information located in the computing engine to determine which services are available and which machines offer these services? How is this issue affected by the scale of the computing engine?

• Selection. If multiple machines provide a service then how is one of the machines selected to perform the service?

• Invocation. Once a machine is selected to perform a service how is that service invoked? What name is used to invoke the service? How is the user authenticated for permission to invoke the service? Does the invoked service share the same execution environment as the invoking process for access to named objects?

1.4 Outline of Dissertation

In Chapter 2 we develop the action execution model for executing actions in a distributed environment and define much of the terminology used in this dissertation. Our definitions are reproduced in the glossary. We identify the orthogonal components that comprise the model and discuss the implications of different orderings and binding times for the components.

In Chapter 3 we use the action execution model as a basis for examining related work. We apply the model to specific systems for distributed computing to characterize these systems in terms of the model. In addition, we examine other work as it applies to each component of the model.

In Chapter 4 we state the set of principles that our service execution mechanism should adhere to based on our design philosophy and apply the action execution model to these principles to derive the ordering for its components. We then describe the details for each component of the service execution mechanism design.

In Chapter 5 we examine an issue raised in the presentation of the service execution mechanism of how to store and access information available in the computing engine. We present techniques for managing information, and give criteria for evaluating the techniques based on the scale and parameters of a particular problem. We then derive specific parameters for the problems of locating service information and machine load information and evaluate the best technique for the management of this information.

In Chapter 6 we describe the implementation of two service execution mechanism prototypes and the experience gained from these prototypes. They reflect an initial and final version of our service execution design.

In Chapter 7 we review the results of our research on the use of a service execution mechanism for transparent execution of computations in a distributed environment. We conclude with a discussion of future directions for our work and a summary of our research contributions.
2. ACTION EXECUTION MODEL

2.1 Introduction

We view a distributed environment as a collection of objects that are dispersed across multiple machines interconnected by a network. Objects can either be passive or active. Passive objects have work performed on them, and include files, devices, and mailboxes. Active objects, such as processes, perform work and manipulate the passive objects in the system. We refer to the work done by processes as computations and use the term action to represent a named computation that can be specified for execution.

In our model of a distributed environment we identify two types of machines. Server machines invoke actions based on requests from client machines. Client machines execute actions for users interacting with the machine, and request actions to be invoked by server machines on behalf of the user. A machine may be both a client and a server machine, thus supporting user interaction and executing actions for client machines.

Many types of actions may be executed in a distributed environment. For example, a user can specify, using a command interpreter, that a command be executed, which causes a process to be created with its binary image loaded from a file. In another example, a process on one machine may execute a network service on another machine by contacting a process listening at a well-known port on that machine. In a third example, a computation on a client machine may initiate a call to a process on a server machine through a remote procedure call mechanism.

The common denominator between the execution of these different types of actions is the underlying steps needed to transform an action name into the invocation of the computation named by the action. In this chapter we identify the essential steps, which we call components, necessary to cause an action to be executed in a distributed environment. The components form an action execution model, which is a formal abstraction of the necessary steps that must be performed to execute an action in a distributed environment. We will argue that the components are orthogonal because how each component is done is conceptually independent of the other components, and that the components are complete because taken together the components make all decisions necessary to execute an action.

Our reasons for postulating such a model are twofold. First, the model provides a basis for comparing existing work in distributed systems. A system is characterized by when, where, and in what order the components are performed. Second, the model provides a framework for the design of any mechanism that executes actions in a distributed system. Specifically for our work, the model provides a framework for transparent execution of the actions available to a user in a distributed system.
2.2 Approach

In the development of an action execution model for a distributed environment we define an action as a named computation. Similar to Watson's [Wat81] definition of an identifier, we define an action name as a string of symbols, usually characters, used to refer to a specific computation. A set of action names that follow the same naming convention form an action name space. The execution of an action translates an action name into the invocation of the computation needed to carry out the action.

As a first step in understanding the problem, we propose a set of questions that need to be answered about any action that is performed within a distributed system. How, when, and where these questions are answered in the course of executing an action both characterize a system and influence the interface presented to its users. For any action, the questions that must be answered are:

- **When** should this action be executed?
- **What** is the action to be executed?
- **Where** should the action be invoked for execution?
- **How** is the action going to be invoked for execution?

This list of questions arises from observation of existing systems and from thinking about what decisions must be made to execute an action in a distributed system. These questions compare favorably with work done by Shoch [Sho78] concerning naming in computer networks. In his work, Shoch distinguishes between names, addresses, and routes as follows:

The name of a resource indicates what we seek,

an address indicates where it is, and

a route tells us how to get there.

Shoch's model decomposes the steps of a name resolution system, which he applies to two examples: the telephone system, and interconnected computer networks. In a similar fashion, our work decomposes the steps needed to execute an action in a distributed environment. Using similar notation as Shoch we have:

The specification of an action indicates when to execute,
the name of an action indicates what to execute,
an address indicates where to invoke it, and
a means tells us how to invoke it.

As seen, our notation is similar with two exceptions. First, we include the question when, given by a specification, as part of our model. The specification is a mapping between a condition and an action name to execute when the condition is true. Second, how is a means for invocation rather than a route. This distinction is analogous to traveling from one city to another. For the question "how did you travel?", the answer could either be the
means of transportation you used or the physical route that you took. The question must be answered in context and for invoking an action the means is the appropriate answer.

The four questions we pose identify the decisions that must be made. The visibility of these decisions varies between systems—if only one means exists to invoke an action then how an action is invoked is not an issue, but if many machines can invoke an action then selection of a machine is an important component. What is important to realize is that whether the decision is explicit or implicit, each of these questions must be bound to an answer. The decisions may be made at system design time, at the time of specifying an action, or in the course of executing the action. Whenever the decisions, they are made by the components of the model.

In the remaining portion of this chapter we identify the essential components for executing an action in a distributed system. When, where, and in what order these components are performed, characterizes the action execution mechanism for a distributed system.

2.3 The Initiation Component

As the name implies, the initiation component, denoted as INITIATE, initiates an action for execution. In terms of the decisions that must be made, the initiation component determines what and when actions should be executed. Actions are specified for execution by giving an action name and a condition for when this action should be executed. When the condition is true the action is executed.

In formal terms, initiation is controlled by a set of specifications $\Sigma$. A specification $\sigma \in \Sigma$ is a mapping between a condition $\gamma \in \Gamma$ and an action $\nu \in N$ where $\Gamma$ is the set of conditions that can be true and $N$ is the action name space. For the specification $\gamma_i \rightarrow \nu_i$, the initiation component presents $\nu_i$ for execution if the condition $\gamma_i$ is true. In general, a specification can contain compound conditions and actions that allow an action to be executed based on multiple conditions, and multiple actions to be executed based on a single condition. This component is expressed as:

\[
\text{INITIATE : } \Gamma \xrightarrow{\Sigma} N
\]

Simple examples of specifications are for a programmer to specify an exception handling routine to handle the occurrence of an exception in his program, or for a user to specify commands for execution using a command interpreter. In the latter case, we view the command not just as an action, but as a specification with a nil precondition so execution is immediate. The command interpreter is an example of a user interface, which allows the user to form specifications and initiate actions for execution.

2.4 The Binding Components

The initiation component introduces an action into the system for execution then the binding components make the decisions concerning how and where the action will be invoked. In addition, the action name presented for execution may not be unique within the action name space so an absolute name must be determined. The binding components
are so named because they bind the decisions necessary to invoke an action once it has been initiated.

The components that perform the necessary bindings are resolution, protocol, mapping, location, and selection. They make the decisions about what unique action name to use, as well as how and where to invoke the action. The timing, location, and the order in which these bindings are made characterize a particular system. In this section we look at the details of each component and express fundamental ordering relations that must hold between the components.

2.4.1 Resolution

The name \( v \in N \) presented for execution by the initiation component identifies the action to execute, but does not necessarily identify a unique action name, corresponding to a single action, within the action name space. The set of action names \( \Omega \subseteq N \) that each map to a unique action are denoted as absolute, complete, or unique names. The use of partial, non-absolute names \( v \in N, v \notin \Omega \) arises in systems that allow shorthand forms for names, or multiple versions of an action. In such cases a name resolution mechanism is needed to translate a partial name into a complete name in the action name space. In our model, this translation is performed by the resolution component, denoted as RESOLVE.

The relationship between the initiation and resolution components determines an ordering relation between the two components if partial names are allowed. An ordering relation, denoted by the symbol ":-, is defined between two components if one component must always precede the other. Thus for any system with \( \Omega \neq N \) the ordering relation

\[
\text{INITIATE} \,< \, \text{RESOLVE}
\]

is defined. This relation indicates that if partial action names can be specified then the resolution component must follow the initiation component to resolve the name into an absolute name.

Resolution of a partial name to a complete name is performed by applying contexts to the partial name. As defined in [CP86], each context defines an interpretation of a set of names, and each "context/name" pair forms a qualified name. In general, each qualified name may be another partial name that needs to be further resolved with another context, or it may be a complete action name that causes the resolution mechanism to terminate. For our work we consider a more limited definition where only one set of contexts is used and each qualified name is either a complete name known to the system or the complete name is not known and resolution fails for that context.

We let \( C \) denote the set of available contexts for name resolution, and let each context \( c_i \in C \) be a partial function that performs the mapping

\[
c_i : N \longrightarrow \Psi
\]

where \( \Psi \) represents the set of all qualified names. Each \( \psi \in \Psi \) is a pair \(< c_i, v >\) for \( c_i \in C \) and \( v \in N \). For our work, we define \( C \) to be the set of contexts \( \{c_1, c_2, \ldots, c_n\} \).

Resolution can be either directed or inclusive. For directed resolution, successive contexts are applied to a partial name until a single absolute name is found. The partial
function for directed resolution can be defined as follows

\[
\text{RESOLVE}(i, \nu) = \begin{cases} 
\nu & \text{if } \nu \in \Omega \\
\psi & \text{if } \psi := \langle c_i, \nu > \in \Omega \\
\bot & \text{if } i = n \\
\text{RESOLVE}(i + 1, \nu) & \text{otherwise}
\end{cases}
\]

If \( \nu \) is already an absolute name that is known to the system, or a valid context can be found then \( \text{RESOLVE}(1, \nu) \) returns an absolute name, otherwise it fails (denoted by \( \bot \)). In contrast, with inclusive resolution all contexts are applied and the result is the set of all valid absolute names that are found. For either directed or inclusive resolution an absolute name could also represent a list of equivalent actions, much like a directory represents a list of files. Resolution of such action names must include expansion of these names to their equivalent set of actions.

Two common types of resolution mechanisms in operating systems are a current working directory and a search path. A current working directory is a single context that is used to resolve file names in a hierarchical file system. A search path is used in conjunction with directed resolution to produce an absolute name from a partial name and a set of contexts. In contrast, inclusive resolution is often used to find a set of objects that share a common property, such as file names that contain a common suffix.

2.4.2 Protocol and Mapping

The protocol and mapping components determine how an action is invoked. These components determine the invocation protocol needed to invoke the action and the underlying name to use for the action given this protocol.

An invocation protocol is a set of conventions for communicating with an existing process or creating a new process to carry out the computation defined by an action. For example, one invocation protocol is to create a new process with a binary image loaded from a file. Another example of a protocol is to contact a process on another machine using a network protocol and make a request for an action to be carried out. We denote the set of invocation protocols as \( \Pi \) and a protocol in this set as \( \pi \in \Pi \). The protocol component, denoted as PROTOCOL, chooses the protocol to use for a particular action, although in many systems this decision is fixed because all actions are invoked in the same manner.

The underlying name is the name needed by the invocation protocol to invoke the action. Because the underlying name is dependent on the protocol we denote the underlying name space as \( \Upsilon \) and a name in this name space as \( v_\pi \in \Upsilon_\pi \). The underlying name for an action \( v_\pi \) should not be confused with the specified action name \( \nu \). The underlying name is a low-level name that is used for invocation. The action name is a high-level name that is specified by the user. The notion of two name spaces is supported by Watson's [Wat81] conjecture that naming systems should "support at least two levels of identifiers, one convenient for people and one convenient for machines." Our identification of an invocation protocol and an underlying name within this protocol for invoking an action is similar to ideas proposed by Lantz, Edighoffer, and Hitson [LEH85]. They identify \( \text{medium name, identifier-in-medium} \) pairs for making requests to servers to access services.
High-level names are mapped into low-level names by the mapping component, denoted as MAP. Formally, the mapping component transforms a unique high-level action name \( \omega \in \Omega \) to a low-level name \( v_{\pi} \in \Upsilon_{\pi} \). Confusion between the two names is possible because many systems use the same name space for high and low-level names. For example, in UNIX, a file name is both the name of a command, and the name of the file needed to load into memory to invoke the command. In such cases \( \Omega \equiv \Upsilon_{\pi} \) and we view the mapping as a static identity function. If the two name spaces are not the same then

\[
\text{RESOLVE} < \text{MAP}
\]

because the mapping component uses an absolute name produced by the resolution component. The relationship between the mapping and protocol components also defines the obvious ordering relation

\[
\text{PROTOCOL} < \text{MAP}.
\]

### 2.4.3 Location and Selection

The location and selection components determine where an action is invoked. These components locate the set of server machines that are available to invoke the action, and select a machine from this set to actually carry out the action.

If the set of server machines is given as \( S \) then the location component, denoted as LOCATE, finds the set of server machines \( L \subseteq S \) that are eligible to invoke the action. For example, if all the server machines are identical then \( L \) may be equivalent to \( S \). In other systems, the set of eligible server machines may vary for each action. The client machine itself may also invoke the action in which case it is included in \( L \).

The selection component, denoted as SELECT, selects a server machine \( s \in L \) to invoke the action. The component is trivial if only one machine is in \( L \). If \( L \) contains multiple machines then the selection policy may be trivial by randomly selecting a machine or more complex by gathering state information about the server machines to select the best server in terms of performance or other criteria. Again, the relationship between the two components defines the ordering relation

\[
\text{LOCATE} < \text{SELECT}.
\]

### 2.5 The Invocation Component

The invocation component, denoted as INVOKE, is always the last component performed. Once the binding components have determined how and where a particular action should be invoked, the invocation component causes the action to be carried out. The binding and invocation components represent the system interface, which executes an action by translating an action name into an invoked action. To invoke an action, the invocation component uses three values that are supplied by the binding components. It uses

- the invocation protocol \( \pi \in \Pi \) determined by the protocol component,
- the underlying name \( v_{\pi} \in \Upsilon_{\pi} \) determined by the mapping component, and
• the server machine $s \in S$ determined by the selection component.

These three values determine how and where an action is invoked. In addition, two other issues are related to the invocation of an action. They are authentication and the invocation of an action within an execution environment.

2.5.1 Authentication

Authentication is the act of determining if the initiator, such as a user, of an action has permission to invoke the action. One method of authentication is to associate a permission list with each action. This method requires that each initiator be assigned a unique identifier and if this identifier is on the permission list of the action then it can be invoked. In other cases an action may require no authentication for invocation, or it may require explicit authentication from the user of the action.

Although authentication is usually performed at invocation, it may also be performed by other components to ensure that invocation does not fail. For example, the resolution component may use authentication to not just resolve an absolute action name, but to resolve an absolute action name that the initiator has permission to use. Another example is the use of authentication in the location component. Rather than locate all server machines that can invoke an action, the component may locate all such machines that the initiator has permission to use.

Another approach to authentication is to check the permission of the initiator in the mapping component. The action name is only mapped to the underlying name needed for invocation if the initiator has permission. Knowledge of the underlying name by the initiator implies the right to invoke an action. This approach is similar to the use of capabilities. Capabilities are a set of access rights for an object and the permitted actions on the object [MT86]. An action can only be invoked on an object if the initiator has the capability for the object and the action.

2.5.2 Execution Environment

An execution environment is the set of bindings between names and values used by an action during execution. At invocation, an execution environment is established for the newly created action. Many systems have the notion that the environment is inherited from the invoker of the action. Inheritance means that the execution environment of the process invoking the action is established for the process carrying out the invoked action.

The execution environment consists of three kinds of bindings between names and values:

• **Parameters** are names known to the invoked action and the values are bound by the invoker. For example, parameters are used to establish values for a procedure call or to give flags or options for a command.

• **Local bindings** are specific to an invoked action and initialized at invocation time. For example, each process in UNIX has its own set of environment variables, which are local bindings between named strings and string values.
• Global bindings are the names and values shared between all actions in a system. For example, the same file name space is often by all processes in a system.

The principal question concerning the execution environment is how much of the environment can be shared between the process invoking an action and the process carrying out the action. If the action expects to manipulate named files then the action can only be invoked on machines that share the same global bindings. In other cases, such as a procedure call paradigm with non-shared memory, all values are established with parameters so global bindings do not have to be shared.

2.6 Analysis of the Model

In the previous sections we identified the components of our model and stated a set of ordering relations that must hold between the components. In this section we define the possible binding times for components and where each component can be performed. We define additional terminology based on the binding time and ordering relations of the components. We also argue that the components of our model are both orthogonal and complete.

2.6.1 Component Binding Times

The binding time for a component is when that component is bound to the decision it makes. In [OD83], three strategies are given for when to bind a name to a value. We use these same binding times and define what they mean in the context of the components of our model. For a given action, the decision made by each of the components is made at one of three binding times.

• Static binding is done at system generation time, which means the decision is the same for every action that is executed within the system.

• Early binding is done at session initialization or action specification time, which means the decision is made by the initiator of an action.

• Late binding is done at action execution time, which means the decision is made dynamically after the action has been initiated and execution has begun.

Figure 2.1 shows the components, the values bound by each component, and the possible binding times for each. As shown, the binding time of the initiation and invocation components are fixed. As previously stated, the initiation component uses a specification to initiate execution with an action name $v \in N$. Invocation is always the last component because it uses the invocation protocol $\pi \in \Pi$, underlying name $v_\pi \in T_\pi$, and server machine $s \in S$ determined by the binding components to invoke the action.

The timing and the ordering of the binding components, however, does vary between systems and characterizes a system. In Figure 2.1 the dashed boxes represent positions for each of the five binding components (shown to the right in no particular order) to be placed within the model subject to the ordering constraints previously given. The dashed
Static binding is used for components that have a fixed decision. For example, in UNIX all commands are invoked in the same manner. Early binding is used when the initiator needs to make an explicit decision, such as which server machine to select for action invocation. Late binding is used when the decision can be put off until execution. For example, the resolution of a partial name to a unique command name is performed at execution time in the UNIX system.

The binding time of a component is a tradeoff between flexibility and performance. The later a component is bound to a value, the more easily the system can adapt to change. However, each decision that is made at execution time incurs additional run-time costs, although some decisions made at execution time, such as where to invoke, may actually increase the performance by finding a better machine to execute the action. The binding times also influence a user's perception of the system. Obviously, if the decisions are made statically, the decisions are fixed for the user. If the decision is made at specification time, then the user must be explicitly aware of the decision. Only if the decision is made dynamically is it transparent to the user.

2.6.2 Component Binding Location

Where a component is performed to make a decision also characterizes a system. For an action, each component can be performed in one of three places:

- at a client machine, which initiates the action,
- at a server machine, which invokes the action, or
- at a client machine, which initiates the action,
• at an intermediary machine, which neither initiates nor invokes the action.

The initiation component is always performed by the client machine, and the invocation component is always performed by the server machine. The other components may be performed by any of the three machines given above. For example, the client machine may also resolve the action to a complete name, but an intermediate machine may contain a central database that is consulted to locate which machines provide the action. The server machine may map the action name into its underlying name for invocation.

Where a component is performed is determined by where information needed to make the decision is available. Decisions made by the client machine do not require network overhead, but the necessary information may not be available. An intermediary machine allows information to be centralized, but adds another machine to the task of executing an action. Decisions made by the server machine are transparent to the client, but these decisions can only be made once the server machine has been selected.

2.6.3 Transparency and the Model Components

Given the ordering relations and binding times of the model, we can define different forms of transparency as they relate to action execution for distributed systems.

*Location transparency* denotes a system in which the names for actions are independent of where the action is invoked. Systems that are location transparent exhibit the characteristics that \( \text{RESOLVE} < \text{SELECT} \) and \( \text{SELECT} \) is bound at action execution time. The relation ensures the absolute name for the action is determined prior to the selection of a server machine. The binding time ensures the server machine is not fixed or selected by the initiator.

*Access transparency* denotes a system in which all actions are initiated in the same manner whether invoked on the client or a server machine. This condition is true if the choice of an invocation protocol for invoking an action is not made at specification time. The invocation protocol is either static or chosen after the action has been initiated.

In light of the model, another type of transparency that we introduce is *protocol transparency*. A system is protocol transparent if \( \text{RESOLVE} < \text{PROTOCOL} \) and the protocol is chosen at execution time. Similar to location transparency, the ordering relation and binding time requirement ensure that the name of the action is independent of how the action is invoked.

Finally, *network transparency* means that an invoked action executes in the same manner regardless of where it is invoked. This condition is met if the portions of the execution environment used by the action are the same irrespective of which server machine is selected to carry out the action. This situation requires that the global name bindings be shared between the client and server machines or the invoked action does not use names from the global bindings.

2.6.4 Orthogonality and Completeness of the Model Components

We conclude our analysis of the model with an argument for its validity. In the opening we stated that the model components were both complete and orthogonal, which implies
our model covers all tasks needed to execute an action, and no components duplicate function.

If our model is not complete then the execution of some action in a distributed environment cannot be completely modeled with the components we have given. Although we cannot prove that our model suffices for all systems we do show in Chapter 3 that it can be applied to action execution in many types of systems. In addition, the questions on which the model is based are similar to Shoch’s well-known naming model. This similarity lends support that the model is based on the correct set of questions.

If the model components are not orthogonal then the functionality of one component can be expressed in terms of some combination of the other components. Because the result produced by each component is distinct, the components themselves are distinct. Some dependence does exist between the components due to the ordering relations, but the relations do not imply how any particular component is performed. Functional overlap does occur if authentication is performed by more than one component, but authentication must only be performed once and any other checks are for efficiency to ensure that authentication does not fail at invocation time.

2.7 Summary

In this chapter we have presented a model for the execution of actions in a distributed system. We have identified the components necessary to execute an action within a system, defined the functionality of each component, and showed different types of transparency based upon the ordering and binding time of these components. We have concluded with an argument that the model components are orthogonal and complete.

In the following chapter we use the model to compare a variety of mechanisms, related to our work, that execute actions within a distributed system, such as command execution, remote procedure calls, and network services. The model provides a common basis for comparing these different types of mechanisms.

In Chapter 4 we use the model we have developed as a basis for the design of a service execution mechanism. The model provides a framework for the components that our mechanism must provide, and provides a means to compare our design with existing work.
3. APPLICATION OF THE MODEL TO EXISTING WORK

In this chapter we discuss existing work in distributed systems, specifically distributed computation, using the action execution model developed in the previous chapter as a basis. The body of work is large, and encompasses a number of ideas on how to execute actions in a distributed environment. The purpose of this chapter is threefold: to examine existing work by grouping systems that exhibit similar characteristics; to apply the action execution model to specific distributed systems; and, to examine additional ideas from existing work that are applicable to specific components of the model.

3.1 Command Execution

A number of distributed systems provide their users the capability to execute actions, called commands, using a user interface such as a command interpreter. A command is invoked for execution by creating a new process and loading the image for the process with the contents of a binary file. This style of executing tasks was first used in single machine computing systems and the same principle has been extended to distributed systems. In this section we examine systems that employ a command-oriented model for execution of tasks.

3.1.1 UNIX

Although not originally developed as a distributed system, the UNIX operating system [RT74] exemplifies a system that is command-oriented, and one that has been used as a basis for much work in distributed systems. To characterize the system we apply our model to a familiar version of the operating system, UNIX 4.3BSD [UNI86]. This version is presently used in many production computing environments. UNIX was designed as a single machine system, but now includes explicit commands for accessing files and executing commands on remote machines. For our discussion we assume the user interface is a command interpreter, or shell in UNIX terms, that allows users to specify commands for immediate execution. The model components and their binding times needed to execute the command `tex` are given in Figure 3.1.

As shown, most of the decisions are fixed for all commands. Because commands are invoked on the local machine the location and selection components are trivial. All commands are invoked by creating a process loaded with the contents of a binary file. Because commands are named the same as files, the name space presented to the user for specifying commands is the same as the underlying name space for invoking commands.

Resolution is performed at execution time by the shell. This function alleviates the user from having to specify a complete file name for each command. Resolution uses a
search path, maintained as a local binding for each process, which specifies an ordered list of directories to search when a partially specified command name is given. The search path is iterated over until a valid directory/name pair is found. A pair is valid if the file is executable and the user is authorized to execute the file. In the example, tex has been resolved to the complete file name /bin/tex.

A set of user, group, and world permission bits is used in UNIX to determine the execution access rights for each file. These bits are checked whenever a command is invoked by the operating system, and, to ensure that a valid command is resolved, also by the shell. Thus, commands initiated from the shell are actually authenticated twice in UNIX.

As part of invocation, the newly created process inherits the execution environment and the user identification from the parent process, in this case the shell. This environment includes the current working directory, a context in the file name space, and a set of environment variables, which are local bindings between named strings and string values. The new process also inherits a set of parameters that include textual arguments to the command and a set of standard byte stream descriptors used by the command to read input, and write output and error data. In a single machine environment the global bindings, such as file names, are implicitly inherited.

Of interest to our work is how command-oriented systems have been extended to a distributed environment. In the case of UNIX, a local command rsh is used to invoke commands on remote UNIX machines. This command requires the user to specify the remote machine as a parameter to the command, and authenticates that the user has an account on this machine. Resolution is performed on the remote machine using the search path environment variable from the execution environment that is established on the remote machine from a startup file. Thus, remote command execution in UNIX is neither access, location, nor network transparent for the user.
3.1.2 Locus

In contrast to UNIX, the Locus distributed operating system was designed and built specifically to provide its users a network transparent UNIX operating system [WPE*83]. The system provides a network-wide file system and distributed process execution over a set of machines with heterogeneous architecture. Each machine runs the Locus operating system. Commands are specified by the user in the same fashion as a single machine UNIX system, but their execution is carried out in an access transparent manner [BP84]. A description of the Locus system in terms of our action execution model, again executing the command \texttt{tex} specified from a shell, is shown in Figure 3.2.

\begin{figure}[h]
\centering
\begin{verbatim}
static { IPROTOCOL \quad \pi = load file
early binding
\begin{align*}
\text{LOCATE} & \quad L = \{\text{setxsites()}) \\
\text{INITIATE} & \quad \nu = \text{tex} \\
\text{RESOLVE} & \quad \omega = /bin/tex \\
\text{SELECT} & \quad s = s_{vax} \\
\text{MAP} & \quad \upsilon = /bin/tex/vax \\
\text{INVOKE} & \quad \pi, \upsilon, s
\end{align*}
\end{verbatim}
\caption{Action Execution Model for Locus}
\end{figure}

Commands are invoked identical to UNIX except the newly created process may be on a remote machine. Resolution and authentication are also identical to UNIX. The key differences from UNIX are locating and selecting a machine for invocation and accommodating heterogeneous architectures.

Location of eligible machines to invoke a command is made by examining a set of preferred execution sites specified by the system call \texttt{setxsites()} (also available as a shell command). This primitive is usually invoked for the user at session initialization to specify the set of available machines. Selection of a machine for invocation involves keeping track of the load on each machine and selecting the least loaded eligible machine. Migration of processes between machines at runtime is an additional feature available in Locus that can be used for load balancing.

To accommodate heterogeneous architectures, Locus uses a hidden directory, which is a special directory that contains binary files for specific machine architectures. A command is resolved to a complete name (actually a hidden directory), a machine is then selected for invocation, and the command name is mapped to a machine-specific binary file located in the underlying hidden directory. In the example of Figure 3.2, a VAX architecture machine has been selected for invocation so the associated binary file is mapped for invocation.
In summary, Locus uses a two-level name space where the high-level names appear to the user as normal UNIX file names, and the hidden low-level names include a "tag" that is architecture specific. In the case where all machines in the distributed system are of the same type a two-level name space is not needed, and the same executable file is used by all machines. The Locus distributed system provides its users access transparent command execution over a homogeneous operating system environment. Although not location transparent because the user must specify the set of eligible server machines, the specification of this set is often hidden at session initialization.

3.1.3 Other Command Execution Work

The V operating system was developed to support the use of diskless personal workstations in a local network environment [Che88,TLC85,VDG86]. Within the V-System, well-known servers, such as storage, authentication, pipe, and internet servers, manage resources. Users of the V-System specify commands using an *executive*, which is a command interpreter. The distributed environment is network transparent, but by default all commands are executed on the user's workstation. Explicit facilities are available to the user for executing commands on another machine in the V-System; users can either specify the remote machine or let the system select a suitable machine. Like Locus, the V-System supports heterogeneous architectures by appending an architecture specific tag to each binary file. However the tag is a suffix on the file name rather than a hidden directory.

The Process Server [Hag86] is part of work on the Cedar programming environment [Tei84] that looks at performing computations from a workstation that cannot be conveniently handled by the workstation. The system uses a centralized server to manage a set of commands, grouped into *packages*, that are available from a *cluster* of server machines. A client workstation requests execution of a command by contacting the central server, which selects an appropriate server machine to invoke the command, and passes the selection back to the client. The client then contacts the server machine using a fixed protocol to invoke the command. During execution of the command, the server uses the client to do file name resolution, provide some file operations, and answer questions about the user's execution environment.

The Saguaro distributed operating system is designed for a collection of processors connected by a local-area network [ASHP87]. Its notable features for command execution are a type system for describing command arguments, and the use of *channels* for connecting communicating processes. Channels are an interprocess communication and synchronization mechanism whose functionality is a superset of UNIX pipes. Each command has an associated template to specify the type and default values of arguments. Commands are invoked by command servers, which maintain the execution environment for the command. Commands are invoked on the local machine unless the predicted load would exceed a predetermined threshold, in which case a lightly loaded machine is found for invocation.
3.1.4 Summary

These command execution systems are characterized by an invocation protocol that invokes commands by creating a process loaded with the contents of a binary file. The use of this protocol leads to naming commands as files with a resolution mechanism to alleviate the user from having to specify complete file names. The Locus and V systems have added tags to executable file names to support heterogeneous architectures.

The systems are network transparent, except for UNIX, because all provide support for interpreting names consistently across machines, principally through a distributed file system. As opposed to systems based on a processor pool [NH82], machines in these systems are autonomous. Autonomy ranges from each machine being able to refuse the invocation of a process to being able to control exactly what commands it offers.

3.2 Network Services

Network services are a set of standard services that have been defined for the DARPA Internet community [RP86]. Each service has its own official service name, port number, and protocol for invocation based on the transport protocols UDP [Pos80] and TCP [Pos81]. A variety of services exist:

- mail and name services, which are not used directly by users, but support frequently used operations on the Internet,
- file transfer and remote login services, which are directly available to users, but require specialized protocols for invocation, and
- time, system status, and quote of the day services, which are available to users either by sending and receiving a UDP packet or making a TCP connection with the appropriate port.

We illustrate the application of the action execution model to network services by considering the last set of services given. We assume the existence of a command \texttt{tcpconnect} that takes as arguments a machine name and the service name to invoke. \texttt{Tcpconnect} makes a TCP connection to the port for that service on the given machine. Figure 3.3 illustrates our model for the specification "\texttt{tcpconnect host daytime}" where the \texttt{daytime} service returns the current time.

All service names are guaranteed to be unique through a central registry so resolution is not needed. Any machine in the Internet that implements the service can be selected for invocation. As shown, the user must determine how and where to invoke the service. At execution time, the \texttt{daytime} service is mapped to port 13 using a global database and that port is contacted on the specified machine.

Network services are a means for executing computations in a distributed environment, but require the client of the service, either a user or programmer, to specify where and how a service is to be invoked. In the example we illustrated a protocol that can be used

\footnote{The capability to specify an arbitrary machine and service name for a TCP connection is actually implemented as part of the \texttt{telnet} command in UNIX 4.3BSD.}
### 3.3 Remote Procedure Call

The remote procedure call (RPC) is a mechanism for one process to request that a procedure be executed by another process, most commonly on another machine. Remote procedures are invoked by suspending the calling environment, packaging the list of procedure parameters and sending them across the network to a server process that invokes the remote procedure. When the remote procedure finishes execution, the results are passed back to the calling procedure and execution resumes as if a normal procedure call had been made. In his doctoral dissertation, Nelson [NeI81] examines design possibilities for an RPC mechanism and references much of the previous work on RPC. Additional discussion of ideas in RPC mechanisms is given in [BLL*85] and an implementation of these ideas is discussed in [NBL*88].

A full-scale implementation of an RPC mechanism was made by Birrell and Nelson [BN84] at Xerox PARC as part of the Cedar programming environment. A model of their mechanism is shown in Figure 3.4 for the remote procedure $F$ that is part of the interface $FileAccess.Alpine$.

Procedures are organized into *interface modules*. Each interface is a list of procedure names, together with the types of their arguments and results. An RPC interface is described by a *type* and an *instance*. A type is the absolute name for the interface, thus resolution is not needed, while an instance corresponds to a server machine that exports the interface. In the example, the interface type $FileAccess.Alpine$ has been specified,
Figure 3.4 Action Execution Model for Birrell and Nelson's RPC

and a machine must be selected that exports an instance of the interface. The Grapevine distributed database [BLNS82] is used to locate all instances of the interface and the most responsive server machine is selected for RPC binding. Binding is completed by performing any required authentication on the caller, and then mapping the exported interface to a unique identifier and an index for the procedure within the interface. A stub routine, which is a local procedure that handles the RPC communications protocol, invokes the procedure using the unique identifier of the interface and the procedure index. In the example, \( F \) is the third procedure in the interface.

The RPC mechanism is access and location transparent—the programmer specifies all procedures (remote and local) in the same manner and the specific server machine for each interface does not need to be given. Another characteristic of RPC mechanisms is that communication between the client and server is through the parameters. Unlike a programming language where procedures can share data through global data structures in the invoker’s address space, the remote procedure does not have access to the same address space. Data must be shared through parameters or through shared global data objects such as files. A problem with data sharing through parameters is representation of the data between heterogeneous architecture machines. The eXternal Data Representation [SMI86] and Network Data Representation [DLM*87] formats are two examples of protocols for standard network data representations.

3.4 Object-Oriented Systems

Data abstraction is a popular method in programming language research to group related data and operations into a single module. Smalltalk [GR83], a data abstraction language, termed these modules as objects. Just as procedure-oriented languages have motivated research into distributed computing in the form of remote procedure calls,
object-oriented languages have motivated research into object-oriented distributed systems. Because procedures share many characteristics with data abstraction languages, it is not surprising that the execution of actions in object-oriented distributed systems share many characteristics with remote procedure mechanisms. Rather than associating procedures with an interface, procedures are associated with an object. This similarity is evident when comparing the action execution model for the object-based system Eden [ABLN85] shown in Figure 3.5, with the model for Birrell and Nelson's RPC shown in Figure 3.4.

- **Static binding**
  - `PROTOCOL` \( \pi = \) invoke Eject
  - `RESOLVE` \( N \equiv \Omega \)

- **Early binding**
  - `INITIATE` \( \nu = \text{MailBox.Deliver} \)
  - `LOCATE` \( L = \{s_{\text{Eject}}\} \)
  - `SELECT` \( s = s_{\text{Eject}} \)
  - `MAP` \( u_s = \text{inv. proc. Deliver} \)
  - `INVOKE` \( \pi, u_s, s \)

**Figure 3.5 Action Execution Model for Eden**

The Eden system is composed of objects called *Eden objects*, or *Ejects*. Each Eject has a data part and a set of invocation procedures that can be invoked by other Ejects (all computations are done by one Eject procedure invoking another Eject procedure). Ejects are relatively large, compared to objects in Smalltalk, on the granularity of directories or mailboxes. In the example taken from the mail system Edmas [AH87], `MailBox.Deliver` is a specification of the `Deliver` procedure for a `MailBox` Eject that allows a mail message Eject to deliver itself to a mailbox. At execution time the Eden kernel locates the server machine that contains the target `MailBox` Eject. In Eden each Eject resides at one machine, although Ejects are mobile and may move between machines over time. Once the target Eject has been located, the Eden kernel authenticates that the message Eject has the capability to use this mailbox and the `MailBox` Eject maps `Deliver` to its underlying invocation procedure within the Eject. Capabilities for mailboxes are established through invocation of a lookup procedure in a central directory Eject.

Many other object-oriented distributed systems exist. Among the more well-known are Amoeba [MT86], Apollo [DLM*87], Argus [Lis88], Clouds [DLS85], and Cronus [STB86]. The distinguishing characteristic of these object-based systems is data abstraction and restricting access to the underlying object through a well-defined set of procedures. The
user of an object must hold a capability for an object to invoke a procedure defined by the object. Distributed, object-oriented systems are location, access, and network transparent for the invoker of an object procedure.

3.5 Component Specific Work

Much work in distributed systems has been done that does not encompass the scope of our model, but is applicable to specific components of the model. In this section we examine additional work that is related to specific components of the model.

3.5.1 Initiation

Issues concerning the initiation component are equally applicable to a distributed system as they are to a single processor system. The interfaces for specification of an action, which are independent of where or how it is invoked, are similar in both environments. For command-oriented systems, commands are executed as they are specified by a user from a command interpreter. For programs, procedures (local or remote) are executed that have been specified by a programmer. Two directions for providing more flexibility in when actions are initiated are: the use of explicit "hooks" built into systems to allow actions to be initiated based on certain conditions; and, the construction of tools that allow users to automate when an action should be initiated.

A common example of a hook is an exception handler in a programming language that allows an action to be taken when an exception occurs. The extensible editor EMACS [Sta81] provides a number of hooks for the initiation of actions. For example, an abbreviation facility automatically expands an abbreviation or invokes a procedure based on a table of abbreviations specified by the user. Hooks called portals are used in distributed naming systems to allow arbitrary actions to be executed whenever a name is resolved [LEH85]. Portals can be used for monitoring operations, extended protection modes, or switching name contexts during resolution. Hooks to allow users to define and implement their own semantics for files have been implemented as Watchdogs [BP88] for the UNIX file system. These hooks can be used for alternate techniques for file naming, protection, and storage.

A commonly available condition for automatic execution of actions is time. Two examples of tools for automatic action execution based on this condition are at and usrcron in the UNIX environment [UNI86]. At allows commands to be scheduled for execution at specific times and usrcron allows commands to be scheduled on a regular time basis such as every day. A more general notion of specifying conditions for action execution is introduced in the tool Omicron [KW86b]. Not only is time a condition, but other system events such as a file changing or a user logging on can be checked and used as conditions for action execution. A powerful feature of Omicron is the capability to specify that actions be executed depending on a combination of events.
3.5.2 Resolution and Naming

Certainly one of the most important topics in distributed systems is naming of system objects and how an object is resolved given a partial name. Existing research in distributed systems has dealt with how to name such objects as mailboxes [BLNS82], files [BMR82, CM86b, TR84, WPE*83], hosts [Moc87], and services [Che88]. Other research has examined how to name general objects in distributed systems [CM86a, LEH85, NBL*88, OD83]. Many models of naming have been proposed [CP86, Sal78, Sho78, Wat81]. The sheer volume of literature is indicative of the importance of this topic.

It is impossible to fully summarize this work, but a few observations about important themes can be made.

- **Names are organized into a hierarchy.**
  In almost all of the work the names are organized into a hierarchy that spans a global name space. The hierarchies either have an unlimited number of levels without a specific structuring of the levels, or have a fixed number of levels with a specific meaning for each level. The UNIX file system [RT74] uses an unlimited depth hierarchy for naming files. The Clearinghouse name space [OD83] uses a fixed, three-level depth of the form L:D:O consisting of a local name, a domain, and an organization.

- **Names are controlled by name servers.**
  Hierarchical name spaces conveniently partition the name space so that portions can be controlled by different name servers. This approach allows decentralized control of the name space by distributed name servers [CM86a, WPE*83]. Use of unique prefixes for absolute names is a quick, and efficient way to map a named object to the server that manages the object [CM86a, Com87, W086]. A name server is either integrated with or segregated from the servers that actually implement the objects [LEH85]. The V-System uses an integrated approach so names for an object are managed by the same server that implements the object. Most systems separate the naming service from the objects that are being named.

- **Resolution is used to avoid use of long, absolute names.**
  Resolution maps a partial name into an absolute name using a context. As opposed to an alias mechanism where a partial name always maps into the same (usually longer) absolute name, resolution may not always resolve to the same absolute name. A commonly used resolution mechanism in a hierarchical name space is a current context, which is a node within the hierarchy that is used to interpret non-absolute, partial names. Another common mechanism for resolving a partial name is a list of contexts known as a search path. The resolution mechanism may be performed by a movement between servers in the distributed system [CP86, Ter85]. For example, the resolution of a mail address may require movement through many mail servers as the mail address is resolved to the correct machine address for delivery.

- **Properties can be associated with named objects as another form of resolution.**
  Many systems allow properties in the form of (attribute, value) pairs to be associated with objects. Objects can then be selected based on particular attributes [Cra83,
One issue with this approach is whether the attributes should be typed. Another issue is what to do if multiple objects are matched for a given set of attributes. Depending on the application, any or all of the matched objects may be satisfactory for use, or the attributes may be ambiguous if more than one object is matched.

- **A name may represent a set of names.**
  Normally each name is associated with a particular object. The notion of a *generic* or *group* name is used when a name refers to a set of names [CM86a, LEH85, OD83]. A generic name groups a set of objects together that are considered equivalent. When a generic name is encountered any or all of the named objects may be used. The inverse of generic names are *aliases*, which are alternate names for an object allowing multiple names to refer to the same object.

- **Actions can be associated with name resolution.**
  In the Universal Directory System [LEH85], actions can be associated with the resolution of a name. These actions are initiated as part of a portal mechanism that associates an action with a name. Portals can be used for initiating monitoring operations or switching contexts during resolution.

### 3.5.3 Location and Selection

Given the name of an object (or action), much work has been done to efficiently locate instances of that object within a distributed environment. In early distributed file systems, such as IBIS [TR84] and the Newcastle Connection [BMR82], the server name was an explicit part of the object name making location easy, although not transparent to the user. In systems where object names are location transparent many techniques have been employed for locating a named object.

In the Process Server [Hag86], a central server knows the location of all available computations and all requests are directed to that server. The V-System [Che88] uses a multicast request to locate possible server machines for executing a command. Cheriton and Mann [CM86a] describe a decentralized naming mechanism based on the use of a cache for locating the majority of objects and multicast when an object name is unknown in the local cache. Additional analytical work has been done to determine how different location techniques and naming conventions affect the cost of object location [MV85, Ter84].

Once the set of servers that provide an action is known, one of the servers must be selected for invocation. Selection is commonly used to find the server that will give the best performance among the set of available servers for executing an action. This goal leads to load balancing techniques on a set of server machines, which try to keep the load on each of the servers approximately the same. Many load balancing techniques have been proposed that involve algorithms based on assumptions of complete, up-to-date information about the system [CA82, WM85].

Eager, Lazowska, and Zahorjan [ELZ86] point out that such schemes can lead to high overhead costs and unpredictable results in the face of inaccurate information. They instead propose a much simpler scheme based on a *threshold* policy. If the load on the
local machine is below a certain threshold then that machine is used, otherwise remote machines are probed until one is found with a load less than the threshold. In their paper they make an analytical analysis of three variations of the threshold policy. The threshold policy was incorporated as part of the design of the Saguaro distributed system [ASHP87].

3.6 Summary

In this chapter we have presented existing work related to distributed computing and shown how the action execution model can be used as a framework to compare this work. The presentation of the work clearly shows that the bulk of research in distributed computation has been to extend one of three, well-known action execution models in a single machine environment—command-oriented, procedure-oriented, or object-oriented—to a distributed environment. The command-oriented systems provide distributed computation geared towards the user of a system, while the other two models are more geared towards the programmers in a system.

In the second portion of the chapter we have examined existing work as it relates to specific components. A considerable amount of work has been done in naming objects in a distributed system. Other work has looked at locating and selecting objects in a distributed environment. We plan to build on previous work as we examine a new approach to the use of actions in a distributed environment.
4. A NEW APPROACH

4.1 Introduction

In a computing engine, users reside at personal workstations, which serve as a user's interface to the computing engine providing local computing power, high-resolution display capabilities, multiple input devices, and mechanisms for access to resources on remote machines. The problem addressed by our research is how to provide users a transparent environment for the execution of actions in the computing engine.

We distinguish between the actions used by programmers and users in a distributed environment. Programmers use actions at the procedure or object level to construct distributed applications that require low-level data structures to be shared between a client program and a remote procedure. The interface between the client program and remote procedure is specific to each application. In contrast, users initiate an action from a high-level interface such as a shell. The input and output of the action are often strings and streams of characters, and the action works on large, globally accessible objects such as files. Many user-level actions share the same invocation protocol so the same client routine can be used to invoke many actions rather than needing a new routine for each available action. We believe transparent location and invocation of actions at the user-level is important in a distributed system because users want to use resources of the system without programming distributed applications.

One approach for transparent execution of user-level actions is taken by Locus, which extends the model used in single-machine systems of allowing users to specify executable files for execution. Because all actions are invoked in the same manner the user has easy access to any actions that are made available. However, this approach has three principal drawbacks in a distributed environment. First, it requires that all machines share the same file system. Second, it means that each action can only be invoked by creating a process with the contents of an executable file. Third, it overloads the name of the file to be both the name available to the user for specification, and the name (with an architecture specific tag) used for actually invoking the action.

We find this approach unsatisfactory because it does not distinguish between two key ideas: what an action does and how it is done. Instead, we investigate an approach that explicitly separates the name of an action from how and where it is invoked. As outlined in [Wil86], we use the term service to denote a user-level action whose identification is separated from its execution. The service name conveys to the user what the service does and the underlying service execution mechanism performs the steps necessary to execute the service.

Services are managed by a set of server processes in the computing engine. Each service is offered by one or more machines in the computing engine with each instantiation of a service known as a service instance. A service instance may be invoked in many ways.
For example, a process may be created on the local machine to execute a binary file, or a server process may be contacted on a remote machine that accepts a service name and performs the computation associated with that service.

Our approach for investigating service execution within a computing engine adheres to the following design principles:

1. **The user should only need to specify what service he would like to perform.**
   He does not need to be concerned with where or how it is invoked.

2. **The user should be able to specify partial names for services.**
   The appropriate absolute service name should be resolved. In addition, the user should be able to easily identify and switch use between different versions of a service.

3. **A service may have multiple instances, each offered by a different server machine.**
   Each server machine has control over what services it offers. The addition or deletion (except if it is the last) of a service instance should be transparent to the user.

4. **The implementation of a service instance is machine specific.**
   A service is defined by what it does, not how it is done. Thus the implementation of a service does not have to be the same for each machine offering a service.

Portions of this set of design principles have been incorporated into existing systems, but no system has provided the amount of abstraction that we propose. Through abstraction, we not only hide where a service is invoked, but also how it is invoked. The resultant service execution mechanism hides the details while providing users a uniform interface to the actions in a distributed environment. To better understand our approach we apply the action execution model presented in Chapter 2 to gain two important insights:

- a framework for the design of our service execution mechanism, and
- a basis for concrete comparison between our work and other work already characterized with the action execution model.

In the next section of this chapter we define the relative ordering and binding time for each component in our service execution mechanism. In succeeding sections we look at individual components and describe the details of the mechanism. In the summary we relate the service execution mechanism back to the model as was done for other systems.

### 4.2 Definition of the Service Execution Mechanism

The principles stated in the previous section serve as guidelines for our design, while the components of the action execution model identify the decisions that must be made in executing a service. Taken together they define how the pieces of the service execution mechanism should relate to each other to satisfy the design principles. This section does not describe the details of our mechanism, but instead defines the organization of its components.
The binding time and ordering of the initiation and invocation components are fixed. The initiation component INITIATE uses a specification from the user to initiate execution of a service. The invocation component INVOKE is the last component and uses the decisions made by the binding components to invoke the service. The binding components are placed in the model relative to these two components as described in the following.

4.2.1 Initial Ordering

The first design principle on page 29 states that the name of a service should be independent of where and how the service is invoked. In terms of the model, this name is the high-level name of the service available for specification by the user. Our second principle states that if a partial name is specified then it should be resolved to an absolute name in the high-level name space. Because partial service names are allowed, the resolution component of the model must be done after initiation (at execution time) and produces an absolute service name as the result. The absolute service name must be independent of where or how the service is invoked (by the first principle), so two order relations are implied that must hold between components of the service mechanism. These are

\[ \text{RESOLVE} < \text{PROTOCOL} \text{ and } \text{RESOLVE} < \text{SELECT} \]

The first relation states that resolution must come before an invocation protocol is chosen and implies that a service name must be independent of how the service is invoked. The second relation states that resolution must come before a server machine is selected and implies that a service name must be independent of where the service is invoked. In addition, the ordering PROTOCOL < MAP given in Chapter 2 must hold. This relation states that an invocation protocol must be chosen before an underlying name can be mapped because the underlying name is dependent on the protocol. Using this required ordering, along with the previous two orderings, produces the following three possibilities for ordering relations between the components.

\[ \text{RESOLVE} < \text{PROTOCOL} < \text{MAP} < \text{SELECT} \]
\[ \text{or} \]
\[ \text{RESOLVE} < \text{PROTOCOL} < \text{SELECT} < \text{MAP} \]
\[ \text{or} \]
\[ \text{RESOLVE} < \text{SELECT} < \text{PROTOCOL} < \text{MAP} \]

The possible relations show that the selection component can be placed in one of three positions relative to the choice of a protocol and the mapping to an underlying name. Selection of one of these orderings is dictated by our fourth design principle which states that how a service is provided should be determined independently by each server machine. In the first ordering above, selection follows the mapping to an underlying name, which requires that the same underlying name be used on each machine. In the second ordering, selection follows the choice of the protocol, which means a service must be invoked with the same invocation protocol on each machine, but possibly a different underlying name. In each of these cases the design principle is violated because each server machine must use either the same invocation protocol or underlying name for invoking a service. In
a heterogeneous computing environment each server machine should not be confined to offering a service in the same manner. Hence the third ordering, with selection preceding choice of a protocol, must be selected to satisfy our design principle.

4.2.2 Additional Ordering

With the ordering relations determined for four of the components, the fifth component, location, can be added to the ordering by applying a second required ordering from Chapter 2. The relation LOCATE < SELECT states that the set of machines offering a service must be located before a member of the set can be selected to provide the service. The application of this ordering produces two possibilities for inclusion of the location component.

LOCATE < RESOLVE < SELECT < PROTOCOL < MAP

or

RESOLVE < LOCATE < SELECT < PROTOCOL < MAP

Location must either precede or follow resolution. If it precedes resolution then the set of machines that offer a particular service must be known before the absolute service name is known. However, our third design principle states that each server machine has control over what services it offers. In order to determine which machines offer a particular service the name of the service must be known. Hence, the location component must follow resolution. Because the resolution component is determined at execution time, each of the other components must also be determined at execution time by virtue of the ordering, and the complete ordering for the components is given as follows.

INITIATE < RESOLVE < LOCATE < SELECT
< PROTOCOL < MAP < INVOKE

4.2.3 Summary

The straight-forward "generation" of the framework for our mechanism from a set of design principles is an appealing idea in distributed systems design. The result is not only a framework for our design, but a validation of the model in identifying the essential components for action execution. The resultant mechanism has a number of desirable features:

- It is well-suited for a heterogeneous environment because the underlying implementation for each service can be different, although services are specified in a uniform manner by the user.

- The mechanism adapts easily to different environments. If the user’s machine provides a large number of services then they can be selected. If the user’s local machine provides just a few services then services from other machines can be used.

- Any service, not just a select number of "system" services such as file or name services, may have multiple instances providing better reliability and increased performance on a per-service basis.
The structuring of the mechanism also raises a number of questions about its details. These details include how a service is named, how to guarantee that different service instances provide the same service, how to locate and select server machines providing a service, and how protocols are used to invoke services. In the following sections we sketch an outline of how the mechanism typically works then examine each component of the mechanism and explore answers to these and other questions.

4.3 Overview of the Service Execution Mechanism

Before proceeding with the details, we first give an overview of our service execution mechanism. This overview is intended to familiarize the reader with the basic mechanism of how the specification of a service by a user from a shell interface is translated into the invocation of that service on a machine providing an instance of the service.

At a conceptual level, the set of services available in the computing engine is grouped into a single registry of names. In practice, the set of services are managed by a set of service servers, which are name servers, with one service server resident on each machine. Each service server knows about the services available on its machine, and controls what services are offered to users on the local and remote machines in the computing engine. Services are made available on a machine by a user who registers an application as a service. Registration involves communication with the local service server to supply the service name, its invocation protocol, and its underlying name.

Each service server also knows about characteristics of the machine on which it resides, such as processing power and load. In addition, each service server knows, or can find out, about service information from other machines in the computing engine. Thus, for a user’s service execution request, the local service server is used to resolve a service name and select a machine for invocation.

Once a machine has been selected to provide a service, a cycle server on that machine is contacted to actually carry out the service. The cycle server knows how to invoke the service given the service name by mapping the service to its underlying name. The cycle server also inherits the user’s execution environment from the client machine and checks to make sure the user is authorized to use the service before invoking it. Given this brief overview, an example of the service execution mechanism is shown in Figure 4.1.

The picture captures the essential steps needed to execute a service with our service execution mechanism. As an example, the service \textit{tex-std} (a standard version of the TeX text processor) is executed. The steps are as follows:

1. The service shell, running on a user’s workstation, receives a service specification from a user and creates a client process to handle the execution of this service.

2. The client process sends a message containing the name of the service to the service server on its local machine.

3. The service server resolves the service name to an absolute service name, locates the machines that offer the service, selects the best machine to invoke the service, and chooses the correct protocol to use for contacting the remote machine. It then
Figure 4.1 Execution of a Service

sends the absolute service name (*tex-std*), machine name (A), and invocation protocol (stream) back to the client process.

4. The client process makes a connection to the cycle server on the given machine using the invocation protocol given to it by the service server. It passes the absolute name of the service and any arguments to the cycle server.

5. The cycle server maps the service name to a binary file and creates a process to execute the contents of the file with any output sent back to the client process on the user's workstation using the network connection.

6. The remote TeX service completes, causing the client process to terminate, and the service shell is ready for another service request.

This example shows how one service might be executed with our mechanism and ignores many details about how each component is performed. These details are forthcoming. What is important to note from this example is

- a service server on each machine knows about services on that machine and can find out about services on other machines,

- the service server is responsible for resolving a service name, and determining on what machine and with what protocol the service should be invoked, and

- a cycle server on each server machine knows how to execute services advertised by that machine.
4.4 Service Initiation

As illustrated in the overview of the service execution mechanism, the principle interface for users to initiate services is a service shell. In this section we examine how the service abstraction may be used by other applications to initiate actions in a distributed environment.

The basic observation is that the initiation of services in a distributed environment can be substituted for the initiation of commands in a single machine environment. For example, a tool that initiates commands at pre-specified times can be extended to initiate services. The event-based automation tool Omicron can be extended to trigger services based on events that are observed to have occurred in the distributed environment.

Other tools that initiate services are decision services, which make decisions about where and what services to initiate based on service specific criteria. For example, the UNIX tool make [UNI86] uses file dependencies to selectively cause computations such as compilations to be initiated. This tool can be extended to a distributed environment with the service abstraction. Another decision service is a user location service, which consults a database of information to determine on which machine a user is logged in and then initiates execution of a query service on that machine.

Another example of a service initiating other services is a shell script. A shell script, as used in UNIX, groups shell and UNIX commands together into a single executable file that is available as a command. Extending the notion of shell scripts to services implies that each command becomes a service and that each service within the script is conceivably provided by any server machine in the computing engine. The practicality of treating all actions within the script as services depends on the cost of executing services versus the potential performance benefit of invoking each service on the best machine.

A similar tradeoff exists for a multi-phase compiler where each phase is initiated as a separate process. Should the compiler be viewed as a “small” service that initiates many sub-services or a “large” service that initiates many processes on the same machine? For the user of the compiler service the decision should not be important; for the provider of the service the most efficient approach should be used.

In summary, we expect services to be initiated principally by users interacting with a shell, but as we have shown the service abstraction can also be used to extend event or decision based tools to a distributed environment. With other services that initiate services, such as shell scripts, the question is how far to carry out the abstraction. The answer can only be determined by knowing the relative costs of the service abstraction and the computation requirements of the services to be executed.

4.5 Service Naming

Our service execution mechanism uses two names for each service—the name presented to the user for specification of the service, and the name used by the underlying protocol to invoke the service. The appearance of the name space presented to the user is a fundamental issue in the design. The form of a service name

- influences the user’s perception of the system,
• defines whether partial service names can be used and how the names are resolved to absolute service names, and

• determines the ease that the service name space can be managed.

The use of a separate high-level name space for services leads to the question of how services should be named in a distributed environment. Because the high-level service name is not dependent on where or how the service is invoked, we have freedom to adopt the “best” naming style.

With this freedom also comes difficulties. When a file name is used both as a command name and a means to invoke the command, two desirable characteristics for the command name space are obtained from the file name space. First, the command name space is already partitioned between users. Second, a command always performs the same function because the same executable file is always invoked. In naming services, these characteristics do not already exist. Rather than naming a concrete object, such as a file, a service name denotes a function. How to partition the service name space among the users of the computing engine, and guarantee that services with the same name perform the same function are difficult problems to be solved. In examining alternate styles for naming services we assert the following characteristics are desirable for service names.

• Services that have the same name must provide the same function. Users expect that if different machines offer services by the same name that these services are in fact instances of the same service. Any other definition would force the user of a service name to be aware of differences in function depending on which machine provides the service.

• Services that provide the same function should have the same name. Otherwise the user is forced to select between equivalent services when the execution mechanism could locate the best instance of a service.

• Services that provide similar functions should have similar names. The user should easily be able to recognize when groups of services are related or two different versions of the same service exist.

4.5.1 Naming Styles

Given these set of characteristics, we investigate different styles for naming services. A simple, but undesirable name space for services is a flat name space. In a flat name space an object is named by a single system-wide unique identifier that may be a number or a character name. A flat name space requires a central authority to arbitrate the assignment of names to objects. For object names that never change and must be unique, such as user identification names (user ids), a flat name space under central control is desirable. However for services in a distributed environment, the need for a central authority to assign a name each time a user wants to offer a new service can lead to administrative bottleneck. In addition, the lack of structure in the name space frustrates the goal of allowing partial names to be used.
A popular style for naming objects in a distributed system is a hierarchy. A hierarchical name space is structured as a set of levels, where each level is an attribute, and a name is a set of attributes separated by delimiters. An attribute is a descriptive name for the object. In hierarchies of unlimited depth, there is no fixed relationship between a level and an attribute, but in fixed depth hierarchies, such as used by Clearinghouse, each level corresponds to a particular type of attribute. Hierarchies are popular because the names can be easily partitioned between name servers in the distributed system by explicitly including the server name as the highest level attribute or assigning particular portions of the hierarchy to a server.

A third approach to naming objects in a distributed environment is attribute-based (also called property or descriptive names). Unlike a hierarchy, the attributes are not ordered, and they may be explicitly named with a keyword, and the value of the attribute may be of a particular type. Attribute-based names are often used as an alternate style for naming objects. For example, Peterson [Pet87a] uses untyped, attribute-based names to identify principals (authorities, such as users, that control some portion of the resource space), which are named in their own unique name space. In Clearinghouse, additional named and typed attributes (called properties) are used to augment the three-level hierarchical name space for objects.

Attributes work well to select a group of objects that share a set of common characteristics, such as all users with their mail home on a particular machine, or all PostScript printers. One problem with attributes is getting the authority and user of an object to agree on the set of attributes that describe the object. A second problem identified by Peterson is ensuring that attributes are complete. An attribute is complete if and only if it is associated with all objects it describes. To solve these problems, Peterson uses a set of attributes automatically generated from password and mail alias files as primary attributes for principals. He does allow manually entered attributes for principals, but treats these as secondary attributes.

4.5.2 The Service Name Space

Attribute-based names appear promising for objects in which a set of well-defined, complete attributes can be used to describe the objects. However for services, the number of attributes needed to describe the function of services is not well-defined, which leads to problems for both the authority and the user of the service. The authority must ensure the attributes uniquely describe a service, but are also complete. The user must remember which attributes to use to specify a particular service. To avoid these difficulties we propose a fixed, ordered set of attributes to be associated with each service, similar to the manner used by Clearinghouse for naming objects. For the service name space, we propose a set of three attributes separated by dashes of the form action-authority-class.

The action attribute\(^1\) is the primary name for the service and is always the first attribute given. It is a descriptive name for the function of the service. For compatibility with existing systems it may often be the distinguishing portion of the underlying name of the service. For example, the action attribute for the TeX text processing service might

---

\(^1\)An action is a named computation, an action attribute is the first attribute in a service name.
be `tex`, and the action attribute for a service to sort a file might be `sort`. While the action attribute is a convenient partial name for users to use in specifying a service, the remaining two attributes are used to guarantee the service name is unique.

The `authority` attribute explicitly indicates the person who provides a service and vouches for its function. By convention, the authority attribute is either a user's account name (user id) signifying that he is the authority for the service, or is `sys` signifying that the system administrator is the service authority. A basic assumption of our naming mechanism is that the set of user ids are unique within the computing engine. From experience with our computing environment this is a reasonable assumption. The explicit use of the authority attribute provides a number of advantages.

- The person who provides each service is known.
- Because the set of user ids is unique within a computing engine, each user has his own partition of the service name space.
- Two services can have the same name only if explicitly given so by the same person.

The `class` attribute of the service name is used to distinguish a set of services that share a particular property. A class may be used to collect a subsystem of related services such as a revision control system, or a message handling system; to distinguish a version of services, such as `std` (standard version), `new` (new version), or `local` (version local to a site); or, to group a set of services associated with a project.

We expect different classes to be used predominantly by the system administrator to partition system services; the class `std` will be sufficient to contain the services provided by a specific user. As a consequence, we allow service names to be shortened to two attributes, an action and authority or an action and a class, whenever possible. If only two attributes are given then `sys` is tried as the authority, and then `std` is tried as the class. Thus, a local version of the troff text processing service could be named `troff-local` rather than `troff-sys-local`, and a time service offered by the user `zew` could be named `time-zew` rather than `time-zew-std`. To avoid ambiguities in service names the system administrator must ensure that service class names for the authority `sys` are unique from the set of user ids.

A small point about service names is that any flags and options are not considered part of the service name. One view of naming services is that each flag actually causes a different service to be invoked. For example the `move-std -t "service"` would be different than the `move-std -n "service."` We reject this view because it unnecessarily complicates the meaning of a service for our current research. For now we view that `move-std` is a service with the `-t` and `-n` flags serving as modifiers of its function. In the future we would need to consider more closely how the flags of a service affect its function.

Relating back to the naming characteristics we identified at the beginning of this section, our naming scheme exhibits the features outlined. Because each user has control over his portion of the name space, he can ensure that services offered by multiple machines with the same service name provide the same function. Because multiple instances of services provide increased reliability and performance, users will give the same name to services that provide the same function. Finally, through the use of the class attribute,
services that share particular properties can be given similar names. For example, different versions of a service can be given the same action attribute, but different class attributes.

4.5.3 Service Equivalence

A basic assumption of our naming mechanism is that a service can have multiple instances that each provide the same function. In light of this assumption, we need to examine what it means for two services to provide the same function. Possible definitions are:

1. The binary object code used to invoke the two services is the same.
2. The source code used to create the binary object code for the two services is the same.
3. The two services are implementations of the same design standard.
4. The two services produce exactly the same output for the same input.
5. The two services produce "roughly" the same output for the same input.

Because we are not restricting our computing engine to a particular type of processor or operating system, we view the first two definitions as too narrow to be adopted as a general definition. The first definition is applicable to a homogeneous system (same operating system, same processor) where all services are invoked in the same execution environment. Only in this case is it possible for two services to be verified as the same by checking their "contents"—the binary code. The second definition is used by Locus and the V-SYSTEM where the same operating system is used, but multiple processor types are supported.

For our definition, we view that two services are the same if they are implemented from the same standard, or they can be verified, through a set of test cases, to provide the same function. For the user, this definition means that all instances of a service can be described by the same document.

Standards are often defined for large, frequently used applications such as compilers, interpreters, and text processors. For example, many types of machines may implement the TeX text processor service. In other cases, a standard is not explicitly defined, but an implicit standard exists for what the service should do. For example, a spell service should report any misspelled words in its input. Thus, we view two services the same if they are implementations of the same design standard, whether explicit or implicit. Translated to our name space, this definition means the authority of a service verifies that all instances of the service meet the same standard. Although our definition is less restrictive than the first two definitions of service equivalence, we do expect services to be implemented from the same source code within homogeneous portions of our computing engine for convenience and reliability.

The last definition in our list leads to a weaker, but still valid, notion of consistency for services. Cases occur where a user wants to select one of many services that perform
roughly the same function rather than exactly the same function. To capture this notion of services that are related rather than services that are the same we introduce a new type of service called a *metaservice*.

### 4.5.4 Metaservices

The service name space defines that two services with the same name provide the same function. In addition, we would like to easily identify services that share some other functional relationship. For example, a user may want to initiate a service to print the date, but he does not care the format in which it is printed. In this case, any service that prints the date provides the “same” function, and the service mechanism can select the specific service to use based on performance criteria. The problem is how to give the same name to services that have related function rather than just services that have the same function.

One approach for identifying related services is to augment the service name space with a set of descriptive attributes to describe the function of a service. Specifying a set of related services would then be a matter of choosing the common attributes of these services. However, as noted in the discussion on service names, the use of arbitrary attributes to describe services may not be complete and lead to unwanted ambiguities between the attributes given by the authority of a service and the user of a service.

Instead, we extend our naming mechanism to allow a service name to contain links to one or more other service names. A metaservice is a special kind of service that, instead of containing a protocol and underlying name to invoke a service directly, contains “pointers” to other services. Execution of a metaservice causes one of the other services to be invoked. The authority for a metaservice explicitly chooses the services that provide a particular functionality. From the user’s point of view, using a metaservice is no different than using a service. For example, the system administrator may define a general date metaservice to be provided by many different services, while a specific user may define a metaservice for his personal use. The term “metaservice” is inspired by the term “metafile” in the Saguaro operating system, which is a special file that contains symbolic pathnames of other files [ASHP87].

Metaservices can be used to define any type of functional relationship between services. The relationship is implicitly defined by the authority for the metaservice. A simple type of relationship is to alias a metaservice name for a service name. Aliases are useful on a per-user basis to define alternate names for services. For example in Figure 4.2a, the user ceu has created the metaservice `rn-ceu` as an alias for the local version of the `readnews` service. As opposed to a shell alias, a metaservice alias is available to all applications because the metaservice is treated as a normal service.

In addition to aliases, we envision two other types of functional relationships to be defined with metaservices. One type of relationship is for services that share common functionality, but they are not the same service. In this case, a metaservice groups together a set of services into an *equivalence class* based on a relation. The date metaservice defined above is a good example. Another example is the case of two search services that each
search a file of text for a particular string. Each service can handle simple strings, but each has separate facilities and differing performance characteristics for handling specialized strings. A search metaservice could be created that equivalences these two services. The metaservice would then be useful when a search service for a simple string was needed. The example in Figure 4.2b, shows the search-local metaservice defined as an alias for either the grep-std or bm-local search services.

The other type of functional relationship is useful when the functionality of one service encompasses another. This relation also introduces the notion that a service can be both a normal service (have an underlying implementation) and a metaservice. A good example of this kind of relation, as shown in Figure 4.2c, is two compiler services that each produce code for a particular architecture. One of these services has an option for producing optimized code, the other does not, hence they are different services. However, as far as users of the non-optimized compiler are concerned either service is satisfactory.

In summary, we have chosen to provide services exhibiting a functional relationship, other than "the same," with the concept of a metaservice. Another way to view a metaservice is simply as a service whose function is not carried out by invoking an underlying implementation, but by invoking another service. This approach allows all services (including metaservices) to be initiated in the same manner, but requires users and system administrators to explicitly define metaservices as needed.

4.5.5 Extending the Service Name Space

Although we expect almost all services to be provided by machines in the local computing engine, facilities must exist for users to access services that are provided by machines in other computing engines. This requirement leads us to examine how the service name space can be extended across multiple computing engines.
Two possibilities exist for naming services across computing engines. The first is to ignore computing engine boundaries and use the same name space for all services. The advantage is that services providing the same function can be given the same name across computing engine boundaries. The disadvantage is that each computing engine is under separate administrative control and services with the same name are not guaranteed to provide the same function unless administrators coordinate service names. Furthermore, user ids are not unique across computing engines and the same name for a user service would not always refer to the same function.

The second possibility, and the one we choose, is to name each computing engine and include this name as an implicit attribute of a service name. This attribute is a context that normally defaults to the user's local computing engine. The advantage of this approach is that service names do not conflict between computing engines. The disadvantage is that the same service in different computing engines cannot be given the same name due to differing computing engine names. This problem can be partially solved by administrators of computing engines using metaservices to group equivalent services from remote computing engines into a local service name. Metaservices can also be used by users to equivalence services from other computing engines, or to create local aliases for services that are available in other computing engines.

We postulate using the Domain Name System (DNS) [Moc87] to name computing engines. This hierarchical name space is used to name hosts in the Internet by dividing the Internet into a set of zones, each consisting of a number of hosts under a system administrator's control. Because we also define the scope of a computing engine to be under an administrator's control, domain names are an obvious approach for naming computing engines. For example the Computer Science department computing engine at Purdue would be named cs.purdue.edu. Local facilities can be used to manage and retrieve service information within a computing engine and the DNS can be used to obtain service information from remote computing engines with higher expected retrieval costs.

In summary, we are principally concerned with naming issues within a particular computing engine, but recognize the need to be able to extend the name space. The use of domain names is a practical approach for extending the name space that provides access and protocol, if not location, transparency for initiating services in remote computing engines. Experience with an implementation spanning multiple computing engines is needed to test how well the administrative boundaries of computing engines would match the zone boundaries in the DNS.

4.6 Service Resolution

In general, users are averse to using long, absolute names for objects such as services, and would much rather use short names instead. The resolution component of our service mechanism addresses the problem of transforming partially specified names into absolute service names. The general form of our service name space, concentrating on names within a computing engine, is action-authority-class. Because the action is the primary attribute of the name and the remaining two attributes identify the provider and a version of the
service, we expect services to be specified using the action attribute as a partial specification for the absolute service name. To resolve such partial specifications we use a \textit{service set}.

A service set is an ordered set of attributes that is used to resolve a service name when a partial service name (one with only an action attribute) is given. Each element of the set is an \textit{authority-class} pair, but because a unique service name can be specified with only two attributes, each element of the service set often consists of a single authority or class attribute. Resolution occurs by successively appending each element of the service set to the partial service name until a valid service name is generated. If resolution succeeds then the absolute service name is passed on to the location component. If the end of the service set is reached without generating a valid service name then failure is reported and the service execution mechanism terminates. The service set is specific to each process, but is generally the same for all processes of a particular user. The concept is similar to a search path in UNIX, except each element of the service set is an attribute (or two) rather than a directory. To illustrate service resolution, Figure 4.3 shows the results of resolving four partial service names.

\begin{verbatim}
Available Services
... bm-local grep-std locate-cew
locate-cew-new search-local tex-local tex-std
SERVSET=cew-new:cew:std:local
RESOLVE(tex) → tex-std
RESOLVE(tec) → failure
RESOLVE(search) → bm-local, grep-std
RESOLVE(locate-cew) → locate-cew
\end{verbatim}

Figure 4.3 Examples of Service Resolution

In each case the service set used has the value \texttt{cew-new:cew:std:local} signifying that new services from the user \texttt{cew}, followed by standard services from the user \texttt{cew}, then standard system services, and finally local system services should be used to resolve partial service names. In the first example, the action attribute \texttt{tex} could be resolved to either \texttt{tex-std} or \texttt{tex-local}, but is resolved to the former because \texttt{std} precedes \texttt{local} in the service set. In the second example, resolution fails because the given partial name cannot be matched with any element in the service set to form a valid service. In the third example, the metaservice \texttt{search-local} from Figure 4.2b is the result of resolving \texttt{search} with the given service set. By definition, the resolution component recurses until each metaservice has been expanded to the set of services that are pointed to by the metaservice. In this example, the metaservice expands to \texttt{bm-local} and \texttt{grep-std} denoting that either service can be used. In the fourth
example, an available absolute service name is given so the service set is not used and the name is the result of the resolution component.

In comparison with resolution of commands in a single-machine operating system, resolution of services in a distributed environment raises a number of additional questions.

4.6.1 Treatment of Executable Files

In UNIX, because commands are named and implemented as executable files, any file name is a potential command name. In practice, all commands are grouped in a few directories of executable files, and these directories are commonly specified in the user's search path. However, to allow access to commands in the user's current context, a "." in the user's search path signifies to search the user's current working directory for an executable file when given a partial command name.

The inclusion of a similar notation in our mechanism violates our principle that service names are separate from their implementation. Instead, the use of executable files in our mechanism is properly treated by the use of a "binary" computation service that invokes a named executable file. Not only do server machines offer services to execute specific computations, but they also offer a run binary service that will execute an executable file on their machine. This service allows executable files to be compiled and tested within the framework of the service model. Once an executable file performs the task it was designed for, it may be registered as a service with the service server on the local machine. Registration is the act of instantiating an instance (in this case the executable file) of a service on a particular machine.

4.6.2 Name Resolution Conflicts

In UNIX the most common problem with the use of a search path is name conflicts in resolving partial command names. In some cases, such as having new and old versions of a command, name conflicts are expected and resolution correctly chooses the version desired by the user without intervention on his part. However in other cases unwanted name conflicts occur. An unwanted name conflict occurs when a partial name resolves to an absolute name that is not expected by the user. The user may misspell a partial name, but unexpectedly see it resolved to an unfamiliar command, or an executable file might be added to a directory in the user's search path that causes an unwanted name conflict.

Resolution of services using a service set has the same potential for producing unwanted name conflicts. Some problems are unavoidable, but we propose to lessen the effects by using the class attribute of service names to group each identifiable subsystem of services together with the same class name. This idea is an improvement on UNIX where many subsystems of commands are often grouped into a single directory. By grouping subsystems of services into service classes, users can specify in their service set only the classes they use and reduce the chance of unwanted name conflicts during resolution. In effect, we are trying to make the number of services with the attributes given in the service set be as close as possible to the number of services actually used by a user.
4.6.3 Using Service Information for Resolution

Another resolution problem, unique to distributed systems, is how service resolution should react to the effects of services no longer available because of machine failure. Suppose a user has the service set `std:local` and the only machine that provides a standard version of the `tex` service fails, but many instances of `tex-local` exist. How should resolution proceed? The answer depends on how the service information is managed. If the information is dynamically obtained then only the local version of `tex` will be known and thus be resolved. However if service information is cached, the resolution component could detect that a standard versions exists, despite the providing machine being down. In this case the resolution could fail because the "correct" version of the service is not available.

We take the view that resolution should use the set of services that is currently available. This problem does raise a more fundamental question of how service information is managed and located. We discuss this problem briefly in the next section and in detail in the following chapter.

4.7 Service Location and Selection

Each service may have multiple instances, each offered by a server machine in the computing engine. The task of locating the set of server machines that offer at least one instance of a service is known as service location. The task of selecting a machine from this set that provides the instance at the least cost is known as service selection.

Determination of what service instances are available, needed for service location and resolution (successful if at least one instance of a service exists), is a problem of managing and locating service information. We determine the best approach to managing service information in the next chapter as we present and evaluate different approaches for managing information in a distributed environment.

Service selection involves gathering information about the resolved service and the set of eligible machines that provide the service, and then using a selection policy to make a decision. In the remainder of this section we focus on the details of service selection and develop simple heuristics for selecting a machine to provide a service. As part of the heuristics, the current status of the eligible machines is needed. This problem is another example of a distributed information management problem and is also examined in the next chapter.

4.7.1 The Service Selection Problem

The basic problem of service selection is a matter of optimization. Given information about a service, and the server machines that may provide the service, a decision must be made concerning what is the "best" machine to select for performing the service. To make this decision, a cost metric, specifying what is to be optimized, must be chosen, and a cost function, specifying how to compute the cost metric, must be derived.

The cost metric that we choose to use for the general problem of service selection is response time, where response time is measured in terms of the elapsed time from when a service is initiated for execution until the service has completed execution. Other cost
metrics may be appropriate for specialized services. For example, a talk service, which allows users to exchange real-time messages between terminals, would use a cost metric that finds the message receiver's login session with the least idle time. A printer service would select the printer that has the smallest queue. For most services, response time is the appropriate metric. For a user interacting with a service shell, it is the time between when a service is initiated and when the service has completed.

A number of factors contribute to delay between when a service is initiated and when it is done executing. These factors are:

- **Decision.** The time needed to gather information and make a decision about how to resolve, locate, and select a machine for invocation.

- **Invocation.** The time needed to invoke (create or contact) a process to perform the service on the selected machine. For example, a service invoked as a process on the client machine might be much cheaper to invoke than one on a server machine.

- **Execution.** The CPU time needed by the process during execution, which is directly dependent on the processing power of the selected machine, and the processing demands of the service itself.

- **Ready.** The time spent by the process, ready for execution, but waiting to be served by the CPU. This factor is a function of the load of the processor and the CPU time needed by the process.

- **Data.** The time spent by the process waiting for acquisition of resources, specifically accessing data. This cost depends on the location of the computation in relation to the location of the data.

To encapsulate this information, we derive the selection cost function $C$ where the time function $T$ represents the time spent for each phase of execution.

$$C = T_{\text{Decision}} + T_{\text{Invocation}} + T_{\text{Execution}} + T_{\text{Ready}} + T_{\text{Data}}$$

The optimization of the service selection function depends on a number of factors based on the processing demands of the service itself and on the capabilities of the machine providing the service. An interesting constraint of the service selection optimization problem is that the time needed to gather the information and perform the optimization is part of the optimization itself. Hence, the time spent to select the best machine over an arbitrary machine must result in at least that much better performance. Because execution time and data acquisition costs may be difficult to obtain at selection time, and because the optimization is constrained, selection of the optimal machine in most cases is impossible. Consequently, we choose to use heuristics that attempt to make good, not necessarily optimal, selection decisions.
4.7.2 Our Approach to Service Selection

Our approach is to invoke services on the machine with the lowest load to processing power ratio after adding in a cost for invocation on remote machines. This idea is similar to the threshold idea proposed in [ELZ86] with the cost of remote invocation serving as a threshold for when remote invocation is feasible. A problem with this simple heuristic is that it ignores the processing and data acquisition demands of the service itself. To make better decisions we consider additional heuristics that take into account the demands of particular services in a cost efficient manner.

One approach, presented in a paper related to our work [Com85], is that each user be able to supply parameters to influence how service selection is done for each service. In general we believe users should only specify what they want done (the service) and depend upon the system to find the instance of that service providing the best performance. Therefore users should not need to influence where a service is performed, but users should be able to supply hints for service selection as they desire. For example, a user may hint that a service will be computation-intensive causing service selection to tend towards selecting a powerful server machine.

To reduce the need for users to know the details about a particular service, we place responsibility on the authority of a service to provide hints about the processing demands of the service at registration time. Authorities indicate whether a service should only be available on the client machine, only from a server machine, or from either machine depending on the current system status. As an aid to classification, we identify the following types of services:

- services that require interaction with the user such as editors,
- services that are computation-intensive such as image processors,
- services whose computation requirements are data-dependent such as sort and search services, and
- services that make decisions about where to invoke additional services such as the make service.

Interactive, decision, and non-computation-intensive services are appropriate to be used from the client machine whenever possible. Computation-intensive services should only be offered by and used from server machines. Where data-dependent services should be invoked is difficult because determination of the size and location of data is difficult. Depending on the cost of invoking a service on a remote machine, one strategy is to invoke such services locally under the assumption that most data search and transformation services deal with relatively small amounts of data. In cases where additional information about the size or location of data is known (such as a hint from the user) then remote invocation would be performed as necessary.

In summary, we have chosen response time as the cost metric to optimize in service selection, and identified a cost function that incorporates the potential costs. Problems in measuring these costs, as well as inclusion of the optimization cost as part of the function,
led us to consider low-cost heuristics that attempt to make good, rather than optimal, selection decisions. These heuristics depend on the service authority, and perhaps the user, to supply parameters concerning the size of the service and any special processing requirements. In turn, simple heuristics based on the cost of the invocation protocol, machine processing power, and machine load are used to select a machine for invocation. How to efficiently obtain the load information from the server machines is studied in the next chapter.

4.8 Choosing an Invocation Protocol

In previous sections, we have investigated how a partial service name is resolved to an absolute service name, and how a server machine is selected for invocation. In this section we discuss the issues involved in invoking the service on the selected machine.

A service invocation protocol is a set of conventions agreed upon between a client process wishing to invoke a service and a cycle server process agreeing to provide the service. In general, we assume that these processes reside on separate machines. In cases where the client machine is selected for invocation, a more efficient means of invocation may be used as will be detailed in the discussion of our prototype. Among the conventions defined by an invocation protocol are how a service name is mapped into the appropriate underlying name, how a service invocation request is authenticated, and what portion of the execution environment of the client process is inherited for invocation.

The choice of an invocation protocol in many systems is not an issue. System designers choose, at system design time, a particular protocol for invocation and all actions invoked in the system use the same protocol. Although the invocation of actions is transparent to the user, all actions must be invoked in the same manner. To use an action requiring a different protocol, users must explicitly invoke an action that implements this protocol.

In a distributed environment, we view the use of a single protocol for invoking all services as the wrong approach. Instead, the appropriate protocol to invoke a service should be chosen at service execution time. Late-binding for the choice of a protocol is good because not only is the choice of a protocol transparent to the user, but the choice can be dependent on requirements of the service itself. For example, a time-of-day service may only require a simple datagram protocol with no authentication for use and no inheritance of the client's execution environment. A text processing service may require a stream (reliable stream of bytes) protocol that includes inheritance of the client's execution environment. Thus, two services initiated in the same manner by the user may be invoked in different ways.

The use of late binding ideally leads to the model of service invocation shown in Figure 4.4, a more complete version of Figure 4.1. The cycle server on the server machine understands how to invoke any service advertised by its service server that is presented to it from a client process on a client machine. For a client process to invoke any service on the server machine, it contacts the cycle server on that machine with the name of the service. Because the cycle server shares access to the service information with the service server, it can map the service name to the appropriate underlying name needed for invocation. As part of the exchange between the two processes, the client's identification and execution
environment can be passed as needed. After setup, the connection is used for passing input to the invoked service and sending output back to the client process.

This model of service invocation is desirable because it means that the client machines need only know about the cycle server on the selected server machine to invoke a service. Conceptually, the client and cycle server processes can negotiate the necessary protocol to use for actual invocation of each service. However, this model of service invocation does not appear suitable in our present environment for two reasons. First, the connection between the client and server process must always be made with the same transport protocol. In the prior example, the time-of-day service was best served by a datagram protocol, while the text processing service by a reliable stream protocol. Second, the model does not incorporate the use of services that can be contacted directly by client processes. Thus, a more realistic model of service invocation is shown in Figure 4.5.

The figure shows two (of possibly many) different cycle servers, each sharing the service information but implementing a separate invocation protocol on the server machine. The protocol implemented by each cycle server is fixed in terms of the authentication it performs and the execution environment it inherits. To use a service, a client process must not only know the service name, but also the invocation protocol needed by the service so it can contact the correct cycle server. The invocation protocol needed for a service is registered by the service authority and is advertised to client machines along with the service name.
This approach still allows the protocol to be bound late, but allows the client and server machines to support invocation protocols on an "as need" basis.

In addition, the special service in Figure 4.5 illustrates a server process that does not have access to the local service information. This situation may occur when a particular service is offered at a specific port, such as an Internet service. In these cases the client machine needs to not only know the service name and the invocation protocol, but also the underlying name, the port number. Although we want to avoid situations where the client machine needs to know the underlying name such exceptions appear necessary.

4.8.1 Execution Environment

The key observation about the execution environment is that enough of it must be inherited from the client process to result in network transparent execution of the invoked service. For services such as a time-of-day service, any server machine can provide the service because no portion of the invoker's execution environment is needed. However for services that expect to access files named by the user, the global name space of the server machine must be the same as the client machine.

The execution environment classifies the set of services offered by server machines available to a client machine into three groups:

- services that inherit parameter bindings from the client process,
- services that inherit local bindings from the client process, and
- services that share the same global bindings as the client process.
Each service is advertised with what level of execution environment sharing it expects. For example, a service may expect to access files named by the user so it expects to share the same global bindings as the client machine. The client machine can check if this service is available to its users by comparing if it lies in the same global name space as the server machine. The classification of services based on the execution environment leads to the realization that services are not universally available to all users, as is further discussed in the following section.

4.8.2 Authentication

As part of the invocation protocol, each cycle server has the right to accept or refuse any service request. The cycle server may only provide services to users on a particular set of client machines, or it may only provide services to a particular set of users that have permission to invoke services on the machine. Thus part of the invocation protocol involves authenticating that a user, identified by his user id, has permission to invoke a service. In addition, the service itself may require additional authentication information, such as a password, from the user.

Although authentication is ultimately done as part of the invocation protocol, it may be useful to check prior to invocation if a service can be invoked by a user on a particular machine. These “availability” checks may be used by the resolution and location components to avoid the situation that a particular instance of a service is selected for invocation only to discover that the user is not permitted to use the service instance. For example, associated with each service instance is a flag indicating whether a user of the particular instance must have permission to invoke services on the server machine. Using a local database of where a user has permission to invoke services, the client machine can avoid selecting a service instance that the user is not permitted to use.

Our treatment of authentication is analogous to the end-to-end argument made by Saltzer, Reed, and Clark [SRC84] concerning reliability in a distributed system. They argue that reliability for a transaction, such as file transfer, is the ultimate responsibility of the end points of the transaction. Consequently, any intermediate reliability guarantees cannot be used to guarantee the end-to-end reliability of the transaction, but can only be justified on efficiency grounds. In a similar manner, we argue that authentication of service invocation must ultimately be done by the servers that control the services, but we find the use of intermediate availability checks valuable for increased efficiency.

4.8.3 Summary

The late-binding of a service to a protocol to invoke an instance of the service allows the appropriate protocol to be chosen for each instance. Our viewpoint is that one protocol is not satisfactory for all services, but neither is a specialized protocol for each service. Instead, we propose the adoption of a few protocols that can be used by most services and allow for specialized protocols as needed. This decision places the onus on the authority of a service to register the service with the invocation protocol that provides the needed level of environment inheritance and authentication for the service to work correctly. A major gain with common protocols is that new services can be offered by server machines
and be immediately available to users on client machines because the client routine for the protocol is already available.

4.9 Summary

In this chapter we have described a service execution mechanism designed to provide users transparent access to computational services in a distributed environment. To summarize our design we apply the action execution model to the execution of the service \texttt{tex}, which invokes the TeX text processor, shown in Figure 4.6 and discuss its implications.

```
static binding 
  early binding 
    INITIATE \( \nu = \text{tex} \)
    RESOLVE \( \omega = \text{tex-std} \)
    LOCATE \( \{s_1, s_2, \ldots, s_n\} \)
    SELECT \( s = s_{\text{good}} \)
    PROTOCOL \( \pi = \text{stream} \)
    MAP \( \nu_n = /bin/tex \)
    INVOKE \( \pi, \nu_n, s \)
```

Figure 4.6 Action Execution Model for the Service Execution Mechanism

Resolution of the partial service name \texttt{tex} is performed using a service set that results in a standard system version of the service being resolved. The set of server machines that offer the service is dependent on the service, and we use heuristics that select a good, not necessarily optimal, machine to invoke the service. A client process on the user's client machine contacts a cycle server on the selected machine using a stream protocol and passes authentication and execution environment information as needed. The cycle server maps the service name to an underlying file name that is used to start execution of the process to perform the service.

The service mechanism exhibits a number of transparency characteristics.

- The mechanism provides network transparency by limiting the use of a service to the machines that share the necessary execution environment of the service. Thus, many services can be shared between machines in homogeneous portions of the computing engine while fewer services are offered between machines in different global execution environments.
• The mechanism is location transparent because the user does not need to be aware of which machines offer a service. Services from newly added server machines are transparently added to the set of services available to the user.

• The mechanism is both access and protocol transparent because all services are initiated in the same manner, and the protocol used for invocation is transparent to the user. New services that use a common protocol for invocation are available to users at client machines without any additional work at the client machine.
5. LOCATING DISTRIBUTED INFORMATION

5.1 Introduction

In the previous chapter we described a design for transparent access to services in a distributed environment. Among the problems encountered in the design were how to manage and locate information about services and the status of server machines in the computing engine. Location of the available service instances from the server machines is a central problem in service resolution and location. Retrieval of status information about server machines, specifically their load, is needed to select a machine for service invocation.

The key observation about these two problems is that they are representative of a class of distributed information problems. This class of problems deals with status or availability information that is supplied on a per-machine basis. This class of problems arises not only for locating service and machine status information, but also for such problems as locating the current status of a user interacting with machines in a distributed environment or locating the machine that contains a particular resource. The collective information available from all machines defines a distributed database. Traditional distributed database designs maintain replicated, consistent copies of a database in the face of machine failures and multiple sources of updates. Our class of problems is distinguished from the traditional problem by the following characteristics.

• The information is inherently distributed with each server machine supplying a portion of the total information. Although information may be replicated on other machines, each piece of information has only one source of updates.

• Slightly out-of-date copies of the information can be used but should eventually converge to the most current information. Weaker consistency requirements for copies of the information allow updates to be made without using commit protocols to ensure consistency of all copies.

• Performance in locating information is important. Depending on the desired consistency constraints, accuracy of the obtained information may be traded off for quicker access to the information.

• Information is volatile and may go away. Because all information is particular to a machine, failure of a machine makes its information unavailable and copies of the information to be invalid.

The last characteristic leads to the need to detect and possibly measure negative information. We define negative information to be information about the unavailability or negative status of a resource. Examples of detecting or measuring negative information
are determining if a service or file is not available from a server machine, how long a user has been logged out, or if logged in then how long he has been idle. Techniques used for location of information may not be appropriate, or have to be modified, for detection or measurement of negative information. For example, querying a machine directly can be used to obtain its current status, but if the machine fails the same technique cannot be used to find out how long it has been down.

Thus, the problems we wish to investigate are actually just instances of a class of distributed information location problems. Rather than investigate the problems directly, we examine different techniques for storing, updating, and accessing information for the class of problems. Our examination of the appropriate technique for a problem is similar to Mullender and Vitányi’s [MV85] study of distributed match-making between a process asking for a service and a process giving that service. However, our problem space is more general, and our approach is an analytical rather than a theoretical comparison of the techniques. In other related work, Ammar et al. [AAA88] address the specific issue of how system parameters affect the performance of location operations when hint tables (caching) are used.

Our objective is to understand the general issue of how to choose a technique based on the parameters of a problem. The parameters include how often the information is accessed, how often the information is updated, if the location of the resource can be determined from its name or indirectly through a database, and the communication costs to access information on other machines. Communication costs may vary widely depending on whether the communication medium is a local area or long haul network.

Choosing a technique for a problem implies evaluating each technique to determine which minimizes potential costs. A number of criteria can be used to measure costs such as the response time to access information, CPU time spent by each machine to manage information, number of network packets, and accuracy of the information. In this chapter we first examine general techniques for managing information and how various evaluation criteria, when applied to the techniques, are affected by the parameters and scale of a specific problem. We then determine the parameters of the two problems we are interested in, and evaluate the best techniques for solving them.

5.2 Management Techniques for Distributed Information

In this section we describe techniques for managing information that originates on a per-machine basis. We denote each machine that supplies a portion of the total information as a server of the information, and each machine that makes a request for the information as a client of the information. Machines may be both clients and servers.

Despite the many types of information location problems that exist in distributed systems and the seemingly large number of solutions to these problems, the solutions can be reduced to a few basic techniques. The basic techniques are derived from the observation that information can be stored in three places within the distributed environment:

1. Information can be stored at its origin and not replicated in the distributed environment. Thus, information for each server machine is stored only by that machine.
2. Information can be stored close to where it is accessed. Thus, information is collected from the servers and stored on each client machine.

3. Information can be stored by machines well-known to both clients and servers. Thus, information is collected by intermediate machines, known as masters, that either know all information or cooperate to share all information among themselves.

Three of the basic techniques for managing and accessing information are direct consequences of these methods for storing information. A fourth technique is available if the specific server machine that contains the information is known. In addition, variations of these techniques exist such as caching information or giving preference to information on the client machine. In the following we name, describe, and illustrate each of these techniques. We also discuss the appropriateness of each for detection and measurement of negative information.

5.2.1 Direct Access

Direct access is used when the location of the information is given or can be derived from the name of the information requested. For example, in the IBIS [TR84] and Newcastle Connection [BMR82] file systems, the location of the file is explicitly included as part of the file name. In Sun's NFS [SGK*85] and the Tilde file naming mechanism [Dro86], the location of the file is not explicitly given but can be determined by consulting a local table. When direct access is used, each server maintains its own information and only corresponds with other machines when information is requested. Negative information such as the absence of a requested file is immediately available if the server machine is up, but negative information concerning if the machine is down can only be detected by using a timeout period in waiting for the response.

5.2.2 Multicast

In situations where the location of the information cannot be determined directly, one possible technique is to let each server store its own information and let clients multicast requests to the server machines for information as needed. Multicast allows a message to be sent to a number of machines using a single address [CD85]. It is typically supported by local area network hardware, but its support by network software is currently limited, with the V-System [CZ85] as one exception; broadcast or multiple messages are typically used in place of multicast.

We use the term multicast for this technique because information is retrieved using multicast from the server machines on demand. The multicast technique is useful if the rate at which information is accessed is slow in comparison to the rate at which the information changes. Two problems exist with this technique. First, all servers get information requests even though only a small proportion may respond to the request. Second, the amount of time the client should wait for responses to the request is unknown if the number of responses is not known a priori. In this case, the client is reduced to waiting “long enough.”

This technique also makes detection of negative information difficult because the lack of response from a machine may mean it is down, the response was lost, the machine does not
supply any requested information, or it did not respond soon enough. Consequently, this
technique works best when a small, fixed number of responses is expected. The Sun RPC
rstat and rusers programs use this technique to collect information about machine and
user status in a network [SMI86]. From experience, these programs take an exceedingly
long time to timeout and require multiple broadcasts to guarantee that responses from all
active machines are received.

5.2.3 Local Store

In contrast, if the client machines collect and store all information from the servers in
the network then each client machine has a complete store of information. A server process
on each client machine maintains the store of information available in the network and all
requests for information can be satisfied on the client machine. To access information a
request is sent to the server process on the client machine, when information changes on a
server machine a multicast message must be sent to all client machines informing them of
the change. The local store technique is useful when the rate of requests is much greater
than the rate of change for the information, or when the time to access information must
be minimal. It also requires that it should be feasible for each client machine to store the
total amount of information.

Negative information is easy to detect and measure only if each client machine can
expect periodic updates of information from the server machines. In this case the client
is guaranteed to know of changes to the information and can assume the information is
unavailable if it does not receive an update over a period of time. This technique is used
by the rwhod protocol in UNIX to manage user and machine status information with
the file system as the communication medium for requests [UNI86]. From experience in
our computing environment, this protocol is not used by workstations in an environment
consisting of many machines to avoid overloading the workstations with update packets.

5.2.4 Network Master

As opposed to storing all information at each client, another technique is to designate
a well-known machine in the network as a master machine that knows all information.
Whenever information changes the server machines send updates to the network master,
and whenever information is requested the client machines send requests to the master
machine. This approach is appropriate when the total amount of information is large, and
the use of multicast or broadcast messages to all server machines is not satisfactory. An
obvious problem with this approach is vulnerability to failure if one master server is used.
To combat this problem, multiple masters can be used that duplicate or share the infor-
mation between them. Requests and updates then can either be multicast to all masters
or sent to one with masters propagating messages among themselves as necessary. As with
the previous technique, the master servers must be able to expect periodic updates to sup-
ply negative information. This technique has been used in Grapevine [BLNS82] and the
Domain Name System [Moc87] to manage information in large, distributed environments.
5.2.5 Variations

Many variations of these four basic techniques exist. Two common ones that we will discuss are the use of a local cache, and giving preference to information on the client machine.

Caching is a common technique for reducing the cost of accessing data in a distributed system [Ter87]. As opposed to the local store technique described above in which each client machine is informed of updates to the information, a cache is only used to retain information that was discovered by a previous request. Thus a cache can only be treated as a set of hints that is not necessarily complete nor accurate. However, if the cache miss rate is low, or the cost to recover from inaccurate hints is small then a cache is a valuable technique in conjunction with one of the information location techniques. The Address Resolution Protocol (ARP) for mapping between machine internet addresses and physical hardware addresses maintains a local cache on each machine to significantly reduce the number of broadcasts that must be performed [Plu82]. Cheriton and Mann [CM86a] have successfully combined caching along with multicasting in the study of a decentralized naming facility.

Another practice is to associate quality with information. Some systems give preference to information that is “good enough.” For example, Eager, Lazowska, and Zahorjan [ELZ86] propose the idea of a threshold for load sharing. If the load of the local machine is below the threshold then the local machine is used to invoke a computation, otherwise direct requests are made to one or more other machines to check their load. In Maitre d’, a load-balancing system, a threshold is also used, but if the load of the local host is not satisfactory then a server is selected using locally stored load information [Ber86]. Another example is outlined in a paper related to our work concerning services. Comer [Com85] proposes that each client machine use a local version of a service if it is available, otherwise broadcast a message to locate the service. This scheme is applicable when local services should be used whenever available.

5.2.6 Summary

These four techniques are the basic means to store and access information in a distributed environment. If the location of the information can be determined then a directed request for information is made, otherwise a non-directed search must be used. A non-directed search means multicasting a message to all server machines, or looking up the information in a store of information, either on the local or a network master machine. In addition to the four basic techniques, two common variations of these techniques are the use of a local cache, and giving preference to information on the client machine.

5.3 Evaluation Criteria

Given these different techniques for managing and accessing information we need a means for evaluating which is the “best” for a particular problem. We use an analytical analysis to derive cost functions for various evaluation criteria based on the parameters and scale of a problem. This approach provides an objective basis for comparing different
techniques for a specific problem over multiple evaluation criteria. In this section we concentrate on evaluation of the four basic techniques and then discuss how the variations we described affect the cost of these techniques.

The cost function for each criterion is expressed in terms of two types of values—problem and scale parameters. Problem parameters are dependent upon the problem being analyzed. They include the rate at which information is accessed and updated. Scale parameters are dependent on the size and composition of the distributed environment. They include the number of client and server machines in the distributed environment. By changing the scale while keeping the values of the parameters constant, techniques can be evaluated for environments of different scale.

For purposes of our research we choose to consider the following criteria for comparison of the different techniques: the number of messages sent and received per machine, the total number of messages sent on the network, the storage required at each machine, the amount of CPU time devoted to information management for each machine, the response time to obtain information, and the accuracy of the obtained information. The importance of each criterion varies for each problem so we do not place any relative weights on the criteria.

In deriving formulas for each of the criteria and techniques, we make a number of assumptions about the distributed environment to simplify the analysis.

- All requests for information are initiated from a client process on a client machine. Hence all information requests cause at least one message to be sent and received on the client machine.

- All messages are reliably received. Although this assumption is not valid for an unreliable network, we assume the number of extra messages needed to correct for unreliability is low in comparison to the total number sent.

- All messages can be encapsulated in a single network packet so the number of messages and packets is the same. Again this assumption may not always be valid, but we assume the number of additional packets is small.

- A message sent to a multicast address requires only one network packet, but is delivered to all machines in the multicast group.

- Network master machines each contain a copy of all information and no messages need to be exchanged among the master machines.

Given these assumptions, we can derive cost function formulas for each of the techniques. Two good measures for the cost of a technique for a given problem are the number of messages that are handled (sent and received) by each machine and the total number of messages sent on the network. Formulas for these costs are given in Tables 5.1 and 5.2.

The parameter \( A \) is the access rate for information per client machine. \( U \) is the update rate for information per server machine. The scale parameters \( c, s, \) and \( m \) are the number of client, server, and master machines, respectively. The parameter \( r \) is the expected number of responses from servers when a multicast message is sent. Even though \( r \) messages are
Table 5.1 Rate of Packets Handled for Each Machine

<table>
<thead>
<tr>
<th>Technique</th>
<th>per client ( (p_c) )</th>
<th>per server ( (p_s) )</th>
<th>per master ( (p_m) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Request</td>
<td>( 2A )</td>
<td>( 2A-c/s )</td>
<td>(-)</td>
</tr>
<tr>
<td>Multicast</td>
<td>( A\cdot(1+r) ) ( 1\leq r \leq s )</td>
<td>( 2A \cdot c )</td>
<td>(-)</td>
</tr>
<tr>
<td>Local Store</td>
<td>( 4A + U \cdot s )</td>
<td>( U )</td>
<td>(-)</td>
</tr>
<tr>
<td>Network Master</td>
<td>( 2A )</td>
<td>( U )</td>
<td>( 2A \cdot c + U \cdot s )</td>
</tr>
</tbody>
</table>

Table 5.2 Rate of Packets Handled on the Network

<table>
<thead>
<tr>
<th>Technique</th>
<th>Network Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Request</td>
<td>( 2A \cdot c )</td>
</tr>
<tr>
<td>Multicast</td>
<td>( A\cdot(1+s) \cdot c )</td>
</tr>
<tr>
<td>Local Store</td>
<td>( U \cdot s )</td>
</tr>
<tr>
<td>Network Master</td>
<td>( A\cdot(1+m) \cdot c + U \cdot s )</td>
</tr>
</tbody>
</table>

received by a client machine, we assume that all \( s \) server machines may respond to the request. For the network master technique a message is multicast to all master machines and the first response is used. Again we assume that all master machines will respond to the request, but the client machine will ignore all responses after the first. Directed requests are assumed to be equally distributed between the server machines. For the local store technique, each access generates two packets on the client machine, which are each handled by both the client and server process.

Another criterion for evaluation is how much CPU time is needed by each machine to store and access information. This criterion is related to the number of packets handled by each machine as given in Table 5.1, but also includes time spent storing and looking up information. If we let \( I \) be the processing time to store or lookup information (we assume the costs are comparable), and \( H \) be the processing time to handle a sent or received packet then the rate of CPU use for each technique is given in Table 5.3. The rate of handling packets for each type of machine \( p_c, p_s, \) and \( p_m \) is obtained from Table 5.1.

A criterion that is crucial for time critical problems is the response time that can be expected to retrieve information. Unfortunately, the response time is a difficult value to express because of variability in machine and network response time. We express the communication cost in terms of two parameters: \( E_{local} \), the expected round trip delay to echo a packet from the client process to a server process on the same machine; and \( E_{remote} \), the expected round trip delay to echo a packet to a server process on a remote machine. In addition to the expected round trip and lookup time, the actual response time depends on indirect factors such as network and machine load, and direct factors such as the time to load executable code for the server process into memory. We represent the additional time used because of these factors as \( V \) due to its variability. The response time for retrieving information for each of the techniques is given in Table 5.4. Remote requests require a
Table 5.3 Rate of CPU Time Use for Each Machine

<table>
<thead>
<tr>
<th>Technique</th>
<th>per client</th>
<th>per server</th>
<th>per master</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Request</td>
<td>( H \cdot \text{p}_c )</td>
<td>( A \cdot \text{c/s} + H \cdot \text{p}_s )</td>
<td>-</td>
</tr>
<tr>
<td>Multicast</td>
<td>( H \cdot \text{p}_c )</td>
<td>( A \cdot \text{c} + H \cdot \text{p}_s )</td>
<td>-</td>
</tr>
<tr>
<td>Local Store</td>
<td>( I \cdot (A + \text{U} \cdot \text{s}) + H \cdot \text{p}_c )</td>
<td>( \text{U} \cdot \text{I} + H \cdot \text{p}_s )</td>
<td>-</td>
</tr>
<tr>
<td>Network Master</td>
<td>( H \cdot \text{p}_c )</td>
<td>( \text{U} \cdot \text{I} + H \cdot \text{p}_s )</td>
<td>( I \cdot (A \cdot \text{c} + \text{U} \cdot \text{s}) + H \cdot \text{p}_m )</td>
</tr>
</tbody>
</table>

timeout period \( \tau \) in case no response is received. When a multicast message is used, as done with the multicast and network master techniques, the client must wait for the timeout period or until the correct number of responses is received. The notation \( \max_k \) denotes the maximum response time of the first \( k \) responses.

Table 5.4 Response Time to Retrieve Information

<table>
<thead>
<tr>
<th>Technique</th>
<th>per access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Request</td>
<td>( \min(\tau, E_{\text{remote}} + I + V) )</td>
</tr>
<tr>
<td>Multicast</td>
<td>( \min(\tau, \max_r(E_{\text{remote}} + I + V)) )</td>
</tr>
<tr>
<td>Local Store</td>
<td>( E_{\text{local}} + I + V )</td>
</tr>
<tr>
<td>Network Master</td>
<td>( \min(\tau, \max_l(E_{\text{remote}} + I + V)) )</td>
</tr>
</tbody>
</table>

Two other criteria are important for evaluation, but are more difficult to express in analytic form. The first, storage requirements, depends on the size of each piece of information and the total amount of information. Each of these values is extremely dependent on the problem in question. Table 5.5 indicates that for each technique, each server machine must store its own information and that the local store and network master techniques store all information at the clients, and masters respectively.

Table 5.5 Storage Required for Each Machine

<table>
<thead>
<tr>
<th>Technique</th>
<th>per client</th>
<th>per server</th>
<th>per master</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Request</td>
<td>none</td>
<td>( \Sigma(\text{local info}) )</td>
<td>-</td>
</tr>
<tr>
<td>Multicast</td>
<td>none</td>
<td>( \Sigma(\text{local info}) )</td>
<td>-</td>
</tr>
<tr>
<td>Local Store</td>
<td>( \Sigma(\text{info in net}) )</td>
<td>( \Sigma(\text{local info}) )</td>
<td>-</td>
</tr>
<tr>
<td>Network Master</td>
<td>none</td>
<td>( \Sigma(\text{local info}) )</td>
<td>( \Sigma(\text{info in net}) )</td>
</tr>
</tbody>
</table>

The second criterion is the timeliness of the information received. The "older" the information, the more likely that the information is out-of-date and inaccurate or the server machine that supplies the information is down. Table 5.6 shows that the direct request and multicast techniques respond with up-to-date information directly from its
source, but the other two techniques depend on how closely updates generated by the server machine correspond to actual changes in the information.

Table 5.6 Timeliness of the Information

<table>
<thead>
<tr>
<th>Technique</th>
<th>Information Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Request</td>
<td>up-to-date</td>
</tr>
<tr>
<td>Multicast</td>
<td>up-to-date</td>
</tr>
<tr>
<td>Local Store</td>
<td>dependent on update rate vs. actual change rate</td>
</tr>
<tr>
<td>Network Master</td>
<td>dependent on update rate vs. actual change rate</td>
</tr>
</tbody>
</table>

The formulas in these tables express the costs of the four basic information management techniques. To incorporate variations on these techniques, such as a cache on the client machine, we assume that the probability of a “hit” in the cache is $p$. If the cost for an operation without the cache is $C_{\text{normal}}$ and the cost for a cache lookup is $C_{\text{lookup}}$ then the total cost using a cache is

$$C_{\text{cache}} = C_{\text{lookup}} + (1 - p) \cdot C_{\text{normal}}$$

For example, using caching with the multicast technique and assuming negligible cost lookup for the cache, the rate of packets handled by each client is

$$C_{\text{cache}} = (1 - p) \cdot A \cdot (1 + r)$$

The formula analytically states what is intuitively known: if the cost of maintaining and looking up information in the cache is low and the hit ratio is high then a cache can significantly reduce the overall costs. The costs for the cache are the amount of storage that must be used on the client machine and the potential staleness of the cached information that is used. If local information is given preference, a similar cost results except $C_{\text{local}}$ (the cost for local information lookup) is substituted for $C_{\text{lookup}}$ and $p$ represents the probability of local information being used.

In summary, we have identified the basic techniques that can be used for information management and location in a distributed environment, and shown how these techniques are evaluated given the parameters of a specific problem. It is important to realize that there is no one technique that is satisfactory for all problems, but rather we need to understand the parameters and scale of a problem before deciding which technique to employ. Another important point is to realize that the techniques we have identified are only the basic building blocks. Other methods such as the use of caching or reduction of an information location problem to many smaller, simpler information location problems may be necessary to obtain a satisfactory solution. In the remainder of this chapter we apply our understanding of techniques and evaluation criteria to the two problems of locating service and machine status information.
5.4 Application of Information Management Techniques

In this section we shift from a discussion of management techniques and evaluation criteria for a class of problems to application of these techniques to the two problems encountered in our service execution mechanism. The various techniques can be objectively evaluated for each problem once its parameters are known. The principal parameters for the problem of managing service information are how often services are initiated compared to how often services are added or deleted. Similarly, the important parameters for managing server status information are how often the information is needed for service selection compared to how often the status of the server machines change. In this section we derive expected rates for how often service and server status information is needed in a distributed environment. In succeeding sections we examine the other parameters and possible techniques for each problem.

Unfortunately, data concerning the rate of use and mix of computations in a distributed environment is limited. Sheltzer and Popek [SP86] gathered data on command use frequency in local Locus systems as representative of interactive computing demands in a distributed environment; other studies gathered similar information on traditional UNIX time-sharing systems [HKF84,Kri87a]. These studies concentrated on computations initiated from a shell, but we were also interested in the rate of initiation and computations specified by users in other contexts such as shell scripts.

To have control over what data was gathered we used readily available information from existing time-shared UNIX systems at Purdue. We believe this approach was valid because we use the obtained values only as guidelines of system activity in evaluating different techniques. In the discussion we consider the effect of deviations from these values. In addition, the use of data from a single-machine interactive environment to approximate service use in a distributed environment appears to have validity judging from the similar results obtained by the Locus and the other studies.

To compute the expected rate of service initiation in a distributed environment we gathered system accounting information for processes from our VAX machines running UNIX 4.3BSD in the computer science department at Purdue. The point of our study was to identify processes that were initiated as UNIX commands and would be initiated as services in a distributed environment. Because services are computations available to a user, the set of UNIX commands represents services offered by a particular machine. We were not only interested in commands initiated directly from a shell, but also commands that may be specified by the user for initiation within shell scripts or other commands such as make. Our strategy for using the process data was to filter out all processes that we would not expect to be used as services, and classify the remaining processes into different types of services. The classifications were used to determine what proportion of services would be invoked on server machines and thus need server status information during service selection to select the appropriate machine.

We collected process data from six department machines during two one-week periods in February and April, 1987. The six machines included four VAX 11/780’s, a VAX 11/785 and a VAX 8600. At the time the data was gathered these machines provided the majority of computing resources in the department for both research and administration. One of
the machines was used principally for the teaching of an operating systems course using XINU [Com84,Com87]. To eliminate processes that definitely would not be initiated as services, we filtered the data to discard processes that were not invoked by users (but by system accounts), and processes that were not associated with a terminal. We also eliminated processes that would not have been directly specified for execution by the user such as shell and other processes initiated indirectly as the result of user requests. The results were 958 unique process names invoked a total of 424,869 times by 204 users. The results are shown in Table 5.7 with processes grouped together that provide a similar class of service. The table shows the number and frequency of processes within each class.

The point of classifying the process data was to determine what percentage of these processes initiated as services would be invoked on the client machine as opposed to a server machine. Those services invoked on server machines would be expected to need server status information during service selection. We grouped the services within each classification into three groups based on where (client or server machine) they could be expected to be invoked.

1. **Client**, which are interactive, decision, or non-CPU-intensive services that are invoked on the client workstation. This group also includes the class of “other” processes that we assume contains user-specific services and binary files.

2. **Data**, which are services that search, sort, and transform data whose costs are dependent on the amount of input data. These services might be invoked either on the client or a server machine depending on the cost of moving data in relation to the cost of remote service invocation.

3. **Server**, which are specialized or CPU-intensive services that are invoked on a server machine.

Table 5.7 shows the expected frequency of use for each of the service classes and groups with dashes indicating no services in the group. The totals indicate that roughly 10–20% of the total number of services would be expected to be invoked on a server machine depending on the invocation location of data dependent services. We conclude that server status information would be needed approximately 10% of the time for service selection because data dependent services dealing with a small amount of data would be invoked on the client and services provided by only one server machine would not need status information for selection.

To determine the specific rates at which users initiate services and need server status information for service selection, we applied the classifications given in Table 5.7 to usage patterns of specific users in the data we gathered. Rather than use the rate of service initiation over the entire day, we were interested in the peak usage period of the computing system. To determine the peak usage period we plotted the number of processes invoked every hour during the day over all data collected. The results are shown in Figure 5.1. The graph indicates that the period 1:00–5:00 in the afternoon (Monday–Friday from further examination of the data) is the peak time period for user activity.

---

1 Less than 0.5% of the process information was discarded (mostly long, session oriented processes such as remote logins and editors) in the course of accumulating the data.
### Table 5.7 Classification of UNIX Processes as Services

<table>
<thead>
<tr>
<th>class</th>
<th>usage (N=958)</th>
<th>frequency (N=424869)</th>
<th>frequency by groups</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>%</td>
<td>#</td>
</tr>
<tr>
<td>UNIX(^a)</td>
<td>156</td>
<td>16.3</td>
<td>268706</td>
</tr>
<tr>
<td>text editors</td>
<td>10</td>
<td>1.0</td>
<td>28342</td>
</tr>
<tr>
<td>Purdue(^b)</td>
<td>51</td>
<td>5.3</td>
<td>23752</td>
</tr>
<tr>
<td>compilers</td>
<td>22</td>
<td>2.3</td>
<td>19362</td>
</tr>
<tr>
<td>mail handlers</td>
<td>27</td>
<td>2.8</td>
<td>23213</td>
</tr>
<tr>
<td>text processing</td>
<td>32</td>
<td>3.3</td>
<td>10635</td>
</tr>
<tr>
<td>XINU compiler and tools</td>
<td>11</td>
<td>1.1</td>
<td>8114</td>
</tr>
<tr>
<td>news and messages</td>
<td>5</td>
<td>0.5</td>
<td>5625</td>
</tr>
<tr>
<td>games</td>
<td>15</td>
<td>1.6</td>
<td>2334</td>
</tr>
<tr>
<td>revision control system</td>
<td>7</td>
<td>0.7</td>
<td>1572</td>
</tr>
<tr>
<td>other(^c)</td>
<td>622</td>
<td>64.9</td>
<td>32214</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) Unclassified standard UNIX commands.

\(^b\) Unclassified non-standard UNIX commands available within the department.

\(^c\) Process names unknown as UNIX commands.

---

**Figure 5.1** Frequency of Process Invocation During an Average Day
Analysis of specific users over the peak time period yielded the results shown in Table 5.8. The table indicates that on the average an active user initiates 20 services per hour with a high (not necessarily maximum) of 80 services per hour. Assuming that each user resides at a client machine, the typical rate is used in analysis over all client machines, while the high rate is used for the worst-case analysis of a single client machine. Application of the service classifications to the data results in the rate of 2 services invoked on server machines per hour, with a high of 10 per hour. We use these rates for how often server status information is needed by a single client machine.

<table>
<thead>
<tr>
<th>Table 5.8 Access Rate per User for Service and Server Status Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Information</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Server Status Information</td>
</tr>
</tbody>
</table>

5.5 Locating Service Information

In the previous section we derived an expected access rate for service initiation, which is the rate at which service information is needed for resolution and location. In this section we determine the expected rate at which services change, and use the access/update rate ratio as a guide in identifying possible information management techniques for service information. After determining additional parameters, we use the criteria given in Section 5.3 to evaluate which technique is best for this problem.

To measure how often services could be expected to change, meaning how often new services become available or old services are deleted, we monitored how often UNIX commands in public directories were added and deleted on the principal computing machine in our department. In addition, we included in the study the user-specific commands of five principal users of the system. The study occurred over an 8-week period in July and August, 1987. Over this period 102 commands were added and deleted with a single day maximum of 40 additions/deletions. Discarding data from weekends when no updates were recorded, the average number of changes in a single day was 2.55, which corresponds to an hourly rate of roughly 0.1. This low figure agrees with our intuition that commands on a single machine, and services in a distributed environment change little. For our evaluation we assume that services change on each server machine at the rate of 0.1 updates/hour. Even this figure may be high because it assumes that each change generates an update when in fact many changes may be grouped into a single update.

5.5.1 Service Information Management Techniques

Given the high access/update rate ratio and that we can expect most service invocations to occur on the client machine, the techniques used for managing the information should keep the service information readily available for each client machine. In considering the
management techniques we eliminate the direct access technique because the location of the service information is not known. The local store technique appears a good choice because it keeps all of the service information at the client machines.

In their basic form, neither the multicast nor the network master technique appear useful for managing information because each technique requires at least one message be sent to a server or master machine to get non-local service information. However, if we consider combining each of these techniques with caching so that most service information can be found on the client machine then each is a possible approach. A problem with caching service information is that service instances or new services with the same partial name as cached services may become available, but because the cache on the client machine is used these new instances and services are never considered. To avoid this problem, we postulate hybrid techniques that cache services as they are used, but also cause servers to multicast updates to the clients. A client machine then only needs to store the updates for services that are in its cache. Information not in the cache is retrieved through a multicast message, either to all servers or just the masters depending on the technique.

5.5.2 Evaluation of the Techniques

To evaluate the three resultant techniques—local store, hybrid multicast with caching, and hybrid network master with caching—we need the values of the problem parameters. Outside of the access and update rate, the values are implementation dependent. The values we use in the analysis were obtained from our prototype implementation described in the next chapter. All packets were sent with the UDP protocol [Pos80]. The measured time for echoing a packet on the local machine $E_{\text{local}}$ was approximately 5 ms on a Sun-3 workstation. The time to echo a packet to a remote machine $E_{\text{remote}}$ varied from 7-17 ms using a Sun-3 as the client machine. The fastest time was between two Sun-3's, the slowest time between a Sun-3 and a VAX 11/780; slower times resulted when using other machines as the client. For the analysis we use an “average” time of 12 ms. The CPU time to handle a packet, $H$, was measured at 2.5 ms and the CPU time to store or retrieve the information, $I$, at 10 ms. The variable time to lookup information, $V$, was zero in the best case, but in the worst case was on the order of seconds for remote access if the remote machine had a high load or the remote server process was paged out.

Using the parameter values and the formulas for each of the evaluation criterion we plot the expected performance of each technique as the number of machines in the network is allowed to vary. We assume that 90% of the machines are client machines, and 10% are server machines. We assume for the network master technique that 3 master machines exist in the network. The plots are shown in Figure 5.2 with the number of machines varied up to 200. We assume that 75% of the client machines are active based on observations that 75% of the users were active at peak usage periods during data collection.

In analyzing the plots, we see that the expected costs of the three techniques vary little for most of the criterion. The amount of storage (Figure 5.2a) used by each client machine is a minimum of 33K bytes because UNIX workstations in our prototype contain a large number (~500) of local services that can be invoked. Additional storage is needed as more service instances are stored from server machines. As expected the cost to store
Figure 5.2 Criteria for Service Information Management Techniques
all information about services is higher than only storing it for those services that have been used.

In generating the plots we assume the cache hit ratio is high (0.98) which is likely to be achieved after a few hours of steady work. The resident set size for services used by a user was observed from the data we collected to be approximately 100 services. The hit ratio is important because a miss requires a multicast message to be sent either to all servers or the master machines to get the service information. For a high hit ratio the number of packets generated by the multicast technique is not unreasonable, but the number of packets for this technique is sensitive to the cache hit ratio. For example, with a hit ratio of 0.5 the number of network and per server packets increases 25 times over a ratio of 0.98.

Other than storage and number of packets, the other criterion have little or no increase in cost as the number of machines increase. These criteria can be evaluated for a few machines, such as a prototype system, with the experience applicable to a larger system. The most relevant of these criteria is response time, shown in Figure 5.2e. As shown, the response time is virtually the same for all techniques for a high hit ratio. However the graph does not show the variability in response time as the cache is getting filled from information on server machines.

One criterion not shown by the plots is the timeliness of the information. If updates generated by the servers closely match the actual changes then the information maintained on the client machines is up-to-date or will be in a short time. However, a problem with a cache or complete local store is that inaccurate information due to a down machine may not be detected until the service is actually invoked on the machine. This problem does not appear to be too costly because in our computing environment machines go down on the average of about once a week.

Another problem with infrequent updates is the slow accumulation of information by client machines that come up. One solution is to require periodic updates by the server machines regardless of any changes. This solution does not appear satisfactory because it may significantly increase the rate of the update messages. The other, more satisfactory, solution is to require newly booted client machines to multicast an "update request" message to servers asking for updates, thus using update messages only when needed.

5.5.3 Summary

In summary, there is little difference in the performance and cost of the three techniques if the hit ratio is high. The difference is whether all service information, or only the information that has been used, should be stored on the client machine at the cost of retrieving information as needed. The use of a cache causes more packets to be generated and the expected response time to increase and be more variable if the cache hit ratio is smaller, which can be expected at startup time. Consequently we choose the local store technique because it provides the most consistent response time and the number of network packets is not dependent on the hit ratio or the number of client machines in the network. This technique requires the most storage for each client machine, but from experience with our prototype the storage requirements are feasible for even a relatively slow workstation (Sun-2) with relatively small amount of memory (4MB).
In addition, the local storage of service information facilitates the use of query operations on the set of services. For example, the local store can be used to list the services available to the user, or list services with a particular class attribute. The local store may also be used by the user interface to provide features such as service name completion.

5.6 Locating Server Status Information

Having chosen the local store technique to locate service information used by the service resolution and location components, we need to determine how to locate server status information for service selection. We previously derived the expected access rate for server status information. In this section we define what it means for the status of a machine to change and derive an expected rate at which status information changes and therefore needs to be updated. As before, we identify the techniques that are applicable for the problem and determine the best technique based on the parameters of this problem.

The principal measure of the status of a server machine is its load, which is defined as the number of processes ready for execution. One approach for measuring the rate of change of the load is to realize that the load is dynamic and updates can be generated as often as it is sampled. However, we take the approach that small load changes between samplings are not important and updates only need to be reported when the load difference is "significant." A significant load difference (SLD) occurs when the current sampled load of a machine and the last reported load differ by more than a certain value. This value may be fixed or vary depending on the load itself. To test different values, we gathered load data from three of our department VAX machines over seven weekdays in August, 1987. As before, the data was restricted to the period from 1:00-5:00 in the afternoon to test under peak usage. The load information is a one-minute load average that was sampled every minute and recorded.

To determine the update rate for load information we first filtered the data through SLD's of fixed values and recorded the number of updates that would be reported for each. In addition, we filtered the data through a variable SLD that was 25% of the last reported load, with a minimum value of 0.50. The rationale behind this approach was that load changes are less significant as the load increases. The results of our study for fixed SLD's are shown in Figure 5.3.

The lines in the figure for the fixed SLD values are labeled with the machine type, processing power, and average load for each of the three machines. As expected the machine with the highest load was also the most variable and consistently generated the most updates. Although also loaded, the more powerful machine generated fewer updates because its load was less variable. In comparison, the variable SLD approach yielded update/hour values of 15.9, 11.8, and 7.6 for the machines indicated by the dashed, dotted, and solid lines, respectively. This approach generated updates at the same rate as the 0.50 fixed SLD for low loads, but generated fewer updates as the load increased. This approach appears desirable for use in selecting a server because finer granularity is more useful for low loads, and once the load becomes high then that server machine would be less likely to be selected. Therefore, we assume the variable SLD approach and use the median value
from the three machines to obtain an update rate of 12 updates/hour for evaluation of the different techniques.

5.6.1 Server Status Information Management Techniques

Any of the four basic techniques discussed in Section 5.2 could be used because the server machines that are under consideration for service selection are known from the location component. The multicast, local store, and network master techniques allow for the load from all machines to be gathered and the machine with the best processing/load ratio can be selected. However, in the case of the direct request technique, many servers may offer a service and a request would have to be sent to each in order to select the best. Instead we modify the technique with a threshold as outlined by Eager, Lazowska, and Zahorjan [ELZ86]. Their idea is to select the client machine if it offers the service and its load is below a predetermined threshold. Otherwise, probes (direct requests) to servers are made until a server with a load less than the threshold is found (up to a fixed probe limit). This technique does not necessarily choose the best server, but seeks to minimize the cost of information retrieval in a large network. As an estimate in our evaluation we assume that 1.5 probes per selection are needed on the average to find a server with a load less than the threshold. This figure is derived from using a probe limit of three as proposed by Eager et al. and assumes most queries will be satisfied on the first probe, with fewer queries taking two or even three probes. If not satisfied after three probes a server machine is chosen at random.
5.6.2 Evaluation of the Techniques

Given values for the access and update parameters for the server status problem, the values of the other parameters are almost the same as given before. The CPU time to retrieve the information $I$ is smaller at 2 ms. The evaluation criteria for each of the four techniques is shown in Figure 5.4. The costs for gathering service information are included as a base cost, so the plots indicate the total costs for the service execution mechanism if each of the different techniques for managing server status information is used.

The multicast technique generates the largest number of packets handled by each server and the network, and is the most sensitive to the request rate. The number of packets on the network also increases rapidly as the number of machines grow for this technique. In terms of packets on the network, the local store approach is the least costly, however the number of packets handled per client machine is dependent on the number of machines in the network. The local store approach is attractive because there is no increase in the total response time to select a machine. The network master technique has similar characteristics as the local store except each client machine does not handle so many packets, but each access of information costs an additional network access. The direct request technique has potentially the most variable cost response time because it must send and receive messages from one or more server machines. The most appealing aspect of this technique is its use of directed requests for information, which minimize the number of packets received by client and server machines. In all cases the amount of additional storage needed by the client machines is negligible.

Another criterion that is important, but not plotted is the accuracy of information used to make the selection decision. The direct request approach obtains up-to-date information, but only collects information until a satisfactory value is found. The multicast technique collects up-to-date information from as many machines as deemed necessary. The local store and network master techniques each store all information, but its accuracy varies depending on the update rate.

To detect down machines the local store and network master techniques must guarantee that servers multicast updates at periodic intervals. Additional tests with the load data indicates that a guaranteed update every 10 minutes increases the total number of updates from 12 to 15 per hour. The direct request technique provides immediate detection (or at least suspicion) of a down machine if a timeout occurs on a request. The multicast approach will never select a down machine, but detecting one is difficult with the approach.

5.6.3 Summary

Unlike the management of service information, the strengths and weaknesses of techniques for management of server status information are much more disparate. The evaluation of the techniques is more dependent on the scale of the network, and more sensitive to any changes in the parameter values. Because of its sensitivity to the size of the network and the number of requests, the multicast technique does not appear appropriate for this problem, especially if the rate of information requests is increased. The local store technique is good because it is impervious to the request rate, but its dependence on the size of the network and the rate of updates is not attractive. Although our analysis for update
Figure 5.4 Criteria for Server Status Information Management Techniques
rate is worst case, it is not clear that the load behavior of a single time-shared machine will be the same as a server machine in a distributed environment. The network master technique decreases the load on the client machines from receiving updates, but places the load on the master machines as well as requiring a network access for each request.

With these factors in mind, the best technique for the problem appears to be the direct request approach with a threshold. This technique holds down the number of messages sent and received by each machine while finding up-to-date information about possible server machines. The technique also detects possibly down machines before service invocation. The chief drawback to this approach is determining a suitable threshold. One idea is to use a hybrid technique where a couple of master machines monitor the load on the network machines, and then the client machines can occasionally gather this information and set the threshold at the median value. Another problem is that the retrieval of the status information takes additional time. This problem appears less serious because remote invocation also takes more time so the additional information location time is negligible.

5.7 Summary

In this chapter we have identified the problems of locating service and server status information as two instances of a class of distributed information problems. This class of problems is characterized by the inherent distributed nature of the information where each server machine supplies a portion of the total information. Replicated copies of the information may be maintained, but guaranteed consistency of the copies is not a requirement. Instead, the accuracy of the obtained information can be traded off for other costs such as response time. In addition, the dependence on some machine for each piece of information means that negative information concerning the unavailability or negative status of a resource is important to detect.

Four basic techniques for managing and locating information for this class of problems were identified and a cost formula was derived for each technique based on a set of evaluation criteria. These formulas expressed each cost in terms of the parameters and scale of a problem. By determining the parameters and scale of a specific problem the evaluation criteria allow us to make objective comparisons of different techniques for the problem. We used this result to evaluate management techniques for two specific problems of our service execution mechanism—locating service and server status information.

For management of service information we chose to use the local store technique, which stores all information at the client machine. This approach was chosen because the rate of requests for information far exceeded the rate at which information changed. In addition we expected most services used by the user to be provided by the workstation itself.

For management of server status information we chose to use the direct request technique with a threshold, in which client machines make requests for status information directly to server machines until a satisfactory status, as defined by the threshold, is found. This approach holds down the number of packets handled by each machine as the scale increases. It has the highest response time to obtain information, but remote invocation also takes more time so the additional information location time is negligible.
6. SERVICE EXECUTION MECHANISM PROTOTYPES

6.1 Introduction

This chapter describes two prototype implementations of the service execution mechanism. Each was constructed to test the design of our service execution mechanism and evaluate its practicality for executing computations in a distributed environment.

The testbed for constructing the prototypes was a set of VAX machines, Sun diskless workstations, and Sun file servers each running a version of the UNIX operating system augmented with Sun's Network File System (NFS) [SGK*85]. NFS allows file systems from remote machines to be added to a machine's local file name space thus providing transparent access to remote files. The machines were interconnected with a single 10 Mbps Ethernet network. This environment was used because of its availability and our familiarity with building applications in the UNIX environment. In addition, the UNIX environment with NFS provided a rich set of services that could operate on files and data shared between a number of machines. In the summary we discuss the implications of implementing the service mechanism in environments other than UNIX and environments where files are not shared between machines.

The first prototype, Proto-A, was constructed early in the design process and implemented our preliminary ideas for a service execution mechanism [Wil86]. In addition, this prototype incorporated the use of the Tilde file naming mechanism for naming files in a distributed environment [CM86b, Dro86]. The availability of this prototype allowed us to test the feasibility of treating computations executed by the user as services within the Tilde file naming environment.

The second prototype Proto-B was constructed based on lessons learned from Proto-A and from further definition of our service execution design. The construction of Proto-B evolved with changes in the design. Its final form is true to the design described in Chapters 4 and 5. It is currently in use within a subset of the experimental computing environment at Purdue. Proto-B does not use the Tilde file naming mechanism to share files, but instead uses NFS directly to share files in a global name space between machines.

6.2 The Proto-A Prototype

Proto-A was constructed to provide us experience with an implementation of the service execution mechanism. In this section we briefly describe the Tilde file naming mechanism used as the implementation environment for the prototype, then describe the organization and implementation of the components for the service execution mechanism. We conclude with a summary of our experience with and possible improvements for this prototype.
6.2.1 Implementation Environment

We constructed Proto-A within a prototype environment of the Tilde file naming mechanism. Tilde file naming breaks the name evaluation procedure for files into two components: a per-process local naming environment, and a global access transparent mechanism. Rather than a global or hardware-dependent name space, the Tilde naming system organizes network files into collections of related files known as Tilde Trees. Each Tilde Tree is organized in a hierarchical manner, much like the file system in UNIX. For example, the name of a Tilde Tree would be `-system` and a file within the Tree would be `-system/db/passwd`. Each Tilde Tree name is bound to an underlying Medusa Name, which is a universally known name, independent of any particular network component. Associated with each process is a Tilde Forest, a set of bindings between Tilde Names and Medusa Names that describes the file naming environment for the process.

An implementation of Tilde naming was constructed by modifying the UNIX 4.2BSD operating system running on a network of VAX computers. The implementation incorporated the Tilde naming mechanism into the UNIX kernel and associated with each process a Tilde Forest that was used to evaluate all file names. In order to gain access to remote files, NFS was used as the underlying file access mechanism. For the implementation, file systems between machines were mounted and named in a similar fashion on all machines in order to ensure the same Medusa Name on one machine referred to the same physical file as a Medusa Name on another machine (see [Dro86] for more details). Thus, the Tilde file naming environment was supported by an underlying, network-wide remote file access mechanism.

6.2.2 Prototype Organization and Management of Information

Proto-A was implemented by adding a service server and cycle server, executing as separate user processes, to each participating machine. Each machine was running the modified UNIX kernel that supported the Tilde naming mechanism. For this prototype all machines were both clients and servers of services. In the implementation, both the service and server status information was managed by the service servers with the local store technique so that each machine knew its own service and load information as well as this information from other machines in the computing engine. Each machine communicated its information to other machines by broadcasting (no multicast facilities existed) an update message every minute. An update message contained any new or deleted services since the last update and a machine's load and computing power. The other service servers used the messages to update their local store of information. If an update was not received from a machine for a designated time period then this machine was assumed to have failed.

6.2.3 Service Registration

Before services could be used they had to be registered. For Proto-A the set of computations was restricted to commands so the underlying names for all services were the names of executable files. A service was registered using the routine `addserv` to communicate with the service server on the machine that should provide the service. `Addserv`
was used by both system administrators and users to bind a compiled program or shell script to a service name. `addserv` registered a service by passing to the local service server three pieces of information. First, `addserv` passed the name of the service. Service names were of the form `action-class` where the `class` was either a user id or a class name given by the system administrator. Only the system administrator could register services with a `class` other than a user id. Second, `addserv` passed to the service server the name of the executable file and the Medusa Name bound to the file's Tilde Name. To maintain familiarity for UNIX users the `action` portion of the service name was commonly given the same name as the last component of the underlying file name. Third, `addserv` passed whether this service was available to users on other machines or only users on the local machine. Services that could be computation-intensive such as compilers, text processors, and data transformation services were registered on server machines as available to users on client machines.

In addition to `addserv`, `rmsero` unregistered services, and `lsserv` listed the available services. These three routines were registered (using `addserv` as a UNIX command) as local services on each machine and used to manipulate the available services of the machine. The service registration information was not only stored by the service server, but also stored in the file system so it could be recovered when the machine failed and the server was restarted. A startup file of system information such as the processing power of the machine was also read on startup.

6.2.4 Service Execution Interface and Components

In a UNIX system, the system interface to invoke a command is the `execve()` system call, which overwrites a process image with a new piece of data and executable code. The `execve()` call is commonly invoked after a `fork()` system call creates a new “child” process. To incorporate the service abstraction in a compatible manner, a new library function `sexecve()` was added. The new function could be substituted for `execve()` to execute services rather than commands. The arguments to `sexecve()` were a service name (partial or complete), the service parameters, and the environment variable values for the service to be executed. The only difference between the interface of the two procedures was that `sexecve()` allowed partial service names to be given.

To make services readily available to users the obvious solution was to substitute `sexecve()` for `execve()` in a UNIX command interpreter. Thus `sexecve()` was substituted in the UNIX command interpreter `csh` [UNI86], and known as the service shell, or `scsh`. The `csh` was the principal user interface for initiating services in Proto-A, although a distributed C compiler service `scce` that treated each phase of the compilation as a service was also created.

The use of `sexecve()` made the details of service execution transparent to the user. It worked by first contacting the local service server and passing a service name and the

---

1 All communication with a service server was done using a UDP-based protocol [Pop80].
2 As with `execve()`, many alternate forms for `sexecve()` existed to execute a service.
3 As an escape to the UNIX system, the service shell checked if the given service name was actually a full path file name or a file located in the current directory. If so then the file was loaded directly. No other changes were made to the shell.
service set in case resolution was needed. If a complete service name such as *tex-std* was given, the service server located all available server machines and then selected the "best" machine (based on load to processing power ratio) for invocation. If a partial service name was given the service server first resolved the name to a complete service name by iterating through the service set until an available service name was generated. The store of local services and server load information was then used to locate and select a machine for invocation of the service.

6.2.5 Service Invocation

After selecting a machine for invocation of the service, the service server passed back to the client the name of the machine and the complete service name. In addition, if the local machine was selected then the underlying file name and its associated Medusa Name were also passed back to the client and a process was spawned that loaded the file directly for invocation of the service. If not already present in the Tilde Forest, the Tilde Tree and Medusa Name binding for the file to be invoked needed to be added to the Tilde Forest to provide access to the executable file. A problem occurred if this Tilde Tree conflicted with any in the client’s Tilde Forest. Any conflicts were resolved by binding a different Tilde Name to the Medusa Name of the executable file and changing the name of the file accordingly.

If a remote machine was selected, a client process was spawned to serve as the client for communication with the remote cycle server. The client process communicated with the remote cycle server using a TCP-based protocol [Pos81]. The protocol was similar to the stream-oriented *sh protocol in UNIX except the cycle server invoked the service with an execution environment passed to it by the client process. When invoking a service, the client process connected to the appropriate cycle server and sent the user id, service name and parameters, current working directory, environment variables, and Tilde Forest. The cycle server then used the connection for exchanging input and output data with the client process.

To obtain the underlying file name for invoking the service, the cycle server on the remote machine sent a message to its local service server and received the Medusa Name and Tilde Name of the file to be loaded. Before loading the file, the execution environment for the spawned child process was set with the Tilde Forest, the current working directory, and the environment variables passed from the client process. The Tilde Tree for the file to be loaded was also added to the Tilde Forest of the child process with any conflicts in the Tilde bindings resolved as previously discussed.

6.2.6 Experience with Proto-A

Proto-A provided us with valuable experience concerning the feasibility of treating all computations initiated by a user as services. We found that the increased overhead due to contacting the service server was not significant for services invoked on the local machine.

---

4 The service set was maintained by each process as an environment variable.
5 The user could also force the execution of a service on a specific machine by preceding the service name with *machine:*.
while services from remote server machines were invoked in a transparent manner. The resultant environment allowed the user transparent access to all services in the computing engine at no visible increased cost for local services.

For the small scale of the prototype the frequent generation of update messages by each server machine was not a significant factor for the machines or the network. The use of stored server status information allowed service selection to be performed on each client machine, but resulted in using load information that was slightly out-of-date. A more serious problem with the implementation of the periodic updates was that they could be lost and not recovered so client machines could miss services advertised by other machines.

The prototype also provided us with much experience using the Tilde file naming mechanism for describing the file naming environment of a process in a distributed system. We found Tilde file naming to work well for the distributed environment with our strategy satisfactory for resolving Tilde Tree conflicts between the client process and the Tilde Tree of the executable file. The use of the Tilde file naming mechanism did limit our prototype to a set of VAX machines because the mechanism was only available on those machines.

### 6.2.7 Improvements on Proto-A

In constructing and using the Proto-A prototype we learned a number of other lessons and discovered some shortcomings that needed to be improved about our service execution mechanism. These lessons are summarized in the following.

A simple two-attribute name space generally worked well for naming services. A problem with only two attributes was that users had to use their user id for the class name and could not classify their own services separately. This problem necessitated changing our design to include three attributes for a service name. Another problem with service names was the inability to express functional relationships between two services other than "the same." This observation led to the development of metaservices.

Two problems were encountered in the prototype concerning how a programmer should build and test new services. First, machine or architecture dependent services such as compilers were difficult to name. For example, the standard C compiler on two different architecture machines would intuitively be given the name `cc-std`, but these two services are not the same. Although each takes the same input, the output produced is architecture specific. Either another way to name such a service or a method to choose between two such services must be found. The second problem was how to execute the binary files generated by the compiler before the files were registered as services. In the `scsh` we searched for an executable file in the current directory before trying to execute a given name. However, to maintain our service abstraction, the correct solution would be a "run" service that invokes a named executable file, although again this service is architecture dependent.

The simple notion that services were either local (available only to users on the local machine) or global (available to users on all machines) is not extensible to a larger, heterogeneous environment. Users may only have authority to execute services from particular machines, or some services may only be available to machines that share access to the same files. These observations led to the registrar supplying additional accessibility attributes for each service at registration time.
A major limitation of the prototype was its restriction that all services were invoked as an executable file and that only one service invocation protocol could be used to invoke remote services. A range of invocation protocols capable of different levels of environment sharing and authentication would be preferable. In addition, different protocols and other formats for the underlying name would allow other types of computations to be treated as services.

6.3 The Proto-B Prototype

The purpose of our second prototype was to correct problems identified from Proto-A and to test additional refinements in our service execution design. In addition, we wanted to test the use of workstations as the client machines for the computing engine. The overall organization of the service mechanism remained almost the same, however, many details of the service mechanism were changed. In this section we describe the implementation environment for Proto-B then describe the details of the prototype itself, and conclude with a summary of our experience with and possible improvements for the prototype. The prototype is currently in use within a subset of the experimental computing environment at Purdue.

6.3.1 Implementation Environment

Proto-B was not implemented using the Tilde file naming mechanism to share files, but instead used a partial global file naming convention established within our department to name network files. The name space for files is divided into two name spaces—a user file space that is shared with NFS mounts among VAX (UNIX 4.3BSD) and Sun (Sun OS3.2,3.4) machines, and a system file space that is particular to each machine. A representation of this environment is given in Figure 6.1.

![Figure 6.1 Implementation Environment for Prototype Proto-B](image-url)
The user file space contains directories of all user files. It is composed of file systems named in the form /ui where each machine exports such file systems it contains and machines mount these file systems in their local name space with the same name. The user file systems are shaded in Figure 6.1. The system file systems are outside of the user file space and particular to each machine. The unshaded, unnamed file systems shown in Figure 6.1 contain system specific files for each machine, including workstations.

For the prototype we assume that any server machine that mounts the file system providing a user's current file context can be used to provide services that require access to named files. We discuss the implications of this assumption in our experience with the prototype.

6.3.2 Prototype Organization and Management of Information

Unlike Proto-A, the service server and cycle servers are implemented as a single UNIX process in Proto-B, although we will refer to them as conceptually separate servers in the discussion. Because UNIX processes cannot share data, this change was necessary to allow a cycle server to quickly map a service name to the underlying name without having to send a separate message to the service server. Multiple invocation protocols are supported in Proto-B with a request for service invocation causing a new process to be spawned and the cycle server supporting the protocol for invocation of the service used to interact with the remote client process. In the prototype, the VAX machines and Sun file servers are both server and client machines while the workstations function only as client machines and do not require any cycle servers.

Service information is stored by the service server on each machine just as Proto-A, with the services provided by that machine also stored in the file system. On startup, the service server reads the services provided on its machine from disk to initialize its local store. Services provided by server machines in the computing engine are obtained from update messages sent by the service servers on those machines. Update messages contain information about any services that have been added or deleted since the last update. In addition, the status of the server machine such as load and processing power is also included in the update. To avoid lost updates in Proto-B, each update is timestamped with a start and end time over which the update is valid. This technique is analogous to publishing volumes of a book and allows each client to easily detect when it misses an update message from a machine. Recovery from such a situation either requires the server machine to resend prior updates or in the worst case causes the complete list of services offered by the server machine to be sent. The latter case is typically the scenario when a new client machine starts up, although client machines may also store previous timestamps and services available from other machines in their file system to make startup faster.

Another change from Proto-A is updates are no longer guaranteed to be broadcast at periodic intervals. In Proto-B, load information is typically found by probes so updates are only sent when services are added or deleted. This change required a modified protocol for a client machine at startup because it is not guaranteed an update message will come.

Some system file systems are also shared between machines using NFS, but these remote mounts are not relevant for our prototype.
from each server machine soon after startup. The solution is for it to broadcast an “update request” message to the server machines that causes an update message to be broadcast by each server machine over the next minute. For reliability the update request message is repeated three minutes later so after a few minutes a new client machine knows about the services available in the computing engine.

6.3.3 Service Registration

Addserv, rmsserv, and lsserv are again used to register and manage services in the prototype. Services have names of the form action-authority-class, but the names are often shortened to two attributes as described in Section 4.5.2. In addition, services can be metaservices, which allows one service name to point to one or other services. Additional information is required at registration time besides the name of the service and the underlying name for invoking it. This information is the availability of the service, if the user needs to have authorization on the machine offering the service, and the invocation protocol.

Services have three types of availability—to users on the local machine, to users on machines that share the same global file space as the machine, or to users on any machine that implements the needed invocation protocol. For example, a tape archive service that uses a local device fits into the first group, a compiler service that accesses user files in the second, and a weather information service that does not depend on the user’s environment in the third. In addition, services can be restricted to only users who have accounts on the machine providing the service. Because each machine is autonomous, users may not have uniform access to all machines.

The invocation protocols currently available are a stream protocol similar to the protocol in Proto-A, a datagram protocol that passes a minimum amount of environment information and can be used for low-cost, low-output services such as date, and an interactive protocol that establishes a connection capable of supporting terminal manipulation required for highly interactive services. Although this type of service is best done on the workstation, such services may not always be available locally, and this protocol allows interactive services to be used from another machine. In addition to these three protocols, protocols to contact Internet service ports directly with either a UDP datagram or a TCP connection are also available. In these cases the underlying name is a port number rather than a file name. The invocation protocols supported by a machine and the appropriate client routines needed to invoke the protocol are included in the startup file for the machine.

Registration of a service causes it to be added to the next update message sent by the server machine, along with the invocation protocol, and who can use the service if it is available to users on other machines. The underlying name for the service is not included in the update message unless the service name itself cannot be used to invoke the service. For example, the underlying name is included for Internet services invoked at a particular port and metaservices where the underlying name is actually the equivalent service name.
6.3.4 Service Execution Interface and Components

The `sexecve()` system interface for service execution is identical to Proto-A, and again the service shell `scsh` serves as the primary user interface for initiating services. In addition, `sexecve()` was added to the UNIX utility `make` to allow computations to be conditionally performed based on file dependencies given in "makefiles" [Fel79]. The resultant service `smake` is a decision service that initiates other computations that are not limited to computations available from a single machine. `Smake` could be modified to allow parallel execution of services, since multiple server machines are available. Another decision service is `sman` that locates a machine that provides a given service, but instead of invoking the service causes documentation for the service to be retrieved. If each server machine supplies documentation for each service it offers then documentation is available for each service. A complete list of services we have created is summarized in Table 6.1.

Table 6.1 Service Related Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>smake</code></td>
<td>Allows services to be initiated based on file dependencies.</td>
</tr>
<tr>
<td><code>sman</code></td>
<td>Locates and prints documentation for a service.</td>
</tr>
<tr>
<td><code>whichserv</code></td>
<td>Determines which service is resolved and where it will be invoked given a partial service name.</td>
</tr>
<tr>
<td><code>run</code></td>
<td>Executes an executable file on a particular architecture.</td>
</tr>
<tr>
<td><code>readinfo</code></td>
<td>Causes the service server to re-read its startup file.</td>
</tr>
<tr>
<td><code>echoserv</code></td>
<td>Sends packets to a service server that are echoed for timing measurements.</td>
</tr>
<tr>
<td><code>addserv</code></td>
<td>Registers one or more services with the service server.</td>
</tr>
<tr>
<td><code>rmnserv</code></td>
<td>Unregisters one or more services with the service server.</td>
</tr>
<tr>
<td><code>lserv</code></td>
<td>Prints information about service and server information.</td>
</tr>
</tbody>
</table>

Because the available services are known at each client machine, service resolution and location is the same as Proto-A except for a few additional checks for accessibility. For example, if the service requires the user to have authorization on the server machine then a local table of user account information is consulted. If the service expects to use the file space then the service server has to check if the server machine has mounted the user's current file system. Because the calculation of the current directory is expensive in UNIX, especially for file systems on remote machines, the cost added for each execution of a service is significant. To increase performance we added another library function `sexecveinit()` that is embedded in `scsh` and called whenever the current context changes. This function caches the current working directory for subsequent calls to `sexecve()`.

---

7 User account information is obtained at startup time from a file mapping user ids to machines for which the user id is valid.

8 Mounted file system information is advertised during updates from a server machine as a bit vector with bit i set if /ui is mounted.
The selection component changed significantly from Proto-A because the technique for gathering load information changed. If the service is available on the local machine and the local load is below a threshold plus an estimated cost for remote invocation then the service is invoked locally. Otherwise, as long as multiple eligible machines exist that provide the service, a probe is sent to one of the eligible machines to check its load.\footnote{Actually load information is cached and a new probe is not sent if the cached information is less than 30 seconds old.} If a response is received within a short time and the load is below the threshold then this machine is selected for invocation. If not, the machine is eliminated from consideration and another machine probed until a machine is found or only one machine is left in which case it is chosen by default. If only one machine provides a service then no probes are sent. As the number of server machines grow we can think of adding a probe limit. In addition to finding up-to-date load information, the probes also detect down machines. If a response is not received to a probe request then the machine is marked down and an additional probe sent. If any subsequent replies (or updates) are received from this machine then it is again marked active.

6.3.5 Service Invocation

If the client machine is selected for invocation and the underlying name is an executable file then this file is loaded directly. If a remote machine is selected for invocation then the client routine that implements the invocation protocol for the service is invoked. This routine communicates the service name and details of the execution environment and authentication information to the remote cycle server. The execution environment includes the current file context and the values of environment variables. Authentication is done using the permission checking facilities of UNIX. These facilities allow services to be invoked from client machines whose names are either in a machine- or user-specific permission file on the server machine.

Another invocation problem is how to build and invoke binary files that are eventually offered as services. One problem is naming compiler services that produce architecture-specific code. To handle the problem in Proto-B we chose to make the \textit{class} attribute of such services reflect the machine architecture for which these services are valid. Hence the C compiler service for the VAX architecture is named \textit{cc-vax} with similar names for the C compiler on the Sun machines. Specifying the correct compiler service for a particular machine requires specification of the complete service name or inclusion of the correct \textit{class} in the service set for resolution.

In a similar manner, a \textit{run} service to invoke architecture dependent binary files is also available for each architecture type. However, if the name given to the shell begins with "," or "/" then the name is assumed to be an executable file to be run on the client machine. The special characters serve as an implicit invocation of the \textit{run} service on the client machine. A more general solution to invoking files would be to make \textit{run} a decision service that determines the required architecture for a binary file and invokes the binary file on an appropriate machine. An even more general solution to building and invoking executable
code would be for run to take arbitrary source code as input and either interpret or compile and invoke the code as necessary. Caching could be used to improve performance.

6.3.6 Experience with Proto-B

The overall experience with the second prototype of the service execution mechanism has been positive. The use of the service shell from a workstation provides the user personal computing power while affording him transparent access to services that are more efficiently offered by other machines. The service abstraction alleviates the user from establishing multiple "login windows" with specific machines; a situation that leads to the user often changing his current window because different machines provide different services. Instead he can have multiple windows with each window specific to related applications that are not bound to specific machines.

The use of NFS to share the global file space between machines is satisfactory. Problems occur when the user's file context is in the user file space and he attempts to use a service along with the absolute name of a file in the system file space of the local machine. If invoked on a remote machine, the service either fails or accesses the wrong file. These instances are rare because most files are specified relative to the current context and if the current file context is a system directory then only local services are used. File naming problems within the user file space were also diminished when common symbolic links such as "/usr/cew" were made to point to the same underlying name "/u3/cew" on all machines.

In analyzing the performance of the mechanism on a Sun-3 workstation, the time to send a request to the service server, have it resolve, locate, and select the service then reply to the service shell is approximately 20 ms for a local service. In comparison, the time to resolve a hashed command in csh takes virtually no time, although for an incorrect hash entry resolution takes 10-200 ms depending whether the workstation has to use NFS to examine a remote directory. Thus, in most cases the service mechanism is more expensive. To judge the increased cost of the mechanism a simple service such as one to print the date takes 150-200 ms to load and execute if the file is from a local file system. If the file is not mounted locally, but over NFS, then the time was measured at 350-400 ms. We conclude the service mechanism is a small cost in the total time to execute a simple service, and an insignificant cost for more computation-intensive services. For services that are invoked remotely, additional costs may be incurred for probing during selection, but these costs are negligible in comparison to the costs of invocation.

A problem with the prototype is deciding what threshold value to use when selecting among multiple machines. From experience the remote invocation costs are significant so the threshold is set high enough to almost always use a local instance of a service if it is available. If a local service is not available then the threshold to use while probing for a server is lower and designed to avoid selecting a machine that is heavily loaded. Our strategy tries to avoid making a bad decision, but does not always make an optimal one. One optimization for probing remote machines is to first check the machine containing the user's current file directory if that machine offers the service. This technique minimizes the file access costs if the machine is selected.
A test of the service mechanism is how well the model of all access from a single workstation can be retained in modifying and maintaining the distributed system. Services for maintaining the set of services (addserv, rmserV, lsserv, etc) are classified into their own adm class. These services only communicate with their local service server, but administrative services for each machine are made available to other machines. Hence, a user or administrator at his workstation can register services on a server machine using the addserv service from that machine. The machine used for an administrative service needs to be explicitly given and defaults to the local machine if not given. One limitation is that supervisory or “root” access is not allowed uniformly between machines so machinespecific administrative duties such as restarting servers has to be done by first establishing a session on the necessary machine and securing root access. Establishment of a session is also necessary to gain access to the system file space on specific machines.

6.3.7 Improvements on Proto-B

As stated, a problem with the performance of the prototype is the time to invoke services on remote machines with the stream or interactive protocols. With these protocols, the time to set up a TCP connection, pass the execution environment, and authenticate access for the service can take a noticeable amount of time, especially if the machines are loaded. In the prototype these protocols are adaptations of existing UNIX protocols with no additional work to optimize the protocols for this specific application. One approach for reducing the overall cost of remote invocation would be to set up connections between machines that are “persistent” and can be used for the invocation of more than one service, thus reducing the costs for subsequent invocations.

A feature of the prototype is the capability for the user to combine many services together to form a single pipeline of actions. A pipeline is a series of actions where the output data of one action is used as the input of the next action. It is a powerful tool in a UNIX environment and the prototype naturally extends the tool to a multi-machine environment. A problem with the prototype implementation of pipes is illustrated in Figure 6.2a. For a pipeline originating from the workstation using services on machines A, B, thru Z, all data must flow through the workstation. If the workstation is of comparable speed to the machines in the pipeline then it is probably not a bottleneck, but if the server machines are more powerful with a lot of data flowing between the services, then the implementation is not satisfactory.

Instead, a better implementation, presented in [KW86a], is shown in Figure 6.2b. A pipe server on each server machine is used to set up the connections as shown so that all data flows directly between the performing services (thick lines) and any error data flows back to the workstation (thin lines). This implementation requires the location of all services to be known at pipeline initialization rather than for a component at a time as in the current prototype. The advantage, as shown in [KW86a], is this method of multi-machine pipes allows for efficient exchange of large amounts of data between services by removing the workstation from the dataflow path.
6.4 Summary

In this chapter we have presented the implementation of two service execution mechanism prototypes and the experience gained from these prototypes. The service mechanism adds a relatively small overhead for locally invoked services while providing transparent access to all services whether local or remote. The service abstraction works well with NFS, with or without Tilde naming. In fact the service mechanism makes remote computations transparently available to the user at his workstation, much like NFS makes remote files transparently available to the user. Because well-known protocols are used for invocation, new services can be offered by server machines and immediately be available for users at client machines without any additional work by users on the client machines.

The use of a shared file space environment allows a large number of existing services to be shared between machines. The use of services from machines not sharing the same file name space only curtails the number of services offered because many services written for a single-machine environment depend on accessing named files for data. Proto-B is also implemented on a Tektronix workstation that does not use NFS, but uses a distributed file system implemented between Tektronix workstations. The number of remote services available to this workstation is limited in the prototype, but the service mechanism could easily take advantage of the Tektronix distributed file system to share a large number of services between a set of Tektronix workstations. In cases where services cannot access data through shared files, the use of multi-machine pipelines appears a powerful mechanism for sharing data between services.

There is no fundamental dependence on UNIX in the design. Because the binding between a service name and the underlying name is maintained by each machine, the service mechanism can be adapted to any underlying environment. Late-binding of how a service is invoked is well-suited to a heterogeneous environment. More work would have to be done on invocation protocols that would work between machines running different operating systems. For example, environment variables in UNIX provide a nice mechanism to allow each process to have a set of local bindings, but they are not understood in the same manner by all types of machines.

In other ways the UNIX environment is not well-suited for the implementation. The use of a single process to share both the service server and cycle servers would better be
served by multiple processes executing in the same address space so they could share the service information. In addition, the UDP protocol is not well-suited for message passing because of its unreliability and lack of support for multicasting. A protocol such as VMTP [Che86] that provides multicast, datagram, and reliable request-response messages would allow better implementations of all components of the service mechanism.
7. CONCLUSIONS

7.1 Review of Goals and Motivation

Our research has been directed towards an understanding of the issues involved in transparently executing computations in a distributed environment. Our approach has been to investigate the consequences of treating actions available to the user in a computing engine as services, where the identification of a service is separated from its execution. This separation allows the user to specify what services he would like to use and not be concerned with where or how the services are invoked.

The motivation of this research has been to provide users, interacting with a workstation, a consistent and uniform interface for executing actions in a distributed environment. The user should be able to initiate actions in the same manner whether the action is invoked on the workstation or on a remote machine. In contrast, current production systems require the user to establish a login session or use an explicit remote access command from the workstation to invoke commands on remote machines. Other distributed research environments provide location transparent execution of commands, but retain the model that commands are named and invoked as binary files. The service abstraction breaks the direct link between what the user wants to do and how it is done.

7.2 Results and Evaluation

7.2.1 Action Execution Model

To attack the problem we first tried to understand what were the essential steps for executing computations in a distributed environment. Using an action to denote a named computation, we studied existing action execution systems—for example command execution systems, which provide user-level actions; and remote procedure call mechanisms, which are used by programmers to construct distributed applications. The point of this study was to relate existing work and develop a framework for the steps needed in executing computations.

The result of this work was the development of the action execution model, which identifies the essential steps necessary to execute an action in a distributed environment. The model consists of seven orthogonal components whose ordering and binding time relative to specification and execution of an action characterize a system.

The initiation component initiates execution using a specification from the user. The invocation component is the last component and uses the decisions made by the other components about where and how to invoke an action. The remaining components are known as the binding components because their binding time and ordering are particular to each system. The resolution component resolves a partially specified action name into
an absolute action name. The location component locates the set of machines that offer the action, and the selection component selects one of these machines to invoke the action. The protocol component chooses the invocation protocol to invoke the action, and the mapping component maps the high-level action name to the low-level underlying name appropriate for the protocol.

We successfully modeled a number of distributed action execution systems with the action execution model including command execution, network services, remote procedure call, and object-oriented systems.

7.2.2 Service Execution Mechanism

We also applied the model to a set of principles for our service abstraction to obtain the framework for the service execution mechanism. The principles follow from our approach that the identification of a service should be separated from its execution. If we denote each instantiation of a service offered by a machine as a service instance then the principles for the service abstraction are:

- The user should only need to specify what service he would like to perform and not where or how it is invoked.
- The user should be able to specify a partial name for a service with resolution mapping it to an absolute name.
- A service may have multiple instances, each offered by a different server machine and each server machine has control over the services it offers.
- The implementation, not the functionality, of a service instance is machine specific and may not be the same for each machine offering a service.

Application of the action execution model to these principles yields the following ordering for the components of the service execution mechanism

\[
\text{INITIATE} < \text{RESOLVE} < \text{LOCATE} < \text{SELECT} < \text{PROTOCOL} < \text{MAP} < \text{INVOKE}.
\]

Using this framework we designed the service execution mechanism and built a prototype based on the design. The specific issues for each of the components and our approach for each are discussed below. For the design we were not concerned with the details of the user interface for specifying services. For the prototype we modified a command interpreter to allow users to initiate services from what we termed a service shell.

7.2.3 Service Naming and Resolution

For the service name space we used a fixed set of attributes of the form action-authority-class. The action attribute is the primary name for what the service does. The authority attribute is either a user's account name signifying that he is the authority for the service, or is sys signifying that the system administrator is the service authority. That person is
responsible to ensure that all instances of a service provide the same function. The class attribute is used to denote related services, such as all services in a subsystem or different versions of a service.

Services with the same name provide the same function. To incorporate the idea that services also have related functions we introduce the idea of a metaservice. A metaservice is a special kind of service that contains "pointers" to other services. Execution of a metaservice causes one of the other services to be invoked. Metaservices are used to group services under the same name that have similar function and may be used interchangeably when the metaservice is requested. This approach allows all services (including metaservices) to be initiated in the same manner, but requires users and system administrators to explicitly define metaservices as needed.

The action attribute is used as the partial name for a service, which is resolved into a complete name using a service set. The service set is an ordered set of authority and class attributes that are successively matched against the action attribute until an available service name within the computing engine is formed. For access to services outside of the computing engine we name each computing engine and include that name as part of the service request. The need for non-location transparent service names outside of the computing engine is a consequence of the limits of administrative boundaries for arbitrating the assignment of names to resources.

From experience, the service naming scheme works well within a computing engine, especially with metaservices to define relationships other than "the same" between services. Service resolution is a simple mechanism that allows short service names to be specified. More research is needed on the use of services between computing engines.

7.2.4 Locating Service Information

To perform service resolution and location, knowledge is needed of what and where service instances are available in the computing engine. We identified this problem as an example of a general class of distributed information problems where the information is inherently distributed across multiple machines. For these problems the accuracy of the obtained information may be traded off for quicker access to the information. We describe techniques for managing information in such an environment and present a set of evaluation criteria for evaluating the best technique to use for a problem based on its specific parameters.

In particular we use the evaluation criteria to evaluate techniques for managing service information in a computing engine. The data gathered from our computing environment at Purdue indicate that the ratio between how often service information is used versus how often it changes is very high. In addition the data indicate that up to 90% of the services executed by the user will be provided by his local workstation. These parameters lead to storing service information at each client workstation with server machines multicasting their service information to the client machines when services change. This technique (with broadcast rather than multicast) was used in the prototype implementation and worked well, although it was not tested on a large scale.
7.2.5 Service Selection

Selecting a machine to invoke a service from multiple machines that provide a service is an optimization problem. We derived a selection cost function based on the processing demands of the service itself and on the capabilities of the machine providing the service. However, because many of these factors may be difficult to obtain at selection time, and because the optimization is constrained by its own costs, selection of the optimal machine in most cases is impossible. Consequently, we chose to use heuristics that attempted to make good, not necessarily optimal, selection decisions.

Our basic approach is to invoke services on the machine with the lowest load to processing power ratio after adding in a cost for invocation on remote machines. This approach depends on the service authority, and perhaps the user, to supply additional parameters for selection. A problem is how to obtain the load information from the server machines. This problem is another instance from the class of distributed information location problems. The basic parameters for this problem are that load information changes more frequently than it is needed for service selection. The data we gathered indicate that selection of a remote machine to provide a service is only needed about 10% of the time. These parameters lead to using a “direct request with threshold” technique where direct requests to eligible server machines are repeatedly made asking for their load until a machine is found with a load less than a threshold value. Because these requests are only made for services that are not invoked by the workstation, the increased costs for service execution are not significant.

7.2.6 Service Invocation

Once a machine has been selected for invocation, the client machine invokes the service with the appropriate invocation protocol. The service invocation protocol defines a set of conventions for how a service name is mapped into the appropriate underlying name, how a service invocation request is authenticated, and what portion of the execution environment of the client process is inherited. In our prototype we defined a number of protocols for invoking different types of services. Each service is advertised along with the invocation protocol that the client machine needs to invoke the service. The principal protocol we use creates a TCP connection between the client process and the cycle server process on the selected machine. The connection is used to pass the execution environment of the client process and is then used for input to and output from the invoked service. The cycle server maps the service name to the appropriate binary file to load for execution. Other invocation protocols are also available, such as one to contact well-known Internet services like \texttt{daytime}, which provides the current time, directly with a UDP packet.

The use of a well-known set of protocols allows new services that are invoked with these protocols to be immediately available without client machines needing specialized routines for each new remote service. This is an important property in a distributed environment to reduce the amount of administrative overhead for each workstation. For a user, new services “come for free” because he has to do no work to locate or access them. The principal problem with the current set of protocols is the time spent by the client process to set up a TCP connection and pass the execution environment.
We did not investigate authentication issues in our design and used existing permission files in UNIX for our prototype. We found the use of NFS for providing a common global user file naming environment to be satisfactory both when used to name files directly and when used as a basis for the Tilde file naming mechanism. Although not all files could be shared, services could be invoked on remote machines with the same user file name space as the client machine. In almost all cases this solution was satisfactory.

7.3 Future Directions

Our experience from designing and implementing the service execution mechanism indicates a number of directions for future research.

The interface for the specification of actions to be initiated is an area for much research in a workstation environment. For the prototype we were able to use a modified shell to demonstrate our ideas for service execution, but with a workstation we can envision better means for presenting information, providing interactive help for the user for specifying services, and other enhancements for specification. These ideas coordinate our work with that done by Krishnamurthy [Kri87b]. Other future work for specification is to extend the ideas presented in Omicron [KW86b] for allowing the user to describe the conditions for when an action should be initiated. This work would give the user flexibility to say when an action is to be initiated.

Naming is an important issue in distributed systems, and it presents a number of possibilities for future research related to our work. One direction for showing relationships between services is to explore use of additional descriptive attributes to augment the fixed set of attributes we use for naming services. These attributes would allow users to detect similarities between services. Another direction of research is to extend the power of metaservices so that an evaluator function, similar to the portal mechanism in [LEH85], could be evaluated at execution time to decide which service instances to consider. For example, a metaservice may point to two services that perform the same function, but each has different options. An evaluator function could choose between the two services based on the options specified by the user.

The technique used for management of service information is based on the parameters of how often service information is expected to be accessed and updated in a distributed environment. Because we derived these parameters from timeshared systems they may not be indicative for a distributed environment. More work on gathering data on what and where services are invoked relative to the workstation needs to be done.

Service invocation on remote machines with the current invocation protocols is costly to set up the connection and pass the execution environment. One approach for reducing the overall cost of remote invocation would be to investigate use of connections between machines that are “persistent” and can be used for the invocation of more than one service, thus reducing the costs for subsequent invocations.

 Separation of interaction from processing for a remote computation is a means to take better advantage of the power of the workstation. Relating to our work, we would like the workstation to handle the interaction, but we would also like for the workstation to not have to be aware of the interaction details before the service is invoked. One idea is for
the server machine to download interaction information to the workstation at invocation time, which would allow service specific information to be used without a frontend service explicitly existing on each client machine. This approach also has the potential for allowing interaction details to be separated from the processing of the service resulting in a uniform, yet customizable, interface for the user over a range of services. This idea naturally extends the ideas in [Kri87b] to a distributed environment. A windowing system such as NeWS [GH86] that allows programs for manipulating windows and handling user interaction would be a good environment for study of these ideas.

Sharing of data in a computing engine, especially in a heterogeneous computing engine where all globally named objects such as files cannot be shared, is another direction of research. In our prototype, NFS allowed the user file space to be shared between machines of different architectures each running the UNIX operating system. Although NFS has been extended to other operating system environments, it is not a distributed file system and not all files are shared between machines. One solution for the sharing of data in a heterogeneous environment is to use pipes between services. Pipes allow unnamed data to be passed from the output stream of one process to the input stream of another process. One approach for extending the use of pipes is to allow multiple input and output streams for a process to further facilitate sharing of information between services.

Use of idle workstations could also be done with our mechanism. Although this approach breaks the model that workstations only provide services to their local user, it allows processing cycles to be used that would otherwise be wasted. After a pre-determined amount of idle time these machines could each advertise their own services or their willingness to provide services as part of a workstation pool. When the workstation user returns, the workstation ceases to become a provider of services. The problem with this idea is handling currently executing services when the workstation user returns to reclaim his workstation. Ideally any executing processes migrate to another machine, but migration of processes is not always possible.

7.4 Summary

In this dissertation we have explored a new approach for the execution of user-level actions in a distributed environment. The following is a summary of what we feel are the research contributions of this dissertation.

The action execution model succinctly describes the essential steps needed in the execution of actions in a distributed environment in terms of a set of components. Examination of the ordering and binding time of these components for a system that executes computations can be used to characterize the system. More importantly we have a common representation that can be used to compare and understand a wide variety of systems that execute computations in a distributed environment, such as commands, network services, and remote procedure calls. Application of the model to a set of design principles yields the framework of our service execution mechanism.

The design and implementation of the service execution mechanism demonstrates the feasibility of treating all actions available to the user in a distributed environment as a service. This concept allows the user to specify what services he would like to use and the
workstation to determine where and how the services are invoked. The workstation is an intelligent interface to the services of the computing engine rather than simply serving as a terminal multiplexor or a dedicated personal computer. The use of well-known invocation protocols allows new services to be executed by the user without the user doing any additional work at his workstation. We conclude that the separation of what a service does from where and how it is invoked is the right approach for transparent execution of computations in a distributed, especially heterogeneous, environment.

Finally, in designing the service execution mechanism we encountered a class of problems that need to manage and locate information that is distributed between many machines in a distributed environment. We present techniques for management of this information and derive evaluation criteria based on problem parameters. These criteria allow for objective evaluation of the best technique for a specific problem. By gathering data from our existing computing environment we were able to calculate parameters for the problems of managing service information and server load information and evaluate the best technique to use for each problem.
BIBLIOGRAPHY


[KW86b] Balachander Krishnamurthy and Craig E. Wills. Omicron: Events \rightarrow Actions. Technical Report CSD TR 594, Department of Computer Science, Purdue University, April 1986.


Craig E. Wills. *The Use of Services in the TILDE Environment*. Technical Report CSD TR 656, Department of Computer Science, Purdue University, December 1986.


VITA

Craig Ellis Wills was born in Wahoo, Nebraska on February 5, 1960. After graduating from Lincoln Northeast High School in Lincoln, Nebraska in 1978, he attended the University of Nebraska, receiving the Bachelor of Science degree (with high distinction) in computer science in May of 1982. Wills continued his education at Purdue University and received the M.S. and Ph.D. degrees in computer science in 1984 and 1988, respectively. While at Purdue, he was supported as a graduate instructor (he was named the department's outstanding graduate instructor for the Spring semester of 1984), as a facilities staff assistant, as a research assistant with the TILDE project, and as a grader.

Also while at Purdue, Wills served as a steering committee member for three years on the local chapter of the Association for Computing Machinery, and is also a member of the national organization.