

AD-A126 003

RESOURCE SHARING IN A NETWORK OF PERSONAL COMPUTERS(U)
CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER
SCIENCE R B DANNENBERG DEC 82 CMU-CS-82-152

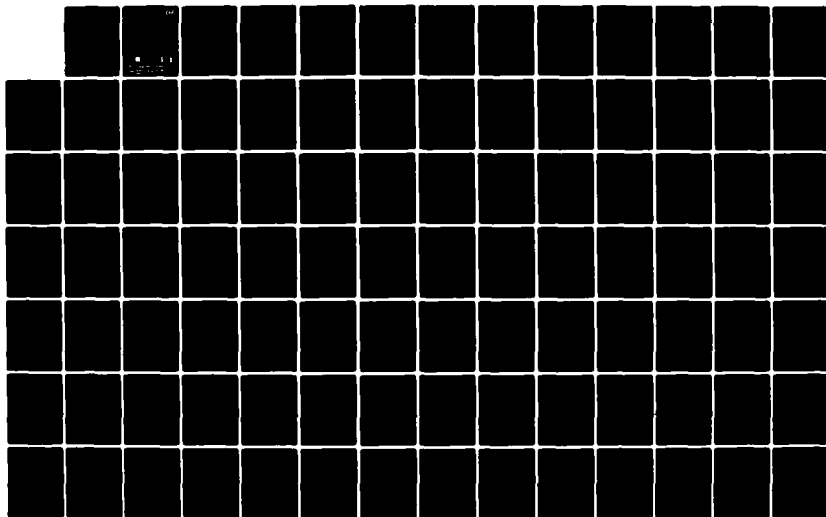
1/2

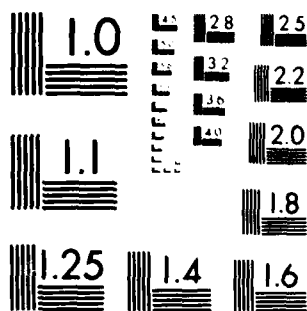
UNCLASSIFIED

F33615-81-K-1539

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

12

**Resource Sharing In A Network
Of Personal Computers**

Roger Berry Dannenberg

December 1982

ADA 126003

**DEPARTMENT
of
COMPUTER SCIENCE**

DTIC FILE COPY



**DTIC
ELECTE
MAR 23 1983**
S D D

Carnegie-Mellon University

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

83 03 03 002

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-82-152	2. GOVT ACCESSION NO. A126003	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) RESOURCE SHARING IN A NETWORK OF PERSONAL COMPUTERS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) ROGER BERRY DANNENBERG		8. CONTRACT OR GRANT NUMBER(s) F33615-81-K-1539
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22204		12. REPORT DATE December 1982
		13. NUMBER OF PAGES 149
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) As networks of personal computers are developed to replace centralized time-shared systems, the need for sharing resources will remain, but the solutions developed for time-sharing will no longer be adequate. In particular, the sharing of network resources is complicated by issues of security and autonomy, since a network of personal computers may be composed of nodes that are completely controlled by their owners.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

2
Hot
#20

To facilitate sharing in this sort of environment, an operating system component called the *Butler* is proposed. As a *host*, the Butler is responsible for administering a sharing policy on its local machine. This includes authenticating sharers, granting rights in accordance with a locally established policy, and creating execution environments for *guests*. As an *agent*, the Butler negotiates with hosts on remote machines to obtain resources requested by a *client*, and performs authentication to discourage a remote host from exploiting the client.

To protect a machine from exploitation by a guest, the host Butler relies upon a capability-based accounting system called the Banker, which keeps track of resource utilization by guests, and provides mechanisms for revoking service. Accounting offers a solution to the problem of *laundered requests*, where a guest performs malicious operations through a privileged intermediary, and the Banker's revocation mechanism is useful in notifying all of a guest's servers that the guest's privileges have been reduced.

Although negotiation is designed to reduce the probability of revocation, a hierarchical recovery scheme is supported by the Butler as an aid to the application programmer in cases where revocation does occur. The three recovery methods are *warning*, where the guest is allowed to perform application-specific actions to free resources, *deportation*, where the guest is transported to another site by Butlers, and *termination*, where the guest is simply aborted.

A number of applications for the Butler are described: these fall into the categories of information exchange, load distribution, and computational parallelism. A prototype Butler has been constructed and used in a real application demonstrating computational parallelism, and the prototype has also demonstrated the deportation of processes.

Abstract

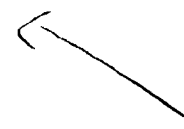
As networks of personal computers are developed to replace centralized time-shared systems, the need for sharing resources will remain, but the solutions developed for time-sharing will no longer be adequate. In particular, the sharing of network resources is complicated by issues of security and autonomy, since a network of personal computers may be composed of nodes that are completely controlled by their owners.

To facilitate sharing in this sort of environment, an operating system component called the *Butler* is proposed. As a *host*, the Butler is responsible for administering a sharing policy on its local machine. This includes authenticating sharers, granting rights in accordance with a locally established policy, and creating execution environments for *guests*. As an *agent*, the Butler negotiates with hosts on remote machines to obtain resources requested by a *client*, and performs authentication to discourage a remote host from exploiting the client.)

To protect a machine from exploitation by a guest, the host Butler relies upon a capability-based accounting system called the *Banker*, which keeps track of resource utilization by guests, and provides mechanisms for revoking service. Accounting offers a solution to the problem of *laundered requests*, where a guest performs malicious operations through a privileged intermediary, and the Banker's revocation mechanism is useful in notifying all of a guest's servers that the guest's privileges have been reduced.

Although negotiation is designed to reduce the probability of revocation, a hierarchical recovery scheme is supported by the Butler as an aid to the application programmer in cases where revocation does occur. The three recovery methods are *warning*, where the guest is allowed to perform application-specific actions to free resources, *deportation*, where the guest is transported to another site by Butlers, and *termination*, where the guest is simply aborted.

A number of applications for the Butler are described: these fall into the categories of information exchange, load distribution, and computational parallelism. A prototype Butler has been constructed and used in a real application demonstrating computational parallelism, and the prototype has also demonstrated the deportation of processes.



Acknowledgements

I would like to thank my advisor, Peter G. Hibbard, and my thesis committee, which consisted of Jim Morris, Richard F. Rashid, and Alfred Z. Spector, for their excellent advice and technical assistance.

The content of this dissertation was strongly motivated by the Spice system, and many of the ideas presented here originated in Spice design meetings. To a large extent, my work has been to integrate the good ideas of other members of the Spice project, who are listed below. Special thanks are due to Richard F. Rashid and George G. Robertson who designed and implemented Accent, the Spice operating system kernel, which formed the basis of my prototype system.

Expressing technical material with clarity can be a difficult job. Cynthia Hibbard's numerous comments on the first draft of this dissertation helped me make the final product much more readable.

I would like to thank everyone in the Computer Science Department at Carnegie-Mellon University for creating such a stimulating and enjoyable environment in which to learn.

And finally, I would like to thank my wife, Frances, for all of her help.

Present and Past Members of the Spice Project

J. Eugene Ball, Mario Barbacci, Bernd Bruegge, Richard Cohn, Marc Donner, Scott E. Fahlman, Robert Fitzgerald, Dario Giuse, Samuel P. Harbison, Greg Harris, Peter G. Hibbard, Paul Hilfinger, Nora Lederle, Horst Mauersberg, David Nason, Douglas Philips, Richard F. Rashid, John Renner, George R. Robertson, Robert Sansom, Samuel Shipman, Alfred Z. Spector, Guy L. Steele, Jr., Mary Thompson, Keith Wright, Barbara Zayas, Ed Zayas

Table of Contents

1. Introduction	1
1.1. Background	1
1.1.1. Personal Computers	2
1.1.2. Resource Sharing	3
1.1.2.1. Sharing for Information Exchange	3
1.1.2.2. Sharing for Load Distribution	3
1.1.2.3. Sharing for Computational Parallelism	4
1.2. Examples	4
1.2.1. Terminology	4
1.2.2. A Personal DataBase	5
1.2.3. Remote Compilation	6
1.2.4. Distributed Program	7
1.2.5. Other Issues	7
1.2.5.1. Protection	7
1.2.5.2. Rights Revocation	8
1.2.5.3. Laundered Requests	9
1.3. Related Work	10
1.4. On this Dissertation	13
1.4.1. Relation to Spice	14
1.4.1.1. Warning	14
1.4.2. Current Status	15
1.5. Summary	15
2. The Butler	17
2.1. Resource-Sharing Issues	17
2.1.1. Protection and Security	17
2.1.2. Autonomy	18
2.1.3. A Digression to Consider an Alternative	19
2.1.4. Practical Considerations	20
2.1.4.1. Support for Sharing	20
2.1.4.2. User Interface	21
2.1.4.3. Policies	21
2.2. The Architecture of the Butler	21
2.2.1. General Design	22
2.2.1.1. Granularity Issues	23
2.2.1.2. Generality	23
2.2.2. Terminology	24
2.2.3. The Butler as Agent	25
2.2.4. The Butler as Host	25

2.3. Negotiation of Resources	26
2.3.1. Revocation of Resources	26
2.3.1.1. Warning	26
2.3.1.2. Deportation	27
2.3.1.3. Termination	27
2.3.2. Resource Specification	27
2.3.2.1. Types of Resources	27
2.3.2.2. Data Types for Resource Specification	28
2.3.3. Negotiation	29
2.3.4. Host Search Strategies	30
2.4. Policy	30
2.5. Protection	31
2.5.1. Types of Threats	31
2.5.1.1. Confiscation	31
2.5.1.2. Sabotage	31
2.5.1.3. Reneging	32
2.5.2. Case Analysis of Threats	32
2.5.2.1. Assumptions	32
2.5.2.2. Protecting the Client	32
2.5.2.3. Guest-Resident Protection	33
2.5.2.4. Protecting the Butler	33
2.5.2.5. Protecting the Guest from the Host	33
2.5.3. Summary of Protection	34
2.6. The Banker	34
2.7. Applications	36
2.7.1. Automated Software Installation	36
2.7.2. Mail Delivery and Bulletin Boards	36
2.7.3. A Distributed, Fault-Tolerant Program	37
2.7.4. Digital Music Synthesis	38
2.7.5. Execution of Engineering Test Programs	39
2.8. Summary	40
3. Security and Protection	41
3.1. Basic Assumptions	41
3.1.1. Protected Address Spaces	42
3.1.2. Protected Microcode	42
3.1.3. Protected IPC	42
3.1.4. Network Encryption	42
3.1.5. Central Trusted Server	43
3.2. Review of Encryption Techniques	44
3.2.1. Conventional Encryption	44
3.2.2. Public-Key Encryption	44
3.2.3. Redundancy for Authentication	45
3.3. Loading An Operating System Securely	45
3.3.1. A Loader Based On Public-Key Encryption	47
3.3.2. A Loader Based On Conventional Encryption	47
3.3.3. A Hybrid Scheme	48
3.3.4. Certifying an Operating System	49
3.3.5. The Local File System	49

Table of Contents

vii

3.3.5.1. Encryption	49
3.3.5.2. Redundancy Checks	50
3.3.5.3. Physical Protection	50
3.4. Secure Intra-Machine Message Passing	50
3.5. Secure Inter-Machine Message Passing	51
3.5.1. Network Servers	52
3.5.2. Passing Port References	53
3.5.3. Secure Network Communication	53
3.5.4. Machine Authentication	54
3.6. Authentication	54
3.6.1. Getting Started	55
3.6.2. Connecting to the CAS	56
3.6.3. Establishing a Connection to a Process	56
3.6.3.1. Authorization	57
3.7. Using Multiple Authentication Servers	57
3.7.1. Loading an Operating System	58
3.7.2. Secure Network Communication	58
3.7.3. Authentication	58
3.8. Resource Sharing Protocols	58
3.9. Summary	59
4. The Banker	61
4.1. Server/Customer/Banker Transactions	62
4.2. Representing Rights	62
4.2.1. RightsValue Type	63
4.2.2. Currency Type	64
4.2.3. Accounts	64
4.2.3.1. Operations On Integer Accounts	64
4.2.3.2. Operations On Set Accounts	65
4.2.3.3. CreateAccount	65
4.2.3.4. Debit	66
4.2.3.5. Credit	66
4.2.3.6. GetBalance	66
4.2.3.7. GetLimit	66
4.2.3.8. GetValue	66
4.2.3.9. SetLimit	67
4.2.4. CurrencyList Type	67
4.2.4.1. CreateCurrencyList	67
4.2.4.2. GrantCurrency	67
4.2.4.3. Iteration Functions	67
4.2.4.4. LessOrEqual	68
4.2.5. AccountList Type	68
4.2.5.1. CreateAccountList	68
4.2.5.2. Deposit	68
4.2.5.3. Withdraw	68
4.2.5.4. SetLimits	69
4.2.5.5. GetLimits	69
4.2.5.6. GetBalance	69
4.3. Basic Banker Operations	69

4.3.1. Defining a Resource	70
4.3.2. Creating an Account	70
4.3.2.1. Initialization	71
4.3.3. Withdrawals	71
4.4. Dependents	72
4.4.1. Withdrawals On Dependent Accounts	74
4.4.2. Access To Account Limits	75
4.5. Server Protocols	77
4.5.1. The Server Interface	77
4.5.2. Server Actions	78
4.5.3. The Overdraft Handler	78
4.5.3.1. Change Limits and Retry	79
4.5.3.2. Service Termination	79
4.5.3.3. Deportation	79
4.5.3.4. Customer Termination	79
4.5.4. Implicit Service Requests	80
4.6. Related Work	80
4.7. Summary	81
5. Negotiation and Revocation	83
5.1. Configuration Specification	84
5.1.1. Server Specification	85
5.1.2. Parameter Passing	85
5.1.3. Resource Specification	86
5.1.4. Environment Specification	86
5.1.5. Server Ports	87
5.1.5.1. Public and Private Ports	88
5.1.5.2. Name-to-Port Translation	88
5.1.6. Representing Configurations	89
5.2. Negotiation Protocols	90
5.2.1. Invoke (Client to Agent)	91
5.2.2. AgentRequest (Agent to Host)	92
5.2.3. AgentResponse (Agent to Host)	93
5.2.4. Additional Operations	94
5.2.4.1. Deportation	95
5.2.4.2. Renegotiation	95
5.2.4.3. Status	96
5.2.5. Alternative Designs	96
5.3. Policy Database	97
5.3.1. Data Structures	97
5.3.2. Butler Access	97
5.3.3. Owner Access	98
5.3.4. Discussion	99
5.4. Warning	99
5.5. Deportation	100
5.5.1. State Location	101
5.5.2. Deport Request	102
5.5.3. State Encoding and Decoding	103
5.5.3.1. A Server Model	104

Table of Contents

ix

5.5.4. Supervision	106
5.5.5. Example	106
5.5.5.1. Model of the Spice Environment Manager	106
5.5.5.2. Implementation	106
5.5.5.3. Deportation	107
5.5.5.4. Importation	107
5.5.6. Discussion	107
5.6. Guest Termination	108
5.7. Summary	108
6. A User Interface	111
6.1. Introduction	111
6.2. Local Program Execution	112
6.2.1. The Environment Manager	112
6.2.1.1. Environment Hierarchy	113
6.2.2. The Terminal Manager	113
6.2.3. Interconnection	114
6.3. Remote Program Execution	115
6.3.1. The Client Server	115
6.3.2. The Forms Interpreter	116
6.3.3. Environment Protection	116
6.3.4. Example	117
6.4. Summary	118
7. Evaluation and Conclusion	119
7.1. Evaluation	119
7.1.1. Support for Sharing	119
7.1.2. Support for Autonomy	120
7.1.3. Efficiency	121
7.1.3.1. Negotiation	122
7.1.3.2. Banking	123
7.1.3.3. Deportation	123
7.1.4. Ease of Use	124
7.1.5. Evaluation Summary	125
7.2. A Prototype Butler	126
7.2.1. Methodology	126
7.2.2. The Application	127
7.2.3. Structure of the Prototype	127
7.2.4. Deportation	129
7.2.5. Invocation Measurements	130
7.2.6. Deportation Measurements	131
7.2.7. Discussion	133
7.2.8. What Was Learned	134
7.3. Conclusions	135
7.4. Future Directions	137
7.5. Contribution to Computer Science	138
Appendix A. Message Specifications	145

List of Figures

Figure 1-1: Terminology for resource sharing.	5
Figure 2-1: Relationship between client, agent, host, guest, and resident.	24
Figure 2-2: Negotiation between an agent and host.	29
Figure 2-3: Using multiple machines to build a fault-tolerant program.	37
Figure 2-4: A distributed music-synthesis program.	39
Figure 3-1: The use of network servers to achieve transparent inter-machine communication.	52
Figure 3-2: Distributing an encryption key.	54
Figure 3-3: Establishing an authenticated secure connection.	57
Figure 4-1: A customer's account and dependent's subaccount.	74
Figure 4-2: The dependent has withdrawn 2.	74
Figure 4-3: Adding another dependent.	75
Figure 5-1: A configuration representation.	89
Figure 5-2: Negotiation messages.	91
Figure 5-3: Abstract state-transition server model.	104
Figure 5-4: Model of a server implementation.	104
Figure 6-1: Port connections associated with an application program.	114
Figure 6-2: Executing a remote application program.	118

List of Tables

Table 7-1: Execution time to invoke and load SigProc using the Butler and Shell.	130
Table 7-2: Elapsed time measurements of the deportation of SigProc.	132

Chapter 1

Introduction

As computer systems evolve away from central, shared facilities to distributed, autonomous, personal machines, the need for communication and cooperation between users will remain. This dissertation addresses the problems of sharing resources in a network of personal computers.

1.1. Background

Within the short history of computers, several modes of computer use have evolved in accordance with rapidly changing technology. Early computers were expensive, and efforts were made to achieve a high utilization of these costly resources. These efforts led first to simple batch systems that process programs and data prepared off-line. Later developments led to multi-programmed systems that achieve still higher utilization of the computer's components by overlapping computation and input-output operations. Experience with multi-programming led to the development of time-sharing, which allows greater interaction between the computer and its users. Time-sharing was effective because large mainframe computers were the most economical machines in terms of instruction executions per dollar, but were far too large and expensive to be dedicated to a single user. These changes in the way computers are used have been accompanied and partially motivated by falling hardware costs. The trend has been to provide computer users with more sophisticated tools and greater computing power in an interactive setting.

1.1.1. Personal Computers

More recently, personal computers have become an attractive alternative to time-sharing. For now, we will define a personal computer as one in which the standard mode of operation is to support a single user. There are currently several factors favoring the use of personal computers. For example, large scale and very large scale integration have made small computers competitive with mainframes, using the metric of instruction executions per dollar. In addition, personal computers can support very high data transfer rates between the processor and input/output devices. This allows more effective interaction through high-resolution display screens, pointing devices, audio input and output, and other forms of man-machine communication. Also, a dedicated computer can provide predictable response times to user requests, unlike a time-shared system where the response time usually depends on the current load or backlog of requests from other users. Another factor is that a collection of personal computers is more reliable than a central machine because a single personal computer failure will affect only one user, whereas a failure in a central time-shared computer will affect all users. Similarly, maintenance of a time-shared computer frequently requires that all service be temporarily halted. Finally, personal computers have some attractive properties from the standpoint of information security and protection. Users can exert more control over their computing environment since they are not sharing their primary resources with others. In particular, users can run various operating systems or write their own microcode without threatening the security of other users.

The capabilities of personal computers are greatly enhanced when they are connected by a data communication network. Computers can then facilitate interpersonal communication through electronic message services. A network also supports cooperation between users on large projects where programs and data are shared, and networks allow a community of users to share expensive resources such as printers and special-purpose high-speed processors and archival memories. Thus, a likely general architecture for the next generation of computer systems is a network of personal computers, with a few shared resources accessed via the network.

1.1.2. Resource Sharing

A network of personal computers has the characteristic that its resources are distributed. In spite of the advantages of distributed resources that are described above, a network of personal computers also has some disadvantages. For example, a user may need to access data that is only available on a remote machine. Security may dictate that the data cannot be transferred in whole to any other machine; thus, the user must use a remote processor to access the data. Another disadvantage is that the physical distribution of resources may not match the distribution of the demands for service. Thus, some resources may be idle while others are overloaded. Finally, even though a personal computer may have significant computational capabilities, its power is less than that expected of a large mainframe computer. As a consequence, a network may collectively have tremendous computing power, but its computing resources are distributed. Programs that might be practical on a time-shared mainframe computer may be inappropriate for personal computers because of the amount of computation involved. All of these problems can be alleviated by resource sharing, as described below.

1.1.2.1. Sharing for Information Exchange

The first problem is related to data security. The standard solution to the problem of controlling data access in a centralized operating system is the use of protected subsystems [Saltzer 75]. However, in a network of personal computers, there is no central trusted operating system, so the way to control data access is to keep the data on the local machine and perform all data operations locally. The owner of the data must then share his machine with users who need to access the data, and in order to protect his data, the machine owner restricts what borrowers can do with his machine. Examples of this type of sharing are appointment-making programs, electronic mail programs, and local databases.

1.1.2.2. Sharing for Load Distribution

The second problem is an imbalance of load on the network of machines, and sharing can be used to distribute the load more evenly. For example, a user may want to run a non-interactive program such as a compiler or text-formatter as a background job while he is using a highly interactive program such as an editor. If there are idle machines available on the network, it may be advantageous to use the remote machine to execute the background job to improve the responsiveness of the editor.

1.1.2.3. Sharing for Computational Parallelism

The third problem is that computing resources are distributed. To obtain more processing power than is available at any given personal computer, some degree of sharing is necessary. The ability to share these resources makes new applications feasible since the computing power available through sharing will always be greater than that available on a single machine. (Note that for this reason, faster machines do not obviate the desirability of sharing.) Examples of applications that can benefit from this type of sharing are computer graphics, image and signal processing, design-rule checkers for computer-aided design systems, and simulation of physical systems.

1.2. Examples

This section introduces some of the problems of resource-sharing in a fairly informal way through the use of examples. The examples correspond to the three uses of sharing enumerated in the previous section, and will serve to motivate the solutions presented in the remainder of this dissertation.

1.2.1. Terminology

Before presenting the examples, it will be helpful to introduce some terminology to name various objects. First, any resource sharing will necessarily involve at least two machines. The machine that belongs to the borrower of resources is called the *local machine*. Any other machines are called *remote machines*. (In this chapter, the term *machine* will be used rather loosely to mean not only hardware, but micro-code and software responsible for executing an application program. For the time being, an operating system is considered part of the machine.) A program that executes on the local machine is referred to as the *user*. In these examples, the user will borrow resources from a remote machine to execute a process. That process is called a *guest* of the remote machine. The owner of the remote machine may also execute processes on his machine. These processes are called *residents*. Figure 1-1 summarizes the relationships between these terms. There is no logical difference between the user and a resident; they represent two views of the same sort of object.

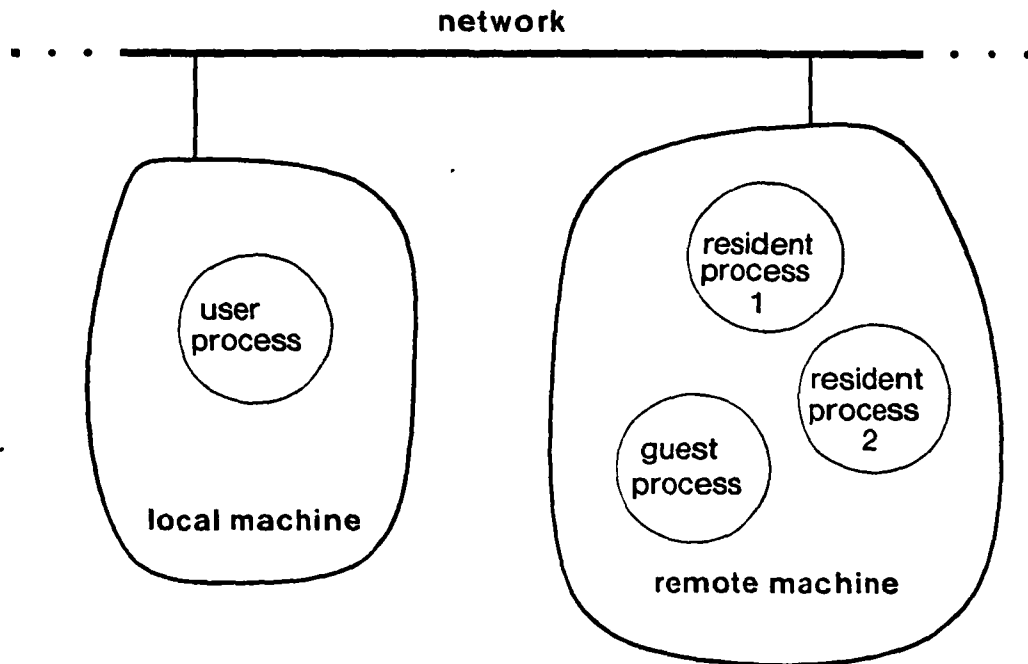


Figure 1-1: Terminology for resource sharing.

1.2.2. A Personal DataBase

In the first example, the owner of a remote machine has decided to implement a small personal database that contains daily appointments. The database is stored only on the remote machine because some of the data is private and the owner does not trust other machines. However, the owner would like to allow limited, controlled access to his database so that his colleagues can make appointments and schedule meetings. To avoid giving users direct access to his database, the owner writes a program that will run on his machine and make appointments for remote users. In effect, the owner has used the physical security of his machine to build a protected subsystem [Saltzer 75].

There are several problems raised by this example. First, there must be some means by which a user can invoke the remote appointment program. Second, the owner may want to restrict access to certain users, so some means of authenticating a user's identity is necessary. Finally, the owner may want to control when users are allowed to use his machine or restrict the priority of some users. This last problem illustrates the concept of autonomy. It

is difficult to make any guarantees to a guest when the guest ultimately has no control over the machine. This problem is discussed further in Section 2.1.2.

Once solutions to these problems are found, the designer must find an appropriate way to implement them. All of these functions could of course be built into the appointments program, but a more general approach is to build an invocation mechanism, authentication protocols, and access controls into a separate piece of software that could then be used by similar applications.

1.2.3. Remote Compilation

Consider the simple case of a user with many compilations to perform. Perhaps the user has just changed the definition of a low-level data-structure that is used throughout a large program, and many modules must be recompiled. It is desired to spread the compilation across several machines. The first problem is to find machines having some idle resources that can be borrowed. The network is used to locate and interrogate remote machines, and some form of negotiation takes place to determine if the remote machine is willing to perform the compilation. There are many issues to be addressed in negotiation:

1. **The Sharing Policy.** The remote machine must know what resources are available to share. Also, the owner of the remote machine may wish to make different resources available to different users, and the amount of available resources may be a function of the current state of the machine. Normally, the owner would want his residents to have priority over guests.
2. **Configuration Specification.** Similarly, the host needs to know what resources are required to execute the guest. In this case, the user only wants resources to perform a compilation. The host is more likely to grant such a request than (for example) a request to load a new operating system. In general, the user needs to specify a *configuration* that names the necessary resources and provides details of the execution environment of the guest.
3. **Negotiation and Authentication.** A protocol for negotiation must exist. The identities of each machine (or machine owner) must be authenticated, and the resource requirements of the guest must be presented and compared with the resources that are available.

Assuming the remote machine agrees to compile programs, a compiler has to be invoked, sources need to be retrieved over the network, and the resulting compiled programs must be returned to the user. Given enough idle resources, many compilations could be performed in parallel on a number of machines.

1.2.4. Distributed Program

In our next example, the user wants to construct a large program that executes in parallel on many machines. One of the problems faced by the designer of such a program is how to handle machine failures and resource revocations. The problem did not exist in the previous example because the user could restart his remote compilation if it failed. In the current example, the user will need to handle many exceptional conditions to avoid restarting his program. Since he is using many machines, exceptional conditions are more likely to arise.

This example raises two problems. First the programmer needs a model of machine failure and resource revocation. To begin with, a machine failure can be modeled as a particular kind of revocation: all use of resources on the failing machine are instantly revoked. (Usually, some timeout period must elapse before the failure can become known.) Less severe forms of revocation can be used in cases where a remote machine owner wants to reduce the resources available to a guest. The second problem is to provide assistance to the programmer in recovering from the revocation of resources. Resource revocation is discussed in Chapter 5.

1.2.5. Other Issues

We have discussed a number of resource-sharing issues in the context of three examples, but several topics remain. For the most part, these are related to the protection of the resource lender.

1.2.5.1. Protection

What happens if the user's source program exhausts the resources of the host's machine? In the case of a compilation, there are some *ad hoc* solutions to such problems. The resource requirements might be guessed from the length of the source code, and the compiler might be designed to abort if its resource requirements exceed certain limits. In this approach, the remote machine would retain control over the situation by using a local, trusted copy of the compiler. On the other hand, it may not be wise to trust a compiler to control carefully its use of resources. Furthermore, the solution is not general enough; it cannot handle situations where the guest's program is not trusted.

Better protection mechanisms are available. Let us assume that nothing is known about

the application program; it would be foolish to turn such a program loose in an unprotected environment, so at the very least, there must be protection equivalent to that provided by time-sharing systems. The remote machine then acts something like a time-sharing system; it creates a process in a separate protected address space for the application program. Depending on the requirements of the application, the host may make a file system or other services available to the application. This can be accomplished in the same manner that time-sharing systems allow protected access to system resources.

In the context of personal computers, a protection problem arises that is not present in time-shared systems: the guest is not secure against the remote machine. In a time-shared environment, the user is not protected from the operating system, but there are usually reasonable grounds for trusting it. In the case of a remote personal computer, the operating system is installed and controlled by an individual who may not be trustworthy. In general, completely protecting the guest from a remote machine is not possible. The options will be discussed in greater detail in Chapter 3.

1.2.5.2. Rights Revocation

In some cases a machine owner may want to change his policies, so that a guest is no longer welcome to borrow resources. For example, the owner may arrive at his machine in the morning to find that a program that was started the previous evening is still running. Clearly, there must be a way of regaining control over one's own machine. The general problem is one of rights revocation. In revoking rights, one can choose actions that favor the guest, the owner, or fall somewhere between these extremes. One could, for example, reboot a machine, but this would abort the guest. The user would be burdened with the task of either restarting his program or recovering from the loss of the guest. Alternatively, the machine owner could be asked to wait while the guest is reconfigured or moved to another machine. This is much cleaner from the borrower's standpoint, but may be objectionable to the remote machine's owner. The problems of resource revocation are discussed further in Section 2.3.1.

1.2.5.3. Laundered Requests

Suppose that the user is malicious and has the goal of controlling or at least crashing the system on the remote machine. Furthermore, assume that the guest is an arbitrary program specified by the user. If the user's application program is executed in a protected address space, there is not much chance that it can directly penetrate the remote system's security; however, the guest may be able to communicate with programs that have greater privileges. For example, the operating system kernel can create new processes, and the file system can access the disk directly. If there are any bugs or design errors in these privileged programs, the guest may be able to use them as intermediaries and take over the system. For example, a bug in an operating system kernel might allow a guest to access memory that would otherwise be protected. Alternatively, the guest may be able to crash the system by overallocating certain resources until no more disk space is available.

This is referred to as the problem of *laundered requests*, because the identity of a request is made to appear "clean" by passing a request through a system program. The problem of laundered requests can be solved by keeping track of the origin of each request for service. When a resource is allocated, it is associated with some guest, and the allocator can test if the guest has exceeded any allocation limits. The problem could be solved on a resource-by-resource basis, but the solution is simpler when there is a more centrally managed source of resource accounting.

In Chapter 4, a server called the *Banker* is described which performs accounting services for real or abstract resources and allows accounts to be held by entities called customers. The purpose of the Banker is to help components of the operating system keep track of the resources that are allocated to a given user. The Banker can be used for fine-grain control of resource use and for controlling privileges which are viewed as abstract resources. It should be noted that the Banker concept is applicable to time-sharing systems as well as to personal computers.

1.3. Related Work

A large amount of work has been done in the area of security. For example, techniques for security in computer systems are surveyed by Saltzer [Saltzer 75], and a survey of encryption techniques is presented by Popek [Popek 79]. Needham and Schroeder [Needham 78] also discuss various protocols for obtaining authenticated communication using both conventional and public-key encryption. One application of these techniques in an operational system is the Liberty Net [Nassi 82] in which naming and authentication services are provided by a single secure server. Liberty Net uses a message-based communication system similar to the one to be described in this dissertation.

Much less work has been done that directly relates to the problems of resource sharing discussed in this dissertation. Svobodova, Liskov, and Clark [Svobodova 79] consider a distributed system composed of autonomous nodes, but focus on primitives for distributed commercial applications rather than on the sharing of resources in the manner I have considered. Further work is reported by Liskov [Liskov 82], who describes an extension to the programming language CLU. Liskov's research is thus concerned with the problem of actually programming a distributed application, while this dissertation addresses problems of obtaining and controlling rights to share resources.

Shoch and Hupp [Shoch 82] describe the "worm" programs implemented on Xerox Alto computers. A worm is a distributed program composed of segments that survive machine crashes and reboots by maintaining copies on other machines. Essentially no protection is provided and there is no mechanism for revocation of resources in a graceful manner. In a similar study, Ball [Ball 81] implemented a fork operation that locates an idle machine and forks a job by copying memory from one Alto computer to another. No further work has been done on either of these projects. [Shoch 81, Ball 81]

While the experiments with Alto computers were oriented toward distributed processing, the issues of sharing data and programs were explored in the National Software Works (NSW) project [Millstein 77, Forsdick 78]. The NSW presents users with a uniform interface to various time-sharing systems on the Arpanet, and is primarily concerned with the user interface. Sharing is motivated not by an integrated environment, but by the incompatibility of software tools written for various time-shared machines.

The NSW is constructed in a modular way, its principal components being *Front End*, *Works Manager*, *Foreman*, and *File Package* processes. On a typical application of the NSW, a user desires access to a software tool, say a compiler, on a remote machine. To compile his program, the user types a command to his *Front End*, which serves as his interface to other NSW components. Since the user's command requires an NSW resource (the compiler), a *Works Manager* process is created. *Works Manager* processes maintain global information about users, the availability of resources, and the state of all current NSW transactions. In this case, the *Works Manager* process will find or create a *Foreman* process to handle the user's compilation. *Foreman* processes serve as interfaces between software tools and other NSW components. If the user's request involves a file access, *File Package* processes are invoked to transfer the file to the proper machine and to transform the file data to the proper format for use by the intended software.

The NSW differs from the system proposed in this dissertation in several ways. The NSW is logically centralized, so resource allocation decisions are made centrally. In addition, the NSW does not allow direct access to the host operating systems that is built upon, so each application must be interfaced individually to the NSW. Finally, the centralized nature of the *Works Manager* requires users to trust and depend upon the correct functioning of a remote machine, and this reduces the autonomy of NSW hosts.

Another system that supports resource sharing is the Customer Information Control System/Virtual Storage facility (CICS/VS), a product of IBM [Eade 77, IBM 79]. CICS/VS was originally developed to simplify the programming interface to terminals in on-line terminal applications, but in later versions of CICS/VS, the notion of terminal was replaced by the more general one of *logical unit*, which can correspond to a terminal, computer, or some other device.

The main function of CICS/VS is to supervise transactions that are originated by logical units. CICS/VS includes support for concurrency in transaction processing, buffer management, the loading and execution of application programs, communication with logical units, and file access. Since one or more logical units may be computers, CICS/VS can support the construction of distributed transaction processing systems, and is therefore similar in intent to the work by Svobodova, Liskov, and Clark mentioned above.

The Resource Sharing Executive (RSEEXEC) [Thomas 73, Cosell 75, Forsdick 78] comes

closer to exploring the issues addressed by this dissertation. RSEEXEC is a network operating system designed to help users of TENEX systems share resources with other TENEX sites on the ARPA network. The general approach of RSEEXEC to the resource-sharing problem is to make network resources available through the existing operating system interface.

An important component of RSEEXEC is its distributed file system, which allows uniform access to files regardless of their locations. The existence of multiple hosts allows RSEEXEC to maintain multiple copies of critical files, a service not available on the single-machine TENEX system. To make file location transparent to user programs, RSEEXEC intercepts user program calls to the local operating system [Thomas 75], and if the request involves a foreign host, RSEEXEC directs the service request to a server process at that host.

RSEEXEC server (RSSER) programs execute on each host supporting RSEEXEC. An RSSER program responds to requests from other machines to perform file access, user status lookup, and other functions. An extensible protocol is used to govern communication between RSEEXEC and RSSER programs.

In RSEEXEC, hosts are autonomous in the sense that a failure in one host will not cause a failure in another; however, hosts cooperate by exchanging status information, and users trust remote hosts not to divulge passwords and other information. This is appropriate for a network of time-shared computers; however, in this dissertation we will consider systems where individuals control their personal machines, leading to the need for even greater autonomy and less trust between hosts. In addition, we will attempt to support a wider class of resource sharing applications by giving users more direct access to network communication facilities and to resources on remote hosts.

Several studies of the scheduling and negotiation aspects of resource sharing have been made. An example of a computer network intended to support resource-sharing among similar machines is the Distributed Computing System (DCS) [Farber 73]. Computers in DCS are not personal computers and are used to support a time-sharing environment, so scheduling becomes an important issue. Resources are located by broadcasting requests to resource-manager processes that then return bids. A request to use the resource is sent to the lowest bidder. Bids are not binding, and a resource manager gives away rights to the first process that asks for them. This may lead to a second round of resource requests and bids.

Another approach to negotiation is Smith's contract net protocol [Smith 80]. This work is

oriented toward distributed artificial intelligence problem-solving programs in which application-dependent knowledge is used to schedule subtasks. One of the goals is to make better global resource-allocation decisions in a distributed fashion. The scheduling and negotiation strategies of DCS and Contract Nets will be compared to our work in Chapter 5.

More recent work on scheduling has been performed by Casey [Casey 81], whose goal is to provide a time-sharing environment using a network of computers. The basic approach is to describe computations using a list of required segment capabilities, and the processing element that is selected to perform the computation is the one that can most easily acquire the necessary capabilities. The selection procedure also takes the processing load of each computing element into account, and several strategies to achieve load-balancing are described and evaluated using simulation studies.

Most of the references above do not address the problems raised by a network of autonomous nodes. However, in the design of the distributed database R^* , autonomy is an important issue [Daniels 82, Lindsay 80]. In R^* , data is partitioned in such a way that permission to access local data is always granted locally; however, the local system trusts remote systems to correctly identify users (presumably for efficiency). Since R^* is a specialized system, it can provide even finer control over access rights than can a more general system such as the one described in this dissertation. On the other hand, R^* is only concerned with shared access to stored information rather than the more general problem of sharing arbitrary resources.

Another approach to data sharing in a network has been designed by Gifford [Gifford 81]. Unlike R^* , in which security depends upon trusted database managers, Gifford's approach depends only upon encryption for protection, allowing shared, insecure machines to be used for data storage. In fairness to R^* , it must be mentioned that Gifford's approach provides a functionality on the level of a file system, while R^* is a powerful relational database system.

1.4. On this Dissertation

This chapter describes the need for resource sharing in a personal computer network and introduces many of the problems involved in sharing. The next chapter describes our approach to the problems of resource sharing, and presents solutions to the problems raised in this chapter, but at a fairly shallow level of detail. For the most part, the remainder of the

dissertation serves to elaborate what is described in the next chapter. In Chapter 3, we will examine low-level security and protection issues, including software certification, secure network communication, and protocols for authentication and authorization. In Chapter 4, we will describe the Banker, an operating system component that performs accounting to restrict or control the resources used by guests. In Chapter 5, we will return to the Butler to discuss protocols for negotiation and mechanisms for revocation. Having examined the programmer's interface to the Butler, we will consider the problems of a user interface to support resource sharing in Chapter 6. Finally, Chapter 7 contains an evaluation of the design presented in Chapters 2 through 6, a report of some experimentation, and my conclusions.

1.4.1. Relation to Spice

The research reported here is motivated by the Spice project at Carnegie-Mellon University [Ball 82], but this dissertation is not specific to Spice except where explicitly noted in the text. In fact, considerable effort has been made to make this work independent of Spice wherever possible.

In some cases, however, it is desirable to apply ideas to a specific system. This allows one to see the effect of design decisions more concretely, and often raises problems that are not apparent at a more abstract level of design. I have used Spice as a target system in which to apply and evaluate parts of the design.

1.4.1.1. Warning

The reader is warned that this dissertation is not a document about Spice. In some cases, components of Spice are described with simplifications or omissions to address the topic at hand more clearly. The Spice project is currently under development, and the design will certainly evolve as knowledge is gained from experience with the system.

In particular, the design for the support of resource sharing as described in this dissertation is intended to be applicable to Spice, but the full implementation has not been constructed, and some changes will be required to accommodate the particular characteristics of Spice. For example, the Spice authentication server is more specialized than the authentication server described in Chapter 3. This will have some impact upon the protocols for authentication and negotiation that are described in Chapter 5.

1.4.2. Current Status

At the present time (December 1982), a stand-alone system for Spice software development is operational on Perq computers. The system includes an operating system kernel, Accent, that supports multiple processes, each with a protected 32-bit virtual address space. Accent also provides a protected message-based interprocess communication (IPC) facility. Other components of the system are a compiler, editor, linker, file system and utilities, process manager, command interpreter, debugger, and a process to manage text and graphics input and output. At this time, shared files must be kept on a central time-shared computer, which is accessed over an Ethernet local network.

Work is now proceeding along several fronts to enhance the system. First, a network server has been implemented that extends the IPC facility across the network, and is in use on an experimental basis. A shared file system for Spice that includes servers for authentication and authorization is now in the early stages of implementation. Also at the implementation stage are several facilities to enhance the user interface, including the environment manager, forms interpreter (these are described in Chapter 6), and an improved command interpreter.

A prototype facility for resource sharing has been demonstrated by the author and is described in Chapter 7; however, a complete facility as described in this dissertation has not yet been implemented.

1.5. Summary

Economic and technological factors combine to make networks of personal computers an attractive alternative to time-shared mainframes. Although the owner of a personal computer has (by definition) his own computer, there are several reasons for sharing computer resources. Computers may be shared to enforce the protection of data, to take advantage of idle resources, and to achieve higher rates of computation through parallelism.

A number of requirements associated with resource-sharing have been identified:

1. **Invocation.** There must be some means of starting a program or operation on a remote machine.
2. **Access Control.** The invocation mechanism must grant access to resources selectively.

3. **Authentication.** For security purposes, the identities of users must not be forgeable when transmitted to a remote machine.
4. **Autonomy.** The owner of a machine must have control over that machine. In particular, this means that the owner can control the sharing of his resources.
5. **Configuration specification.** Users must be able to specify the resource requirements of guests.
6. **Negotiation.** A protocol must exist whereby the user can negotiate with the remote machine to borrow resources.
7. **Remote machine protection.** The remote machine must prevent the guest from taking control.
8. **Guest protection.** The guest should be secure against attacks by a malicious owner at a remote machine.
9. **Laundered requests.** Guests must not be allowed to use privileged programs as intermediaries to acquire unauthorized resources or access to information.
10. **Resource revocation.** Users and guests must handle machine failures and revocation of resources.

The remainder of this dissertation explores ways of satisfying all of those requirements within a single integrated system. We will begin in Chapter 2 by introducing an architecture for resource sharing referred to as the Butler paradigm.

Chapter 2

The Butler

The Butler is an operating system component that facilitates resource sharing. In this chapter, we will reexamine some of the resource-sharing issues described in Chapter 1 and see how they affect the Butler design. The chapter concludes with a presentation of several Butler applications.

2.1. Resource-Sharing Issues

Before the problems inherent in resource sharing can be fully appreciated, some more characteristics of personal computer networks must be examined. These characteristics differ sharply in certain areas from those of time-sharing systems, and because of these differences, new solutions to the sharing problem are necessary.

2.1.1. Protection and Security

Most mainframe computer systems are physically secure (or at least they are assumed to be). Only authorized, trusted personnel are allowed access to the physical machine, and the typical user can only access information through an operating system interface that protects information from unauthorized access. Operating systems typically allow many users to share such a machine while protecting each user's private data, and the current state of the art [Wulf 74, Organick 72] allows users to be highly selective in their ability to grant authority to other users. However, these techniques rely on the machine being physically secure: if it is not then the software security can be compromised. This could be done by halting the machine and examining the contents of memory, for example.

On the other hand, personal computers might be located at the point of use, in an office or home. If the owner is the only one who has access to the machine, then it might be

considered secure against everyone except the owner. In addition, if the owner is the only user, then there is no sharing and no special software protection is required. From this point of view, personal computers have advantages over time-shared computers in the area of protection. The user does not even need to trust the operating system entirely, since the network is the only way information can be transmitted. In addition, the user controls physical access to his machine, rather than trusting the management and operating system of a time-sharing system.

If, however, personal computers are shared by several users, then protection and security become issues. One goal of this dissertation is to explore these problems; it will be seen that different assumptions about the physical security of machines will lead to varying levels of protection.

2.1.2. Autonomy

Another important characteristic of personal computers is that users are generally given almost complete control over their machine. This characteristic, called *autonomy*, has two advantages for the user. The user can form *stable expectations* of the available computing power, since he can regulate the load on his machine. Users of time-shared machines have not tended to form stable expectations of computing resources since the level of service in most systems depends upon the number of users and the tasks they are performing. The second advantage of autonomy is the ability to run whatever programs, operating systems, or microcode the user desires. This is possible when there are no sharers whose security might be threatened.

The notion of autonomy follows almost directly from the protection characteristics discussed above. If a computer is physically unprotected from a user, then there is little that can be done to prevent the user from gaining control of the machine. Given this state of affairs, it seems reasonable to recognize autonomy of users as an inherent property and seek to exploit that property wherever possible.

The property of autonomy and the desirability of stable expectations might at first seem to be incompatible with resource sharing. There are several reasons to believe that this is not the case. First, in a network of personal computers, one can expect many machines to be idle. When a machine is idle, the concept of stable expectations is not meaningful, and there

is no reason (aside from protection issues) that an owner would not want to share his idle resources. Second, users can cooperate by sharing resources. An example has already been given in which sharing is necessary to implement a protected subsystem. Another example of cooperating users is a project that requires the computational resources of several machines operating in parallel. In an institutional setting, some degree of sharing might be legislated to improve productivity.

Thus, it can be seen that a problem that must be addressed is how to control and regulate sharing. In time-sharing systems, common goals are to maximize throughput, provide quick response, or provide a "fair" allocation of limited resources. In a network of personal computers, the goal of autonomy dictates that each user must be able to decide to what extent his machine is shared. For human engineering reasons, the user should be able to create *policies* that constrain sharing. These policies must then be administered by some component of the operating system.

2.1.3. A Digression to Consider an Alternative

At this point, the reader might wonder what would happen if some of these assumptions are incorrect, or become invalid due to some technological advances. One could conceive of a personal computer system in which machines were physically secure from everyone including their owners. To achieve this, the machines could be locked inside secure enclosures or moved into a secure room apart from the users' consoles. Alternatively, one might be willing to assume that owners are either honest or lack the technical skills required to access information that is "protected" by software. In either case, protection problems are not as difficult to solve, and the autonomy assumption might not be so necessary, since the protection of machines from users allows the system designer to determine the degree of control each user has over his machine.

A system with protected machines is in many ways like a time-sharing system, since access restrictions can be placed even upon machine owners. The difference between this hypothetical personal computer system and a time-sharing system is that in the personal computer system, users may be given greater control over their machines than would be appropriate if they were using a time-shared mainframe computer. (A network of computers might also have different performance characteristics than those of a central computer system.) The range of autonomy can extend from none, which is equivalent to a distributed

time-sharing system, to complete, which means that users can determine how all of their machine resources are used. The Eden system [Almes 80] and the Cambridge Ring [Wilkes 79] are examples of networks that fall into this range between fully autonomous personal machines and a completely shared central facility.

In this dissertation, this sort of system is not specifically considered for the following reasons. First, it is felt that protection is a real problem that should receive serious consideration. Fortunately, many aspects of the sharing problem are independent of protection so the consideration of protection issues will not lead us down a completely different path of research. Second, the cases where autonomy is eliminated have been studied extensively under the topic of time-shared systems. The implementation of a time-sharing system on a network of personal computers is perhaps interesting and certainly raises some new problems, but they will not be addressed in this dissertation. Finally, as argued above, the protection characteristics of most personal computer systems dictates that machines be considered autonomous. In summary, this dissertation is concerned with resource sharing in a network of personal computers, not in distributed time-sharing systems.

2.1.4. Practical Considerations

The principal problems of resource sharing in the context of personal computers are protection and autonomy, however other problems must be addressed before sharing becomes practical. First, there must be software support for sharing, which might be regarded as an extension to an operating system, through which programs can obtain resources on other machines. Second, a user interface must be constructed to support sharing. Users should be able to control easily programs executing on several machines. Finally, a user must be able to change the set of policies that determines how his machine is to be shared.

2.1.4.1. Support for Sharing

Sharing can be supported in at least two ways. First, conventions and protocols which are obeyed by all machines on the network can simplify the task of borrowing resources. Second, most of the details of borrowing resources and executing programs remotely can be hidden from the programmer. High-level operations should be provided to request and use resources.

2.1.4.2. User Interface

A common form of resource sharing is likely to be of the type illustrated in Section 1.2.3, that is, the remote execution of application programs. The added complexity of running a program remotely must provide only a small disadvantage compared to the performance advantages; otherwise, users will always execute their programs locally. In addition, it should be possible to execute most programs remotely with no changes to the code, and remote programs that interact with the user should use the user's local display and keyboard.

2.1.4.3. Policies

The problem of controlling one's personal computer has already been mentioned. Any facility to support resource sharing should provide the *mechanisms* for sharing without dictating how these mechanisms are used. For example, a machine owner should be able to decide not to share his machine at certain times. Control is provided through *policies* which dictate how sharing may take place. By decoupling policies from mechanisms, the behavior of the system can be modified simply by rewriting policies, which is expected to be much simpler than altering the sharing mechanisms.

It should be simple for users (not just the implementors) to change policies. Two approaches might be taken to help users implement policies. One approach is to represent policies as programs (procedural knowledge) in some appropriate language. In this case, a policy interpreter would be responsible for executing the programs. Another approach is to represent policies as data (declarative knowledge) which is consulted whenever a policy-related decision must be made. This latter technique is advocated because it is simple to implement and easy to understand, although it is not as powerful in some cases. Policies are discussed further below.

2.2. The Architecture of the Butler

Our approach to resource sharing is based on the concept of the Butler¹, an operating system component that helps users to borrow resources and supervises guests. A Butler program runs independently in each personal computer.

¹The Butler is so named because it manages a personal machine on behalf of the owner in a way that is loosely analogous to the way a (human) Butler manages his employer's household.

Although in principle the Butler's functions could be entirely implemented within application software, a number of reasons justify the existence of the Butler as a separate software entity:

1. The Butler provides a single point from which users can control their system. This permits the enforcement of policies that relate to the global state of the machine.
2. One of the functions of the Butler is to enforce security. It protects the machine by supervising potentially malicious software of other users. The Butler should be a trusted piece of software that exists as a separate entity from the applications it supervises.
3. Furthermore, if the Butler is distinct from the applications it supports, then the Butler will be used in a variety of circumstances, and users will be able to gain confidence in its security. If each application reimplemented software that is critical for protection, then errors would be more likely, and users would be less likely to share their machines.
4. The Butler is more than a set of conventions or subroutines. It serves as a general model for the top-level structures of distributed programs. The Butler paradigm is thus a conceptual tool for the programmer.

Because of the principle of autonomy, a separate instance of the Butler is executed by each computer. Butlers are better thought of as independent, but cooperating programs, rather than as a single distributed program.

Since users are free to execute any program, they can implement their own Butlers if desired. Given the difficulty of a Butler implementation, it is unlikely that more than one Butler will exist.

2.2.1. General Design

The purpose of the Butler is to allow clients to invoke operations on remote machines. In addition to simply providing the capability of running a program remotely, the Butler allows a client to invoke arbitrary services, such as the program for making appointments in Chapter 1. In this way, the Butler supports the sharing of information as well as resources. Other applications are discussed at the end of this chapter.

2.2.1.1. Granularity Issues

The Butler will provide the user with a high level of support that addresses all of the problems raised in the examples given earlier. The Butler is therefore a rather "heavyweight" mechanism. To avoid a large overhead, it is intended that the Butler be used to initiate operations and then intervene only when necessary to handle exceptional conditions. There remains a certain amount of overhead for protection, but this overhead will be present in any design.

Even the overhead to initiate an operation may be too great if a client wants to invoke many small operations remotely. For example, consider the task of performing many small operations on a remote database. Although the Butler mechanisms are too expensive to invoke on every remote operation, the Butler can be used to establish a connection to a server that handles the small operations without further intervention by the Butler. The initial cost is then amortized over many transactions, which may be based upon efficient communication primitives such as remote procedure call [Nelson 81] or remote references [Spector 82].

From the discussion above, it should be clear that the Butler is not an interprocess communication facility, nor is it a form of remote procedure call. The Butler is a higher level program that must be constructed on top of lower level communication or invocation primitives.

2.2.1.2. Generality

In this dissertation, the design of the Butler is deliberately as general as possible. The goal of the Butler design is to support many forms of sharing and to address many of the problems. Once practical experience is obtained, some features of the Butler might be deemed unnecessary, perhaps leading to a simplified design.

As in Chapter 1, some new terminology is necessary to refer to various participants of resource sharing. The terminology presented below is an extension of that used earlier.

2.2.2. Terminology

At least two machines, the *local machine* and the *remote machine* are involved in sharing (see Figure 2-1). On the local machine, the program that needs to borrow resources, the *client*, relies on the local Butler to perform the borrowing. In this role, the Butler is called an *agent*. The agent communicates with the Butler on the remote machine which acts as a *host*. The host creates one or more processes, called *guests*, on behalf of the client. Any program that is sharing the remote machine (including another guest) is called a *resident*.

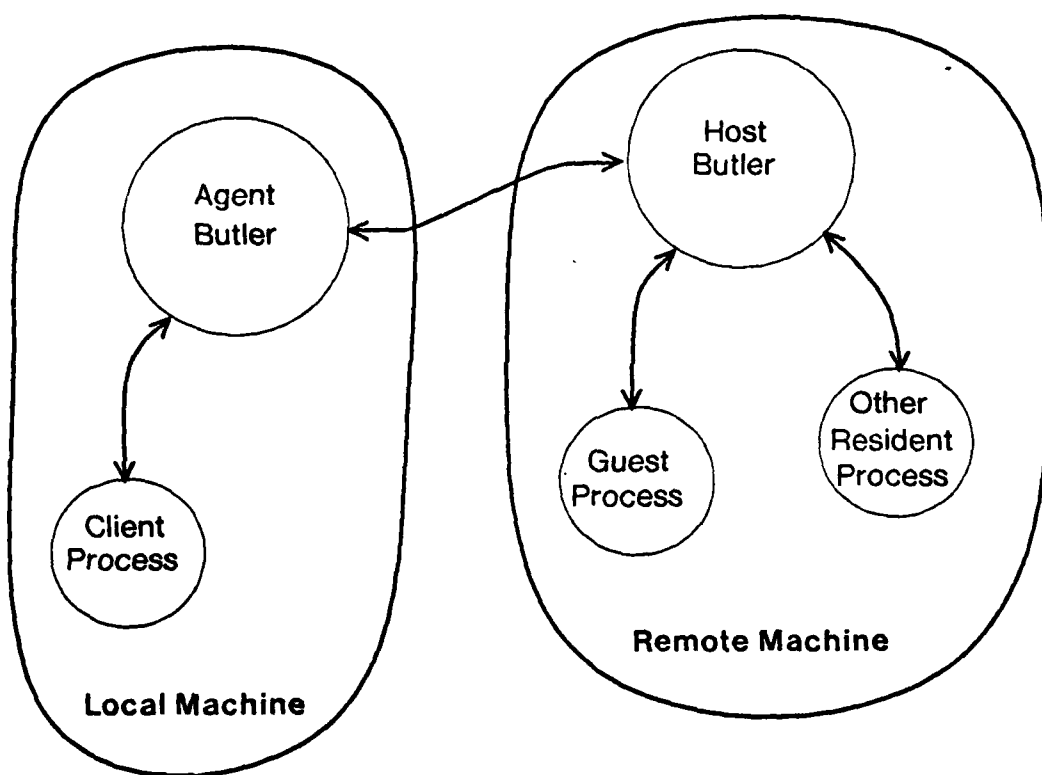


Figure 2-1: Relationship between client, agent, host, guest, and resident.

2.2.3. The Butler as Agent

The job of the agent is to locate a host with the required resources, negotiate with the host, and invoke a service requested by the client. To borrow resources from a remote machine, the client presents his agent with a request for some service. The service request specifies the configuration required by the client. For example, the request for a compilation would contain the name of the compiler, a list of machine resources, information for exceptional condition handling, and perhaps a request to use the remote machine's file system.

The agent locates a suitable host and negotiates with it to obtain the required resources. If the resources are not available, the agent looks for another host. When a suitable host is found, the identity of the host is authenticated. (Authentication could also be performed before the negotiation takes place, but this would make the search for a host slower. Only the chosen host's identity needs to be verified.) The host then invokes the requested operation. At this point, the agent's job is done unless exceptional conditions arise. The primary exception that the agent handles is resource revocation. In some cases, the agent may be able to locate a new host and *deport* the guest in a way that is transparent to the client and guest.

2.2.4. The Butler as Host

The job of the host is to loan resources while protecting the interests of the machine owner. When a request arrives from an agent, the host consults a policy database to determine whether the resource request can be granted. If so, the host creates an appropriate execution environment for the guest. In general, this means creating a new process and supplying the guest with capabilities to access other components of the system, as specified in the client's original request. Normally, among these capabilities are network connections to the client so that the guest and client can communicate directly.

After creating the guest's execution environment, the host stands by in case the guest attempts to exceed the limits placed on its resource utilization. This will ordinarily be detected by some component of the operating system. For example, the kernel will detect attempts to address memory outside permissible limits or to fork too many processes, and the file system will detect when disk page limits are exceeded. In any case, the host handles the exception. The host's action can take one of several forms depending on what action was requested during initial negotiations.

The host also responds to changes in the policy database. A new policy may reduce a guest's resource rights, so some resources may have to be recovered from the guest. Revocation of rights is handled just as if the guest tried to exceed its rights.

2.3. Negotiation of Resources

Negotiation is the process of establishing the client's intentions and assuring that they will be met. This allows us to formulate rules governing the behavior of programs, thus making program behavior more predictable and easier to make correct. For example, negotiation can assure a programmer that either his resource demands will be met, or a warning message will be sent to a specified port. Negotiation also increases the likelihood that a guest will be able to use resources effectively, improving the overall system performance.

2.3.1. Revocation of Resources

An important part of negotiation is the determination of how resources will be revoked if that becomes necessary. Three methods are used to handle resource revocation. The first, called *warning*, gives the guest a chance to perform application-specific recovery actions. The second is *deportation* which is handled entirely by the host Butler. If all else fails, *termination* is used to recover all resources used by the guest.

2.3.1.1. Warning

The first method augments the guest's current resources by a set of *warning resources* and notifies the guest of the change. For example, the guest may receive 5 additional seconds of CPU time, and 10 additional disk pages along with a warning message. The 10 disk pages would be added to the guest's current allocation to establish the new quota, regardless of the initial one. The purpose of this revocation style is to give the guest the greatest amount of flexibility in recovering from a loss of resources. The warning resources are finite because the host cannot trust the guest to observe the warning.

2.3.1.2. Deportation

The second method is called *deportation* and will be discussed in detail in Chapter 5. The goal of deportation is to provide a mechanism whereby resources can be reclaimed by the host without harming the guest, but without giving the guest any control. Deportation removes all processes created by the guest, as well as the environment created for the guest. Since deportation is transparent to most guests, it is not necessary to add special recovery software to most applications.

2.3.1.3. Termination

The third method aborts the guest process and sends an explanation to the agent. This method is invoked if all else fails or if higher level revocation-handling mechanisms are not requested.

Resources are also revoked if the host machine crashes. In this case no notice can be sent by the host, but the agent is informed of the crash by the network server after a timeout period.

2.3.2. Resource Specification

Before discussing negotiation, a more concrete idea of what is meant by the term "resources" is required. A representation for a resource is also provided so that a client may describe resource requirements to his agent, which uses the requirement specification to find a suitable host.

2.3.2.1. Types of Resources

There are many resources in which the client may be interested. Most of them are abstractions of the physical machine, such as disk pages, processes, or a share of the CPU, but other resources relate to services, such as access to the local file system. Resources can also refer to rights or expected behavior; for example, the right of a guest to receive a warning message before the host revokes any resources is considered to be a resource.

2.3.2.2. Data Types for Resource Specification

Several data types are designed for representing resources. First, all resources not related to negotiation are grouped into a *BasicRights* data type. *BasicRights* is a collection of values representing either numerical limits (how many of resource X) or boolean decisions² (the right to perform operation X). There are several operations besides the normal access functions to read and write fields of *BasicRights*. Since resources are scalar quantities, addition, comparison, difference, maximum, and minimum operations are possible.

GuestRights is a data-type that fully specifies a guest's rights, including revocation. One field of *GuestRights* is a value of type *BasicRights*, and two additional fields specify if warning or deportation is to be performed. If both rights exist, warning will be attempted first, and deportation is only used if the warning is not heeded by the guest; that is, the guest attempts to exceed even the warning rights. In the case that a warning is specified, an additional set of *BasicRights* must also be included to specify the increment of resources required to handle the warning.

Possible Ada type-specifications for the data structures discussed thus far are given below; however, the simple representation suggested here for type *BasicRights* cannot be used to represent dynamically created resources. The representation below is presented for illustrative purposes only, and is just one of many possible strategies. We will treat the topic of resource specification in greater depth in Section 4.2.

```
type BasicRights is
  record
    DiskPages:      Integer;
    NumProcesses:   Integer;
    Priority:        Integer;
    MicrocodeAccess: Boolean;
    .
    .
    .
  end record;
```

²I do not mean to imply that all boolean decisions must be implemented with bits. For example, the representation of the right to use a server might be represented by a string, etc.

```

type GuestRights(Warning: Boolean) is
  record
    InitialRights: BasicRights;
    Deport: Boolean;
    case Warning is
      when true = >
        WarningRights: BasicRights;
      when false = > null;
    end case;
  end record;

```

2.3.3. Negotiation

The client expresses his request to the agent in the form of two values of type *GuestRights* and a specification of the operation to be performed. The first *GuestRights* value expresses what resources the client wants. The agent searches for a suitable host; if the agent finds a host offering these resources, the agent need not look any further. If the agent has difficulty meeting the first resource specification, a client may be willing to accept fewer resources. The second value of type *GuestRights* expresses the minimum acceptable amount of resources.

The negotiation process is outlined in Figure 2-2. The agent sends the client's *preferred rights* to a potential host. The host replies with a list of *available rights* and reserves those resources until receiving a response (subject to timeout). The agent compares the host's response to the client's request and either accepts or rejects the offer. Authentication is not used on initial negotiations during which many potential hosts may be polled.

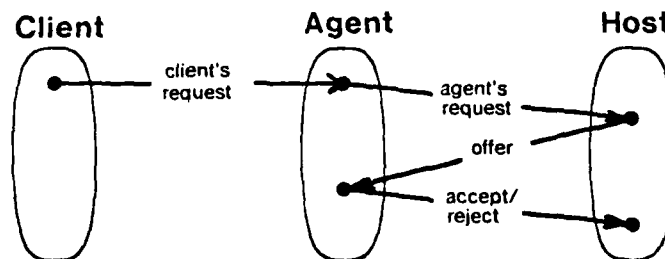


Figure 2-2: Negotiation between an agent and host.

2.3.4. Host Search Strategies

The client has the ability to pass a *host search list* to the agent to specify which hosts to ask for resources. In the absence of a search list, the Butler will find potential hosts through a network name server. The client may also request deportation at any time, so that if a more suitable host is found, a guest may be moved.

I do not propose the use of sophisticated search strategies or attempts to automate load balancing. These are functions appropriate to a distributed operating system, and are beyond the scope of the Butler. The possibility of implementing distributed operating systems at the level above Butlers is not to be ruled out, however. For example, it may be desirable for groups to optimize resource usage of the machines under their jurisdiction. The Butler design allows for this, but does not directly support it.

2.4. Policy

If the host Butler is to negotiate with agents, it must know what resource rights to offer. In this section, we will see how policies are used to control negotiation.

The Butler attaches two properties to potential users of a machine. The first property, called *locality*, is *local* if the user is physically present at the machine site, and *remote* if he is not. This distinction is useful because local users expect to use I/O devices such as the keyboard, screen, and pointing device. The second property is *occupancy*, which is true if the user has rights to the entire machine, and false for users who are borrowing resources. Typically, occupancy is true only for the owner of the machine.

The Butler's interface to the policy database is a function that takes a user's name and properties of locality and occupancy, and returns a set of rights:

$$\text{Policy: UserId} \times \text{Locality} \times \text{Occupancy} \rightarrow \text{Rights}$$

The rights may also be a function of the current machine state, the time of day, and so on. An occupant may also dictate one of two *modes*. *Sharing* mode allows the local Butler to host one or more guests. *Exclusive* mode prevents guests from using the machine. Thus, sharing can be temporarily denied without changing the policy database.

Because access to the database is made through an abstract interface, its implementation is independent of the Butler's. The policy database can evolve without reimplementing the

mechanisms provided by the Butler. The user's interface to the database is unspecified to allow experimentation with different human interfaces.

In addition to the policy function, the interface between the Butler and the policy database must include some way to notify the Butler when policy is changed. Upon receiving a change notice, the Butler reevaluates the policy function for each guest on the machine. This avoids the necessity of continuously reevaluating (polling) the function to keep policy enforcement current.

2.5. Protection

This section presents ways in which unprotected programs can be attacked. The possible sources of security threats are then surveyed and the methods of providing security are described.

2.5.1. Types of Threats

An insecure system exposes users and their programs to a variety of attacks. A program is called *malicious* if it uses unintended rights to access or manipulate another program or its data. The act of using these unintended rights is called *exploitation*, of which there are several forms.

2.5.1.1. Confiscation

First, a malicious program may read data or code belonging to another program, possibly resulting in access to secret information. This is called *confiscation*, which may also be used to acquire rights (capabilities) of the victim, leading to further exploitation.

2.5.1.2. Sabotage

A second form of exploitation is *sabotage*, in which information is manipulated by a malicious program. Random manipulation of data can result in errors that are hard to detect and may result in a failure of the sabotaged program. Sabotage might also be used to control the victim's actions. For example, if a client makes decisions based on information received from a guest, then manipulation of the information might be used to influence the behavior of the client.

2.5.1.3. Reneging

A malicious program can exploit an agent Butler and its client by offering services or resources and then not honoring the offer. If the client is fragile, it may not be able to recover from the revocation. At the very least, the client will experience delays due to recovery and acquisition of new resources. This form of exploitation is called *reneging*.

2.5.2. Case Analysis of Threats

Before discussing protection methods, sources of potential protection violations must be identified. The framework within which this design is placed identifies five principal domains of interest: the client, the agent, the host, the guest, and the resident. The interesting interactions will be discussed below.

2.5.2.1. Assumptions

It is assumed that a user can load a secure operating system that allows multiple processes to run in separate virtual address spaces. It is also assumed that the user has the capability of verifying at load time that his operating system is in fact an authentic one, and finally, we assume that machines can communicate securely over encrypted channels. These assumptions are justified here because we are interested in the security problems raised by Butlers and resource sharing, and because they are likely to be met by any network of personal computers where security is important. In Chapter 3, we will see how these assumptions can be met using encryption techniques and a trusted authority.

2.5.2.2. Protecting the Client

If we assume that the client has a benevolent agent, that is, the machine has been loaded with a secure operating system, then the client's security can only be threatened through the guests which may interact with the client. If the guests are safe, then the client is also safe (protection of the guest is discussed below). The client can protect itself against an unsafe guest by limiting the rights granted to a guest, which can be accomplished by restricting the environment in which the guest executes. The use of message-passing and separate address spaces rather than shared objects for interprocess communication helps the client to maintain firewalls against corrupted guests. An extremely suspicious client could supply the guest with no rights except a communication path to the client. The client could then perform (or refuse to perform) sensitive operations after checking to see if the requested operations are permissible.

2.5.2.3. Guest-Resident Protection

Guests and residents must be protected from each other, just as users of a time-shared system must be mutually protected. The host prevents interaction between the guest and resident through the standard use of separate protected address spaces. Furthermore, the host Butler prevents either the guest or resident from monopolizing physical resources by enforcing the machine owner's policies. The use of laundered requests has already been described as a potential problem. This problem will be dealt with below in Section 2.6.

2.5.2.4. Protecting the Butler

The host protects itself from guests using the same mechanisms that are used to protect residents. The only other threats to the security of a Butler come from other machines via messages, since the local system is secure by assumption. Hence, the Butler must be suspicious of all messages it receives. Since Butlers are autonomous, there are no global states to be protected, and the Butler can treat all incoming messages as suggestions, acting upon them only when the suggestions are consistent with local (trustworthy) data.

2.5.2.5. Protecting the Guest from the Host

Spice machines can be arbitrarily programmed by users, so it is impossible to provide absolute protection for the guest. A malicious user can construct and execute a program that mimics the Butler interface, but provides no protection for guests. Below, a scheme is described that can be used to discourage such behavior. A few stronger schemes, which require stronger assumptions, are then presented.

Authentication can play an important role in discouraging malicious behavior. If illegal conduct can always be traced to the person who is responsible, few people are likely to behave maliciously. A machine owner who allows a guest to borrow resources is responsible for executing a certified copy of the operating system on his machine. If a violation of this rule is detected, authentication allows the responsible user to be identified.

There are two authentication protocols used to support the Butler. The first is used to authenticate a machine owner to a trusted, physically secure *Central Authentication Server*, or CAS. In this protocol, the owner's password is sent over an encrypted network connection to the CAS. The CAS then associates the owner's identity with that of the connection to the CAS, so that further messages to the CAS do not need explicit authentication information.

The second protocol is used to set up a secure and authenticated communication channel between two Butlers. (Each Butler assumes the identity of the machine owner who creates it.) These protocols are described in Chapter 3.

If stronger assumptions are made, it may be possible to provide greater protection for guests. For example, if we assume that machines cannot be microcoded by users, it might be possible to provide remote certification that a particular operating system is loaded; however, it is necessary to assume users do not tamper with hardware, or that parts of the machine are physically secure. An extreme case is the use of tamper-resistant hardware modules [Kent 80]. All of these schemes rely on physical protection in one form or another.

2.5.3. Summary of Protection

An important job of the Butler is to provide protection for resource sharers. Although a secure operating system can implement processes with separate protected address spaces, the operating system must be extended with the Butler to deal with protection problems that involve multiple machines. One such problem is the protection of the guest. Authentication is used to discourage malicious behavior that compromises a guest's security, but stronger techniques are possible if physical protection can be guaranteed. Another important problem is the protection of residents from the guest. The most important task here is keeping track of resources given to the guest to prevent the guest from exceeding its resource limits. Protection problems will be dealt with in greater detail in Chapter 3.

2.6. The Banker

The Banker is used for the protection and tracking of resources, and it represents a solution to the problem of laundered requests in which a guest coerces a server with greater privilege to behave maliciously. The problem of laundered requests also appears when we wish to revoke rights from a guest. Simply migrating the guest process may not recover many resources if the guest has employed local servers. Consider the following: a host wants to recover all of a guest's resources, so it halts and destroys the processes in use by the guest. The guest, however, has previously transferred local file system connections to the client. The client can therefore continue using resources on the remote machine. The problem here is that the host has lost track of the fact that access to the file system and its resources were granted to the guest.

To solve this problem, a new server called the *Banker* is created to manage accounts for all users. The Banker maintains an account list for each guest, and an unforgeable *signature* is given to the created guest process to use in all transactions with servers. The purpose of the account list is to specify a set of available resources. Whenever a server allocates or deallocates an accounted resource on behalf of some process, the account indicated by the process is debited or credited by the server. The Banker informs the server when a debit would overdraw the account.

The Banker makes available several types of accounts, corresponding to different resource accounting methods. Typically, accounts will have operations that are isomorphic to transactions on real-world bank accounts. An example of an unorthodox account type is one for priority levels in which accounts cannot be combined additively. Another example of accounts implemented by the Banker are subaccounts which can draw upon a master account. This type of account can be used, for example, to control resources when a guest process forks.

The Banker is useful for a number of reasons:

1. The Banker provides accounting services for other servers. This simplifies the servers, and allows them to share common operations. In addition, it provides an *identification* service in that it maps signatures to accounts. A user does not necessarily need to authenticate himself to each server, since his signature serves as a capability.
2. Identities maintained by the Banker are abstract. The Banker does not associate resources with any specific object such as a process, (human) user, or console as is frequently done in current systems. Thus, the association of resources to objects can be flexibly determined.
3. The Banker has many of the advantages of a capability-based protection system. Signatures are analogous to capabilities, but the operations on accounts are an extension to the normal operations provided on capabilities. Typically, capabilities carry a small set of boolean values indicating rights. Signatures carry account lists which can be large sets of values, and are not necessarily boolean. The capability-like aspect of signatures allows users to pass subsets of their rights on to subsystems by creating subaccounts. Users can provide their own exception-handlers to be invoked if a subsystem tries to overdraw an account. Policy is therefore determined entirely outside the Banker, which simply provides mechanisms for accounting.
4. The Banker contains all of the data structures necessary to map identities to resources. Thus, it is possible, given a signature, to find all of the resources that have been allocated from that account. This is useful for recovering resources from a guest.

2.7. Applications

The examples in Chapter 1 present three uses of sharing that are good applications for the Butler. A few more examples are considered below.

2.7.1. Automated Software Installation

A potential problem with a network of personal computers is the maintenance of up-to-date system software on local disks. Most users will not want to perform file updates manually as is commonly done in current systems. To avoid this work, users can let a program do this automatically. The policy database is altered to enable a remote program to update the local disk. Ordinarily, only the system administrator would be authorized to perform these updates.

When a new version of a system program is issued, the system administrator runs an update program which has a list of machines requesting automatic file updates. For each of these machines, the administrator's program uses the Butler to access the remote machine's file system. The new version is then copied to the remote machine.

2.7.2. Mail Delivery and Bulletin Boards

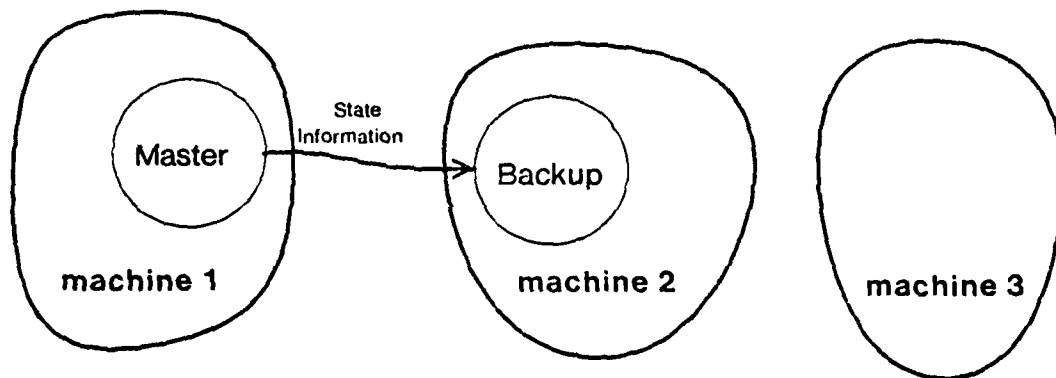
The Butler can be used to help transfer electronic messages. For example, suppose a user wants to transmit a message to some remote machine. The user invokes a mailer program and gives it his message. To deliver the message, the mailer program, poses as a client and requests its agent to invoke a mail server on the remote machine. The agent will contact the host of the destination machine and attempt to invoke the mail server. If the user is authorized, the mail server is invoked, and the message is delivered.

Computer mail systems usually require authentication of the sender and receiver. Since the Butler performs authentication as part of negotiation, the mail system is simplified. Another advantage of the Butler is that it can invoke servers only when they are necessary. Thus, a user need not maintain an active mail server process for each mail format and protocol in use by the network.

2.7.3. A Distributed, Fault-Tolerant Program

The existence of multiple computers connected by a network makes it possible to construct a program that continues its execution despite machine failures. The basic idea is to implement the program as two processes: a *master*, which normally does all the work, and a *backup*, which stands by in case the master dies [Bartlett 81]. The master sends state information to the backup so that, if the master dies, the backup can quickly reconstruct a master, which then continues execution. The Butler is used to locate hosts for the master and backup processes. Figure 2-3 illustrates the configuration of master and backup processes before and after a machine failure. If the backup dies before the master, the master reconstructs a backup process and continues.

BEFORE:



AFTER:

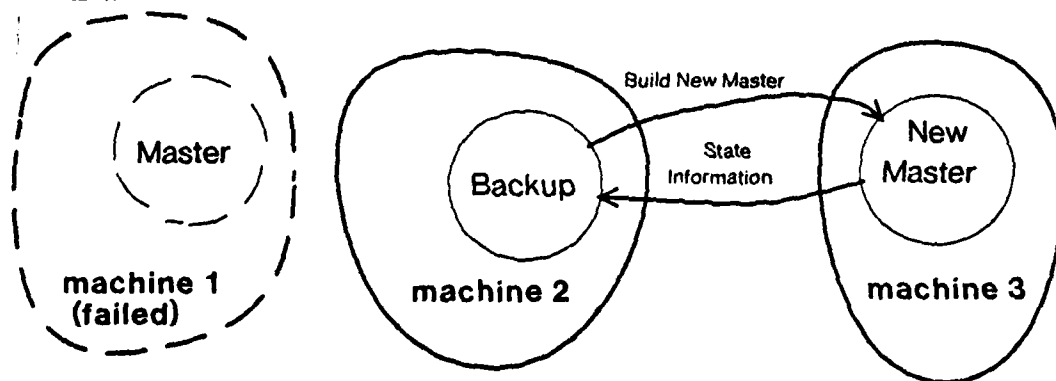


Figure 2-3: Using multiple machines to build a fault-tolerant program.

Special precautions must be taken for this technique to work. Ordinarily, a host Butler will expect to maintain a network connection to the agent. If the agent disappears, the host assumes that the requested services are no longer necessary, and the guest is aborted. For the fault-tolerant program, the host must be instructed (as part of the negotiation) not to abort the guest if the agent dies. Thus, the guest (backup process) survives and can create a new master. There must always be several machines that are willing to execute master and backup processes.

A reliable program might be used as a server that holds messages if the destination machine is down. This technique could also be used for computationally intensive programs that can survive crashes without restarting, and further applications of fault-tolerant distributed programs are given by Shoch [Shoch 82].

2.7.4. Digital Music Synthesis

This application is actually an instance of the distributed program example (see section 1.2.4), but it is included here to illustrate how machine failures can be handled in a particular program. Digital music synthesis is a computationally expensive task [Moorer 77]. For example, the summation synthesis technique³ requires on the order of 10^6 multiplications, additions, and table lookups for each second of sound for each synthesized instrument, and there might be tens or even hundreds of instruments.

To distribute the synthesis task, a master process is invoked by the user to acquire resources from a number of host Butlers. The remote machines execute simple synthesis programs that accept short *sound descriptions* and output digitized audio results. The sound descriptions are obtained from the master which in turn obtains them from a score. The master also mixes the results, and writes them to a file as illustrated in Figure 2-4.

To handle machine crashes and revocation of resources, the master saves each sound description until the corresponding synthesized sound is returned by the remote synthesis program. If the remote program is aborted by the host or by a machine crash, the master retransmits the parameters to another synthesis program. Recovery is simple because the synthesis programs are pure functions: as long as the arguments are known, the master can retry the function until its execution succeeds.

³Summation synthesis constructs a signal by adding sine waveforms of various frequencies and amplitudes.

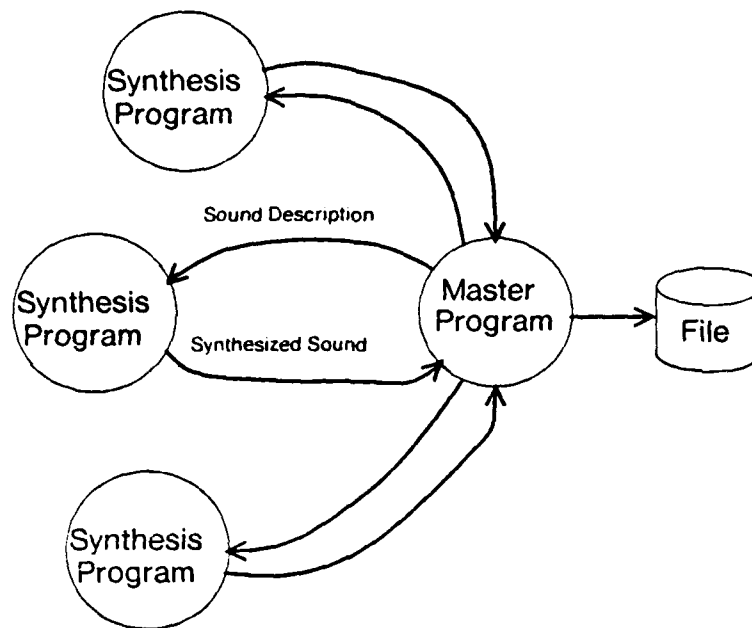


Figure 2-4: A distributed music-synthesis program.

2.7.5. Execution of Engineering Test Programs

Another application of the Butler is in automating the execution of test programs, which are often run as part of a preventive maintenance program. For this application, a machine responsible for testing periodically requests remote Butlers to execute hardware diagnostic programs. Since these programs require privileges to access microcode and device registers, machine owners will want to authorize only certain people to run them. This is easily accomplished by entering the desired policy in the policy database.

If the diagnostic program discovers a problem or crashes on the remote machine, the invoking program (running locally) can report the problem to an engineer. Also, test results can be saved in a database for statistical analysis.

2.8. Summary

The Butler is a system program that facilitates resource sharing in a network of personal computers. The Butler serves the roles of *agent* to locate resources and *host* to loan them. Although most of the Butler's activity involves the initiation of sharing, the Butler also deals with exceptional conditions arising from resource revocation and machine failures. The Butler is primarily concerned with providing support in the following areas: protection, negotiation, and policy administration.

The Banker is a system program that provides accounting services to other servers and application programs. The Banker is used to keep track of the resource utilization of programs so that limits can be enforced and so that programs cannot launder resources.

Chapter 3

Security and Protection

Maintaining security in a network of personal computers is a difficult problem because resources are distributed and often physically insecure. In addition, since a communication network connects all components, there are many opportunities for a malicious user to intercept, forge, or manipulate network messages. Furthermore, the network provides a convenient channel through which a malicious user can attack remote machines.

It is important to realize that all security problems exist in the context of a system that has certain properties (for example physical security and a secure operating system). Further assumptions are sometimes made (users will not tap terminal lines). If one is allowed to make arbitrary assumptions, then security problems can often be greatly simplified, but the resulting solutions may not be applicable to many real systems. Consequently, the designer must find a balance between the level of security obtained and the extent to which simplifying assumptions are made, and the reader must be careful to understand the problem as well as the solution. In this chapter, I try to emphasize the assumptions that are critical to the security of the proposed mechanisms.

3.1. Basic Assumptions

The Butler resides at a high level of control in the personal computer and depends upon lower levels of the system for support. Since the lower levels are critical to the security of the Butler, our presentation will necessarily include a description of how protection mechanisms are implemented "from the ground up". It should be noted that most of the security measures employed are based on known techniques. It is only at the highest level, that of the Butler, that any new ideas are considered.

3.1.1. Protected Address Spaces

Because we are interested in using machines to perform many tasks, it is assumed that each machine can support many mutually protected processes, meaning that no process can affect another without permission by such means as modifying registers or writing into another process's address space. Processes can interact however, by protected communication facilities provided by the system.

3.1.2. Protected Microcode

Writable control stores raise a special problem. With most machines, direct user access to microcode cannot be allowed if we want to prove anything about the behavior of the machine. Therefore, it must be possible for the operating system to prevent a user from directly altering any microcode or microprocessor state in the machine.

This does not mean that users should never write microcode; it only says that a machine owner must be able to protect himself *when he chooses* in order to guarantee some assumptions about protection.

3.1.3. Protected IPC

Interprocess communication is also protected. A process has control over the messages it sends and receives, and the right to send a message to a process *cannot ordinarily* be fabricated without the cooperation of that process. (Special provisions are made, however, to establish communication with a process when it is first created.) We will see that a protected IPC facility is important in the construction of higher-level protection mechanisms.

3.1.4. Network Encryption

Another assumption is that personal computers can communicate with one another over a network. To facilitate secure communication, each personal computer is equipped with an encryption device that implements either a conventional encryption algorithm or a public-key encryption algorithm. It is assumed that the speed of the encryption device is not a significant factor in the performance of the system. With the protocols to be described, computers can use these encryption devices to communicate securely.

3.1.5. Central Trusted Server

The last assumption is that the network is connected to a physically secure central authentication server, or CAS, which must be trusted by all users of the network. The CAS may be replicated to increase the reliability of service. If the assumption that a single machine is secure is too severe, then it is also possible to rely on several authentication servers in such a way that the network security is only compromised when the security of all authentication servers is compromised. The necessary protocols for a multiple authentication server scheme are more complicated and difficult to analyze than those for a single CAS, and will be discussed near the end of this chapter in Section 3.7.

Our general approach is based on the Spice system under development at Carnegie-Mellon University. The security mechanisms of the system will be described in a bottom-up fashion, in steps that correspond to layers in the implementation. At each step, facilities are added to existing ones to extend their functionality, and each step relies upon security mechanisms of the previous one. An outline follows:

1. The first step introduces a technique for loading a secure operating system on a personal computer. A trusted operating system must be loaded before any assumptions can be made about a machine's behavior.
2. A secure, intra-machine message-passing mechanism is introduced as a means of communication between processes on a single machine.
3. To allow processes on separate machines to communicate, network server processes are introduced. The network server is a part of the operating system which extends the intra-machine message-passing primitive so that a process can deliver messages across the network.
4. Encryption techniques are added to the previous step to achieve secure message-passing between machines.
5. Authentication protocols are then constructed so that we can determine *machine identities*. The identity of a machine is the identity of the user responsible for loading trusted system software.
6. Authentication protocols are developed to authenticate users to one another. A user is one who "logs in" or initiates a process, and does not necessarily have the same identity as his machine.
7. Given a secure network communication facility and a means of authenticating identities, protocols for resource sharing are developed.

3.2. Review of Encryption Techniques

Before addressing the issues of security for resource sharing, basic encryption techniques are reviewed. The principal use of encryption is to transmit information securely across insecure channels. This is accomplished by transforming the information so that only the receiver can recover the original information. More details can be found in [Popek 79].

3.2.1. Conventional Encryption

With conventional encryption, a function is defined that takes cleartext data and a key, and produces encrypted text. Using the notation in [Popek 79],

$$E = F(D, K),$$

where D is the data, K is the key, and E is the encrypted text. Another function, F' , is available to undo the encryption:

$$D = F'(E, K).$$

An important property of F and F' is that without knowledge of K , it is impractical to recover D from E . In other words, one cannot decrypt a message without the key. Also, even if the corresponding E is provided for any chosen D , it is impractical to derive the corresponding K . Therefore, one cannot break the code, even if one can obtain encrypted versions of chosen cleartext messages. The functions F and F' are publicly known algorithms, and are never changed, so a hardware implementation is possible. An example pair of functions can be found in the Data Encryption Standard (DES) of the National Bureau of Standards [NBS 77].

3.2.2. Public-Key Encryption

In conventional encryption, the same key must be used to encode and decode data. In public-key encryption, separate keys are used for encoding and decoding:

$$E = F(D, K),$$

$$D = F'(E, K').$$

Again, F and F' are publicly known. In addition to these functions there is an algorithm that generates a pair of keys (K, K') . Given K' , it must be impractical to derive K . Furthermore, F and F' are usually interchangeable: that is, one can encode with F' and decode with F . The keys K and K' are referred to as the *private* and *public* keys, respectively.

Since the encryption function is fixed, there is no need for the notation to explicitly mention it, and we will usually write $K(D)$ as an abbreviation for $F(D, K)$.

3.2.3. Redundancy for Authentication

The receiver of an encrypted message must normally check that the message was encrypted with the proper key, otherwise, random messages could be forged simply by issuing a random stream of bits. The sender therefore adds redundant information to the message which can be checked by the receiver. In some cases, such as text strings, there may be enough redundancy in the data that added redundancy is unnecessary. In this chapter, it is assumed that redundant information (a checksum, for example) is always added so that messages encrypted with the wrong key are detected.

Encryption can be used to authenticate the sender of a message. Suppose it is known that only machines A and B have a conventional key K. If A receives a message K(D), then the message must have come from B, since no other source could encrypt D with K. Machine A must be careful, however, since K(D) could be intercepted and retransmitted by another machine at a later time. This problem is usually avoided by first having A send B an arbitrary piece of data, perhaps based on the time of day, to include with D. This data is checked by A to insure that a message is not a replay of an earlier message. Sequence numbers are also added to the data to prevent the acceptance of copies. Kent [Kent 81] gives an excellent description of these and other techniques for security and authentication in computer networks.

3.3. Loading An Operating System Securely

The security of a machine depends largely upon its operating system, so there must be a way for a user to load a certified copy of an operating system into his machine. We will assume that the desired system is available from a file that can be accessed via the network.

It is assumed that the user's machine is initially in an unknown state, and perhaps the machine has been in the control of someone other than its owner. This situation could arise in several ways. The machine could have just undergone maintenance, or perhaps the machine was left unattended and accessible to other users. The machine may be located in a public place, or the user may have temporarily given away complete access to his machine, for example, by letting a guest process install its own microcode. In all of these cases, it is possible that the machine has been left with a modified, insecure, and possibly malicious operating system. A malicious system could act as a "trojan horse" and obtain rights of the machine owner when he logs in.

Since none of the machine state can be trusted, it must be possible to load a certified operating system without relying upon any alterable state of the personal machine. In fact, it must not be possible for alterable hardware, or for programs or data in alterable memory, to interfere with the security of the loading process. If no assumptions can be made about alterable state, then clearly some requirements must be imposed upon the hardware, which may include read-only memory.

To load a certified operating system securely, a bootstrapping procedure is used, beginning with a loader that is kept in read-only memory (ROM) on every machine. Each machine is equipped with a physical switch, called the *boot* switch that causes the processor to begin executing code from the loader. If the machine has a writable microstore, then either the hardware must initialize the store from a ROM, or part of the microcode must be resident in ROM. The boot switch also causes the micro-machine to begin executing this ROM-based microcode.

Since security depends upon information in ROMs, it must be impractical to replace them physically or logically. For example, even if a ROM is somehow sealed against removal from a processor circuit board, it may be possible to cut address and data lines to the chip and reroute them to another. Another path of attack is to *replace some portion of the personal computer*, such as a printed circuit board, containing the boot ROM. In the extreme case, the entire personal computer could be replaced with one in which a malicious boot ROM is installed. The degree of precaution necessary to guard against such physical attacks depends upon the value of the resources being guarded and the relative effort required to breach other walls of security.

In any case, we will assume that the user trusts the hardware on his machine which includes a ROM-based loader. The problem is to construct a loader that will enable the user to load a certified operating system over the network. In practice, it may be advisable to reduce the problem of loading the operating system to that of loading a reliable loader, using a small bootstrap loader based in ROM. This allows changes in the loader without changing the ROM, which is expensive to replace.

3.3.1. A Loader Based On Public-Key Encryption

If public-key encryption hardware is available, the loader holds a public key, and one or more file systems hold a copy of the certified operating system, encrypted with the private key. The private key itself should be kept in an extremely secure place, since disclosure of this key renders the boot ROMs in all machines insecure. The file systems that store the encrypted software need not be secure at all, since no secret information is maintained on-line.

The loader requests a copy of the operating system from some file server (possibly located through a network name server). The file is delivered to the loader which decrypts it using the public key. Notice that without the private key, an operating system cannot be forged. Also notice that no secret keys need be stored in the loader or in the file server. This implies that the boot ROM does not contain any secret information. The ROM must be protected from modification, but not from access.

One problem with this scheme is that a malicious file server could send an out-of-date version of the operating system, perhaps one with known bugs. The next two sections present schemes that solve this problem, but require secret information to be maintained on-line.

3.3.2. A Loader Based On Conventional Encryption

Conventional encryption has the property that the encoding and decoding keys are identical, so both must be secret. We will assume that each user carries a secret key, K , known only to him and to the CAS. After the user pushes the boot switch, he must enter an identifier I and his key K . This may be accomplished by inserting a card into a magnetic card reader, by inserting a ROM into a socket, or by manually typing information into the machine.

The loader creates a new key at random called R . Then, the loader encrypts I and R with K and sends them to the Central Authentication Server (CAS) along with identifier I in cleartext.

When the CAS receives the messages, it uses I to find K in a securely stored table. The CAS can then use K to decrypt I and R from the remainder of the message. Here, K serves two purposes: it functions as a password authenticating I , since I can only be encrypted by someone who knows K , and it serves to encrypt R which is used in further messages to and

from the CAS⁴. To complete the loader protocol, the CAS either sends a copy of the operating system, encrypted with R, or it arranges for a trusted file server to do the same.

To see that this scheme is secure, note that only the loader and the CAS have copies of K, and K is never transmitted. Therefore, only the CAS can decode R, and at this point only the CAS and the loader can have copies of R. If the loader receives a recognizable operating system, it must have come from the CAS, since the operating system is encoded with R. A malicious machine cannot record the messages from the CAS for replay at a later time, since a new R is generated by the loader at the beginning of the protocol.

3.3.3. A Hybrid Scheme

An advantage of the previous scheme is that conventional encryption is used for messages. Conventional encryption is less expensive than public-key encryption at the present time. However, a drawback of the previous scheme is that the user must enter an identifier and key. This disadvantage can be removed by performing a small amount of public-key encryption and decryption in software.

As before, the loader first constructs a random conventional key R. This key is encrypted in software with a public key K' corresponding to a private key K which is only known to the CAS. The encrypted R is sent to the CAS which decrypts R in software. Now R can be used as a conventional encryption key to encrypt the operating system and send it to the loader which uses its copy of R for decryption.

Only the CAS has private key K, so only the CAS can acquire R. Without R, a program cannot send a recognizable operating system to the personal machine, and therefore, the personal machine is assured of receiving a certified operating system.

A problem with this scheme is that the private key must be used frequently on the CAS. If this key becomes known outside the CAS, then all loader ROMs have to be replaced. However, this hybrid scheme does remove the requirement that machine owners carry or remember secret keys.

⁴K could be used for further messages, but it is safer to avoid keeping K in fast memory where it is more vulnerable. Although R is no less vulnerable, its useful lifetime only extends to the time of the next connection to the CAS.

3.3.4. Certifying an Operating System

A disadvantage of all of the schemes above is that they require a copy of the complete operating system to be transmitted. It is possible to reduce the network traffic and possibly load the operating system faster if the personal computer has a local disk, provided that the local copy can be certified. The loader can certify the system by computing a secure checksum or hash function and comparing the result to the correct value obtained from the CAS. The checksum algorithm must have the property that it is impossible for someone to construct a message that differs from a certified one but has the same checksum. A technique for constructing a checksum has been suggested by Davies [Davies 81].

Davies' algorithm uses successive DES encryption of an initial value I with keys obtained by dividing the message into blocks. (In the case of the loader, the operating system is the message.) Let the message be

$$M = m_1, m_2, m_3, \dots, m_n,$$

where m_i is the i^{th} block of the message. The checksum C is obtained from

$$C = m_n(m_{n-1}(\dots m_2(m_1(I)) \dots)),$$

that is, encrypt I using the first block as the key, then encrypt the result using the second block as the key, and so on. The final result is the checksum.

3.3.5. The Local File System

Earlier, we made the assumption that no alterable state is to be trusted when an operating system is loaded. This implies that we cannot trust any state on the local disk. In particular, we cannot trust the directories and other data structures on the disk to be consistent.

Unfortunately, the cost of restoring the contents of an entire disk from a central server may be very high. Several alternatives are described below.

3.3.5.1. Encryption

The machine owner can encrypt all information written to the disk. This allows the system to detect unauthorized modification of the disk. A drawback of encryption is that only the owner can load and start the machine since knowledge of the disk encryption key is required to use the disk. Furthermore, the owner cannot allow a guest to do anything unsafe, such as to load microcode, since that privilege might be used to steal the disk encryption key.

3.3.5.2. Redundancy Checks

It is possible to write file systems that do not rely on disk-resident data structures other than the file data blocks themselves. For example, in the Alto [Thacker 82] and Perq [Perq 81] computers, each disk block has a header containing the file identifier and block number to which it belongs. This solves the problem of corrupted index and directory structures, since the file system can recover from inconsistent data structures on the disk.

Encryption can then be used to prevent unauthorized access to personal files, and the checksum technique of Section 3.3.4 can be used to certify operating system and other public files. With this technique, a guest can still tamper with the disk, but he cannot steal information, and he cannot construct a trojan horse by manipulating disk storage.

3.3.5.3. Physical Protection

The local disk and a small dedicated processor can be secured physically and accessed over a high-speed bus. The disk processor implements typical file access functions, but prevents direct disk access which might be used to corrupt the disk structure. The processor could also enforce file protection, or encryption techniques could be used as described above.

3.4. Secure Intra-Machine Message Passing

The next task is to provide for secure intra-machine messages. This is a relatively simple problem, since the following assumptions are made:

1. A known and trusted copy of the operating system is loaded.
2. The operating system is secure. By enforcing separate address spaces for all processes, the operating system protects itself against application programs.

This is exactly the situation found in conventional time-shared systems, and there are several examples of time-sharing operating systems that provide secure message passing, for example, Hydra [Wulf 74], and Demos [Baskett 77].

In this dissertation, we will consider the IPC mechanism of Accent, the Spice operating system kernel [Rashid 81]. Accent implements an abstract object called a *port*. Accent protection mechanisms prevent direct access to the representation of ports, but several operations involving ports may be invoked.

An *AllocatePort* operation creates a new port and returns a name that can be used by the application program to reference the port. A *Send* operation delivers a specified message to a port, and a *Receive* operation retrieves a message from a port:

AllocatePort returns *PortName*
Send(*Message*, *PortName*)
Receive(*PortName*) returns *Message*

The actual interface to Accent IPC is more elaborate, but this outline serves to illustrate the important concepts.

As indicated above, programs reference ports indirectly through port names. A port name is used by a process to reference a port. For each process, Accent maintains a table of correspondences between port names and ports. This level of indirection prevents processes from forging port tokens and avoids some naming conflicts since two processes can have different local names for the same port.

A port is made accessible to another process by sending it in a message. To send a port, an indication is made by the sender that part of his message is to be interpreted as a port name. Before delivering the message, Accent translates the name to one through which the receiving process can access the port.

A set of rights is associated with every port name. The most important rights are *send* and *receive* rights. Only one process can have receive rights on a port, but many processes can have send rights. When a port is sent in a message, the sender indicates what rights are to be sent.

This is a very condensed description of the message-passing facilities of Accent. The reader is referred to the description by Rashid and Robertson [Rashid 81] for more details.

3.5. Secure Inter-Machine Message Passing

The next task is to provide secure message passing between machines. I first describe how messages can be sent across a network and then show how to do it securely.

3.5.1. Network Servers

Message passing can be extended to the network by introducing *network server processes* [Rashid 81]. A network server is a part of a machine's operating system; it is transparent to an application program, and its function is to translate between intra-machine messages and network messages. To illustrate this, suppose process A on one machine wishes to send a message to process B on another (see Figure 3-1).

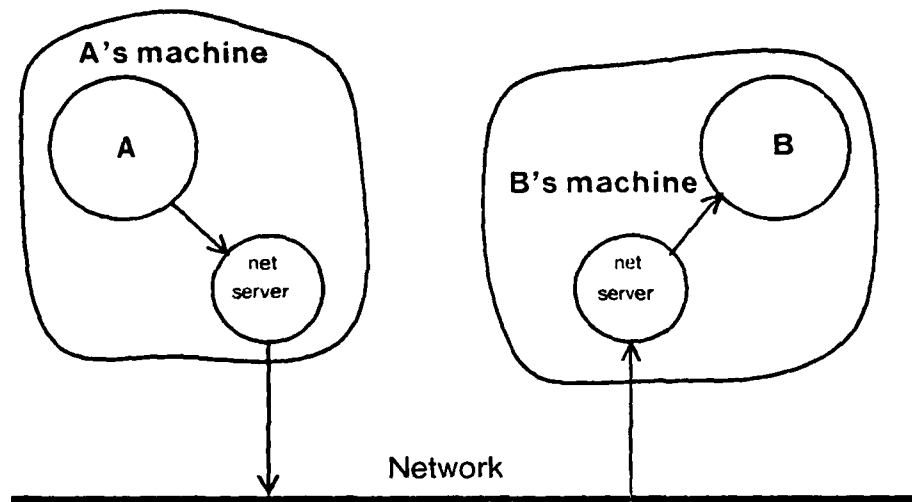


Figure 3-1: The use of network servers to achieve transparent inter-machine communication.

Process A has a port, P_A , which it uses to send messages to B. Since B is on a remote machine, it cannot directly receive messages from this port. Instead, a network server process receives the message from A. Next, the port on which the network server receives the message is used to find a virtual circuit [Tanenbaum 81] on which to forward the message. The message at this point may need to be translated from its operating system message format to a network message format. The message is then sent to network server B. Here the virtual circuit over which the message is received is used to look up a port, P_B , on the remote machine. If necessary, the message is translated back into its original operating system format. Finally, the message is sent to this port and received by B. It is important for network transparency that A and B both use ordinary intra-machine message primitives to send and receive messages across the network. This is made possible by interposing network servers between the application processes and the network.

3.5.2. Passing Port References

Process A can send a port name for port Q_A across the network in the following manner. First, A sends send rights for Q_A to port P_A . (Process A has receive rights for Q_A). The message is received by the network server, which locates or allocates a virtual circuit corresponding to Q_A . An identifier for the circuit is then sent to network server B, which locates or allocates a port Q_B to correspond to the virtual circuit. Finally, Q_B is sent as an Accent IPC message to port P_B . Again, the translation from kernel messages to network messages and back is transparent to A and B.

Some additional bookkeeping is used to optimize routing. For example, if B passes a reference to Q_B on to a third machine C, messages from C to A go directly to NS_A rather than indirectly through NS_B .

3.5.3. Secure Network Communication

Although the techniques above can make the network logically transparent, they do not extend the security of intra-machine messages to inter-machine messages. Network security is accomplished through encryption. We will assume that conventional encryption hardware is to be used. Referring to Figure 3-1, if network server A (NS_A) is to communicate with network server B (NS_B), then each must have an encryption key, say K_{AB} . In general, there must be a key K_{XY} for each pair of servers X and Y. The problem is to obtain the key at machines A and B without ever transmitting the key in the clear or revealing it to an untrusted party. The central authentication server is used for this purpose.

We will assume that each machine has a key that it can use to communicate securely with the CAS. This will be referred to as the *CAS connection key*. The key R obtained by the loader in sections 3.3.2 or 3.3.3 can be used as the connection key. In addition, the CAS must have an identifier, the *CAS connection identifier*, associated with each CAS connection key.

Either NS_A or NS_B can initiate the sequence to obtain a key K_{AB} , but we will assume that NS_A begins. NS_A creates a random key K_{AB} and sends K_{AB} and the connection identifiers I_A and I_B of NS_A and NS_B , all encrypted with the connection key of NS_A , to the CAS:

$$C_A(K_{AB}, I_A, I_B) \rightarrow \text{CAS}$$

The CAS uses I_B to find the connection key, C_B , for NS_B . It then encrypts K_{AB} and I_A with C_B and sends them to NS_B :

$$C_B(K_{AB}, I_A) \rightarrow NS_B$$

Now NS_A and NS_B each have key K_{AB} and can communicate securely. The CAS should destroy its copy of K_{AB} after sending it to NS_B . Figure 3-2 illustrates the messages involved.

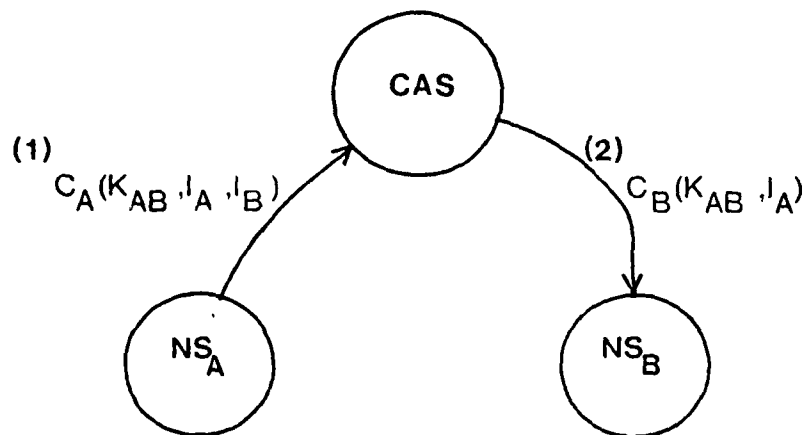


Figure 3-2: Distributing an encryption key.

3.5.4. Machine Authentication

It will become important for one machine to determine the identity of another, based on the person responsible for loading trusted software. This is easily accomplished as part of the key distribution protocol by making the CAS connection identifier be the machine identity. The CAS knows this identity since it is authenticated at the time of initial connection to the CAS (see Sections 3.3.2 or 3.3.3).

3.6. Authentication

We now move from the machine to the process level of abstraction. In the previous section, we were concerned with machine identities. We will now see how to authenticate *user identities*, which are entered when a user initiates a session with his machine. User and machine identities may differ. Our problem is to establish a connection via Accent IPC ports between two processes, each of which knows the identity of the process at the other end.

Again, the central authentication server (CAS) is used, and the protocols are similar to

those used for key distribution. The similarity is not so surprising when it is realized that a secure port system is analogous to a message-passing system based on public-key encryption. Receive rights of a port correspond to a private key and send rights correspond to a public key. Just as a message encrypted with a public key can only be read by the process with the corresponding private key, a message sent to a port can only be received by the process with the corresponding receive rights. Although encryption is usually reversible, that is, a message can be encrypted with the private key and decrypted with the public key, the port system is strictly one-way.

3.6.1. Getting Started

Each personal machine initially requires send rights on a port, called the *machine-to-CAS* port, to which the CAS has receive rights. Ordinarily, port names are only copied by sending them in a message to another port, so the CAS cannot transfer a port token to another machine by ordinary means. Instead, the network servers on the CAS and personal machines must provide non-transparent operations that establish the initial port connection. This is convenient to do at the time that the operating system is loaded, since that is when a secure *network level connection* is established with the CAS. The operating system uses this port to establish secure connections between users and the CAS.

For our purposes, a *secure connection* is a pair of ports that allows two processes, A and B, to communicate. Each process has receive rights for one of the ports, and send rights for the other. Send rights have not been given to any untrusted processes, so when a message is received by A through one of the secure connection ports, it can be assumed that the message was sent by B or by some agent acting on behalf of and trusted by B. Similarly, messages received by B are assumed to have originated with A or an agent of A.

An *authenticated secure connection* between A and B is one in which the identity of A has been authenticated to B, and the identity of B has been authenticated to A. More precisely, when the connection is established, A is given send rights for a port and proof that B *has claimed* to own the port's receive rights. Similarly, B is given send rights for a port and proof that A *has claimed* to own the port's receive rights. Nothing can stop A or B from misusing or giving away rights to ports in the connection, but if A and B cooperate, then no outside party can interfere with the establishment of a secure and authenticated path of communication between A and B.

3.6.2. Connecting to the CAS

When a user logs in, a secure authenticated connection is established between the user and the CAS in the following manner: The operating system first creates a port for the user to receive messages. Then, send rights for that port, the user's name, and the user's password⁵ are sent to the CAS, using the machine-to-CAS port obtained in Section 3.6.1. The CAS checks the password against a table of stored name/password pairs to authenticate the user, and a new CAS port is created to receive requests from the user. Send rights are then returned to the user's port, so that the user and CAS now have an authenticated secure connection.

3.6.3. Establishing a Connection to a Process

Using the central authentication server as a trusted intermediary, two processes can establish an authenticated secure connection. (Hereafter referred to simply as a "connection".) For example, a user process can communicate with a server over a connection to request services and obtain results. Also, Butlers use a connection to determine the identity of the other Butler when resources are shared. To illustrate the connection protocol, suppose processes A and B wish to set up a connection. Figure 3-3 illustrates the messages involved.

In the first step, A creates a port P_A for the receive side of the connection. Send rights for this port and a random transaction key are sent to the CAS in a "register port" message. The CAS retains this information. Next, A sends B a "connect request" message. The message contains the transaction key and is sent to a publicly known port belonging to B. Process B creates the second port of the connection, P_B , and sends the transaction key and send rights for the new port to the CAS. The CAS responds by finding the information from A with a matching transaction key and the name B. The identity of B and send rights for P_B are sent to A. The identity of A and send rights for P_A are sent to B. A and B now have a secure authenticated connection.

This same protocol can be used to authenticate an existing connection. For example, A and B may exchange P_A and P_B in an insecure way for the sake of efficiency, exchange

⁵The password can be a text string typed by the user, a number stored on a magnetic card, or a more secure identifier such as a machine-read fingerprint or voiceprint.

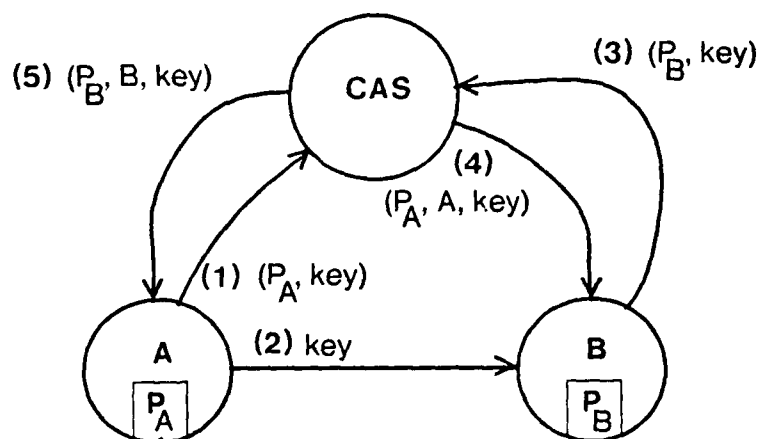


Figure 3-3: Establishing an authenticated secure connection.

several messages, and then decide to authenticate the connection. This technique will be used to make resource negotiation more efficient (see Chapter 5).

3.6.3.1. Authorization

The CAS can also be used to implement an authorization service based on group membership. To do this, the CAS maintains a database of groups and membership relations, where only an authenticated group owner can alter the membership of his group. Servers, including the Butler, are instructed to grant rights to members of certain groups. (This is often more convenient than granting rights directly to individual members.) Servers will now need to know group membership information as well as the identity of the process at the other end of a connection, but this extra information can be supplied by the CAS during the connection protocol without sending additional messages.

3.7. Using Multiple Authentication Servers

A central authentication server provides a single point of attack in an otherwise highly distributed security system. To avoid this weakness, multiple servers can be used in such a way that the system security is only compromised when the security of all servers is compromised. This is unlikely if servers are implemented and maintained separately.

3.7.1. Loading an Operating System

After loading an operating system using the single authentication server protocol, the ROM-based loader computes a secure checksum. This is compared against a checksum independently obtained from each authentication server. All authentication servers would have to conspire to prevent the loader from detecting a forged operating system.

3.7.2. Secure Network Communication

Network servers NS_A and NS_B in Section 3.5.3 obtain key K_{AB} as follows. First, the protocol in Section 3.5.3 is carried out with each authentication server. The resulting keys are exclusive-or'ed together to form a composite key K_{AB} . No authentication server will know K_{AB} , and K_{AB} can only be constructed if all authentication servers reveal their contribution to the composite. This assumes that any string of bits is a suitable key, which is true of DES keys.

3.7.3. Authentication

The use of multiple authentication servers to enhance the security of connections made through ports is an unsolved problem. It is simple, however, for the operating system of one machine to establish a secure authenticated connection over ports to the operating system on another machine, without relying on the CAS at all. No authentication server is necessary once an authenticated and encrypted channel between network servers is established. Messages to a network server are self-authenticating since they are encrypted with a secret key known only to one other network server, and the identity of the system with the other copy of the encryption key is determined at the time the key is obtained from the CAS. The network servers can simply cooperate to generate a pair of ports for use by their respective operating systems.

3.8. Resource Sharing Protocols

Resource sharing is supervised by Butlers. For protection reasons, a separate Butler executes on each machine, and the Butler itself never uses resources on another machine. Since malicious behavior can only be obtained when an uncertified operating system is loaded, it is desirable to identify the person who loaded the system. Before borrowing

resources, the agent Butler finds a suitable host and performs the authentication protocol, authenticating the identities of the two Butlers. The identity of a Butler is the identity of the person responsible for loading the operating system.

The major task of the host Butler is to protect its machine from guests. Several techniques are used:

1. **Policy Enforcement.** The Butler only grants access to resources permitted by the machine-owner's policy.
2. **Accounting.** The Banker keeps track of resources used by the guest. This allows fine control over resources and prevents unauthorized acquisition of resources by laundering requests through servers.
3. **Revocation.** The Butler has absolute control over all guests. If a guest process attempts to exceed its authorized resource limits, then the Butler can issue a warning, deport the guest, or destroy it and all of its rights.

The protective role of the Butler in resource sharing is elaborated throughout the remaining chapters.

3.9. Summary

A layered approach is taken to achieve a secure system whose foundation is an operating system kernel that supports separate address spaces and secure message passing. To extend the message-passing primitives of the local operating system to the network, encryption and network servers are used. Network servers establish secure, authenticated connection through the use of a secure, central authentication server and key exchange protocols. At the highest level of protection, the Butler and Banker protect the local machine from the guest by administering the machine owner's policies, performing accounting, and possibly revoking guest rights.

Chapter 4

The Banker

The Banker is a process that provides accounting services. The Banker's most important role is to implement protection mechanisms so that the host Butler can protect its machine against malicious guests, but the Banker also implements an exception-handling mechanism to assist recovery from the revocation of rights. In addition to the Butler, any process can use the Banker to restrict or control the resources used by a subsystem; for example, the command interpreter for the operating system could use the Banker to terminate runaway programs invoked by the user, even if that program forks into many processes.

The Banker can also be viewed as a type-manager for an extended form of capability called an *account*. An account represents a user's right to use a specific resource, and accounts can be passed like capabilities to other programs in order to grant rights. (Actually, a *signature* representing multiple accounts is passed.) The Banker has an abstract view of resources, and new physical or abstract resources can be described to the Banker at any time. The Banker has no special knowledge of servers; thus, ordinary application programs can become servers with all of the protection mechanisms available to system programs.

The next section presents an overview of the facilities provided by the Banker, and shows how they can be used for protection. In Section 4.2, abstract types for representing rights and bank accounts are defined. The two sections that follow, 4.3 and 4.4, define specifically the operations implemented by the Banker. To achieve protection, the Banker must be used in cooperation with server processes that directly control resources. Section 4.5 describes the standard mode of interaction between the Banker and servers.

4.1. Server/Customer/Banker Transactions

Consider a *customer* process that wants to obtain resources from a server. The *customer* has one or more *accounts* with the Banker, and in each account are rights that authorize the customer to purchase resources from a server. Each customer has an unforgeable reference to his accounts, called a *signature*, which is a capability that enables servers to make withdrawals from the customer's account. Upon request from a server, the Banker performs withdrawals if possible, and replies with an overdraft message when the withdrawal amount exceeds the customer's account balance.

No built-in protection is provided at this level to prevent servers from misusing the customer's signature. It does not seem worth any extra precautions since a client depends upon servers in many other ways. Furthermore, it is difficult to recognize misuse. For example, one might consider preventing the server from passing the signature off to another process; however, in many cases a server must call another server on the client's behalf. In these cases, it is necessary to transmit the signature.

Customers are arranged in a dependency tree. Most of the local Spice system resources are initially represented by accounts at the root of the tree. The Butler allows guest and resident processes to share currency in these accounts in a restricted way by creating subaccounts, which are described below.

In summary, several kinds of objects are implemented by the Banker. A *customer* is a mapping from a *signature* to a list of *accounts*. An account holds rights that correspond to some *resource*. Clients use signatures as capabilities to obtain resources from servers, which consult the Banker to determine the rights that correspond to a given signature.

4.2. Representing Rights

The Banker must have a representation for rights. In this section, we will define several abstract types, which are used not only by the Banker, but also the Butler, the policy database, and servers that use the Banker for accounting. A type called *RightsValue* is used as a resource-independent representation of rights. Another type, *Currency*, combines a *RightsValue* with a resource name to form a representation of resource-specific rights. The type *Account* is slightly more elaborate than *Currency*, and is used by the Banker to keep

track of current rights. Since collections of rights over many resources must often be manipulated, additional types called *CurrencyList* and *AccountList* are defined. *Important:* these types and operations are primarily for use internal to programs. They should not be confused with the external services provided by the Banker, which are defined in Sections 4.3 and 4.4.

4.2.1. RightsValue Type

Two types of values, *integer* and *set*, are used to represent resource rights, and *RightsValue* is the union of these two types. Integer values are used to represent rights over homogeneous sets of resources, such as disk pages, where only a counter is required to specify the number of resource objects. Set values are used to represent heterogeneous resources like priorities or access to special devices. Another use of set values is to specify what operations are permitted on abstract objects. Set values are actually just an optimization, since heterogeneous resources can also be represented by multiple integer values, using one value per resource.

A possible representation of the type is:

type *RightsType* **is** (*IntVal*, *SetVal*);

type *RightsValue*(*TypeTag*: *RightsType*) **is**

```

record
  case TypeTag is
    when IntVal = >
      Amount: Integer;
    when SetVal = >
      Privileges: SmallIntSet;
  end case;
end record;

```

In this type declaration, a small set of integers is used to represent the value for set currency, but a list of strings or atoms would also be a suitable representation. In addition to access operations, a function, *Max*, is defined for objects of this type:

function *Max*(*R1*, *R2*: *RightsValue*) **return** *RightsValue*;

This function returns the integer maximum or the set union of its arguments. If *R1* and *R2* are of different types, then the *BadType* exception is raised.

4.2.2. Currency Type

Objects of type *Currency* have a *Name* that names a resource, and a *Value* that gives the *RightsValue* associated with that resource. The only operations on this type are to access the two components, and a possible representation is given below:

type *ResourceName* **is new** *String*;

type *Currency* **is**
 record
 Name: ResourceName;
 Value: RightsValue;
 end record;

The meaning of integer and set currency is interpreted by the corresponding server. For example, a file server might interpret a rights value as the number of allocated disk pages, while the kernel might interpret another rights value as the number of forked processes.

4.2.3. Accounts

Type *Account* implements objects with three components: *Name*, *Value*, and *Limit*. The *Name* is a resource name as in the currency type. The value field is a *RightsValue* that tells how much has been withdrawn from this account, and *Limit* is a *RightsValue* that gives the maximum allowable *Value*. A possible representation is given below:

type *Account* **is**
 record
 Name: ResourceName;
 Value: RightsValue;
 Limit: RightsValue;
 end record;

Before specifying the formal operations on accounts, we will give an intuitive explanation of account operations, and the interpretation of the limit and value components of accounts.

4.2.3.1. Operations On Integer Accounts

In integer accounts, the limit tells how many units of some resource type may be allocated, and the value field tells how many units have already been allocated. Debits are performed when resources are allocated, so to debit an integer type account, the debit amount is *added* to the account value. To credit an integer type account, the credit amount is *subtracted* from the account value. The balance of the account is the difference between the limit and the value:

$\text{balance} = \text{account.limit} - \text{account.value}.$

4.2.3.2. Operations On Set Accounts

Set accounts represent reuseable rights. In other words, if a user exercises a right represented in his account, that right is not then removed from his account. Note that this is different from the way integer accounts are handled.

The limit component of a set account tells what rights a user has with respect to some resource. The value component tells what rights are currently in use. Before a server performs a service requiring a given right, a debit of that right is requested. To debit a set currency type account, the account value is set to the union of the old account value and the debit amount. If the user wants to perform another operation that uses the same right, another debit will be performed, but there will be no change to the account value (the right is already in the set represented by the account value).

A credit is performed when a user is no longer exercising a right. The credit removes the right from the account value. More precisely, to credit a set type account, any element in the credit set is removed from the value set. Most servers will credit the account when the customer closes his connection with the server and returns all the resources that correspond to the account.

The *balance* of a set type account tells what rights the user has. This is simply the limit of the account. By analogy to integer account types, one would expect the set account balance to be the set difference between the limit and the value, but this tells what rights the user is not exercising. We want "balance" to mean the set of rights the user can exercise, and this set is precisely the account limit for set accounts.

Now that we have explained the purpose and meaning of account operations in an intuitive way, we will present a more formal description of account operations.

4.2.3.3. CreateAccount

```
function CreateAccount(Name: in ResourceName;  
                        Limit: in RightsValue) return Account;
```

The *CreateAccount* function returns an account with the specified name and limit. The value component is set to zero or the empty set, depending on the type of the specified limit.

4.2.3.4. Debit

procedure *Debit*(A: in out Account; RV: in RightsValue);

If *RV* is an integer value, *RV* is added to the value of the account. If *RV* is a set value, then the account value is set to the union of the value and *RV*. In either case, if the resulting account value would exceed the account limit, the account is not changed, and the *Overdraft* exception is raised with a parameter of type *Currency* that gives the amount of the overdraft. If *RV* and the value have different types, the *BadType* exception is raised, and no changes are made to the account.

4.2.3.5. Credit

procedure *Credit*(A: in out Account; RV: in RightsValue);

Credit subtracts *RV* from the value of the account. If *RV* is a set value, any member of *RV* that is a member of the account value is removed from the account value. As with *Debit*, the *BadType* exception is raised if *RV* is not compatible with the account value.

4.2.3.6. GetBalance

function *GetBalance*(A: in Account) return RightsValue;

GetBalance returns the balance of an account. For integer accounts, the balance is the difference between the account limit and the account value. For set accounts, the balance is simply the account limit.

4.2.3.7. GetLimit

function *GetLimit*(A: in Account) return RightsValue;

This function simply returns the *Limit* component of *A*.

4.2.3.8. GetValue

function *GetValue*(A: in Account) return RightsValue;

This function returns the *Value* component of *A*.

4.2.3.9. SetLimit

procedure *SetLimit*(*A*: in out *Account*; *R*: in *RightsValue*);

This procedure sets the *Limit* component of *A* to *R*. If the value of *A* exceeds *R*, then *A* is not changed, and an *Overdraft* error is raised with a parameter of type *Currency* that gives the amount of the overdraft.

4.2.4. CurrencyList Type

An abstract type, *CurrencyList*, is used to represent an unordered collection of resources. The operations are listed below:

4.2.4.1. CreateCurrencyList

function *CreateCurrencyList* **return** *CurrencyList*;

This function creates a currency list representing no rights (the empty list).

4.2.4.2. GrantCurrency

procedure *GrantCurrency*(*CL*: in out *CurrencyList*; *C*: in *Currency*);

If *CL* has no currency with the same name as *C*, then *C* is added as a new member of *CL*. If *CL* has a member, *m*, with the same name, then *m* is replaced with $\text{Max}(\text{Value}(m), \text{Value}(C))$ in *CL*. If *m* and *C* have different types, that is, one is set currency and the other is integer currency, then *CL* is unaffected and the *BadType* exception is raised.

4.2.4.3. Iteration Functions

function *Index*(*CL*: in *CurrencyList*; *N*: *Integer*) **return** *Currency*;

function *Length*(*CL*: in *CurrencyList*) **return** *Integer*;

These functions are used to iterate through each component of a currency list. The function *Index* returns the N^{th} currency record (based at one) of a currency list. *Length* tells how many records are present.

4.2.4.4. LessOrEqual

function *LessOrEqual*(*CL1, CL2: in CurrencyList*) **return** *Boolean*;

This function returns true if no currency in *CL1* represents more resources than the corresponding currency in *CL2*.

4.2.5. AccountList Type

The operations on account lists are *CreateAccountList*, *Deposit*, *Withdraw*, *SetLimits*, *GetLimits* and *GetBalance*. These are defined below:

4.2.5.1. CreateAccountList

function *CreateAccountList* **return** *AccountList*;

This function creates an initial account list that contains accounts for each resource, with all accounts initially having limits and values of zero (or the empty set for set accounts). Obviously, the implementor of account lists will want to do some encoding to avoid allocating an account for each of the infinitely many currency names.

4.2.5.2. Deposit

procedure *Deposit*(*AL: in out AccountList; CL: in CurrencyList*);

For each element *C* (of type *Currency*) in *CL*, this procedure locates an account in *AL* with the same name as the name of *C*. A *Credit* of the value of *C* is then performed on this account (see Section 4.2.3.5). If any credit to an account would raise the *BadType* exception, then no changes are made to *AL*, and the *BadType* exception is raised.

4.2.5.3. Withdraw

procedure *Withdraw*(*AL: in out AccountList; CL: in CurrencyList*);

For each element *C* (of type *Currency*) in *CL*, the corresponding account is found in *AL* and a *Debit* operation is performed (see Section 4.2.3.4). As before, if any currency types do not match, no changes are made, and the *BadType* exception is raised. If any account withdrawal would cause an overdraft, then the account list is unchanged and an exception, *Overdraft*, is raised with a parameter of type *CurrencyList* that specifies the extent of the overdraft.

4.2.5.4. SetLimits

procedure *SetLimits*(*AL*: in out *AccountList*; *CL*: in *CurrencyList*);

CL specifies a set of limits to be put in corresponding accounts in *AL*. If any new account limits are less than the corresponding account values, *Overdraft* is raised with a parameter of type *CurrencyList* that specifies the extent of the overdraft.

4.2.5.5. GetLimits

function *GetLimits*(*AL*: in *AccountList*) **return** *CurrencyList*;

This function returns the limits of all accounts in *AL*. Any account limit of zero or the empty set may be omitted from the returned currency list.

4.2.5.6. GetBalance

function *GetBalance*(*AL*: in *AccountList*) **return** *CurrencyList*;

This function returns the balance of all accounts in *AL*. Again, any balance of zero or the empty set may be omitted from the returned list.

4.3. Basic Banker Operations

In addition to types defined in the previous section, the following types are used in the interface to the Banker:

type *Signature* **is new** *Port*;

type *ResourceId* **is new** *Port*;

Type *Signature* is used to identify a customer, and type *ResourceId* is used to represent a resource. Both of these types are implemented as ports, since port names are convenient unforgeable identifiers, but this could be changed if another representation proved to be more suitable.

As mentioned above, accounts are organized in a dependency tree. To simplify this section, we will ignore this extra complication. Details of dependent customers will be presented in Section 4.4.

4.3.1. Defining a Resource

Any process can create a new type of resource by issuing the following command to the Banker⁶:

```
function CreateResourceType(Name: in String;
                           Server: in Port;
                           Resource: out ResourceId)
    return GeneralReturn;
```

The *Name* parameter is a printname for the resource type. The server parameter contains send rights on a port that is owned by the server that manages the resource. (This port is used primarily to notify servers of exceptions.)

The possible results are:

<i>Success</i>	<i>Resource</i> has the <i>ResourceId</i> that represents the new resource type.
<i>Duplicate</i>	No resource type was created because a resource with the same name already exists.

4.3.2. Creating an Account

A top-level account can only be created by a server that previously created a resource type. The command to create an account is:

```
procedure CreateAccount(Customer: in Signature;
                        Resource: in ResourceId;
                        Limit: in RightsValue);
```

An account is created for the customer with the initial value of zero (or empty set). The limit on the account is specified by the *Limit* parameter. The *Resource* parameter names a resource type created in a *CreateResourceType* operation. Since *ResourceId*'s are ports, we can make this parameter implicit by invoking the operation through the *ResourceId* port.

⁶Message interfaces throughout this dissertation are expressed using a procedure call syntax. See Appendix A for a description of the implied mapping between procedure calls and messages.

4.3.2.1. Initialization

To initialize an operating system, an initialization program first creates the Banker process. The Banker then creates a customer to represent all the local servers and the kernel. (Recall that a customer is just a mapping from a signature to a list of accounts. A signature is represented by a port.) An account is then created for this customer to allow the customer to create new resource types. The signature is returned to the initialization program. As each new server is created, the signature is passed as a parameter so that all servers have the signature and can use it to get resources and create accounts.

Some special provision must be made so that the initialization program can create the Banker, since the kernel will normally withdraw on an account before forking a process.

4.3.3. Withdrawals

Withdrawals are made using the following operation:

```
function Withdraw(Customer: in Signature;  
                  Resource: in ResourceId;  
                  Amount: in RightsValue;  
                  Notify: in Boolean)  
return GeneralReturn;
```

Again, if we assume *ResourceId*'s are ports, we can make the *Resource* parameter implicit by invoking the operation through that port. The Banker finds the account belonging to the *Customer* parameter for the resource corresponding to the *Resource* parameter and attempts to debit the account. The possible results are:

Success The account was debited. In the case of integer currency types, the amount is added to the account value. For set currency types, the new account value is the union of the old value and the amount.

NoSuchCustomer The signature is not valid.

Locked The accounts are locked because the customer is being deported (see Section 5.5.1).

OverDraft The amount of the withdrawal exceeds the current balance in the account. If notify is true, a message is also sent to the overdraft handler port for this customer. (The handler for the "root" customer is the Butler. More details on handlers are in the next section.)

Credits are made to accounts using the following operation:

```

function Deposit(Customer: in Signature;
                  Resource: in ResourceId;
                  Amount: in RightsValue)
    return GeneralReturn;

```

The parameters have the same meaning as those for the *Withdraw* operation. The possible results are:

Success The amount is subtracted from the customer's account value if it is of the integer currency type. In the case of set currency types, any element in the amount is removed from the account value.

NoSuchCustomer The signature is not valid.

NoSuchAccount The customer does not have an account for this resource type.

Locked The accounts are locked (see Section 5.5.1).

Balance inquiries can be made using the following operation:

```

function GetBalance(Customer: in Signature;
                    Balance: out CurrencyList)
    return GeneralReturn;

```

The operation either returns the status of the customer's accounts, or an error message:

Success A currency list is returned that describes the balance of the customer's account list.

Locked The accounts are locked (see Section 5.5.1).

NoSuchCustomer The signature is not valid.

4.4. Dependents

Dependents allow a customer to share his currency with other processes. Each customer has a possibly empty set of dependent customers, and dependent customers can be nested arbitrarily, so that the overall customer structure is a tree. The root of the tree is a customer representing the operating system, and all accounts originally belong to the system. Dependent accounts are created by the Butler whenever a new user comes into the machine, either as a guest from a remote machine or as a user logging in through the terminal.

The accounts of dependent customers are like those of the root customer, except that each has a parent account. The account limits can be used to set up several policies of account

sharing. Two interesting cases are the *account-split* policy and the *pooling* policy. An account-split policy partitions the account and gives a piece of it to each dependent. To implement the account-split policy, limits are set on dependent accounts according to the desired partitioning of currency. A pooling policy allows each dependent to withdraw as much as he needs from the parent account. To implement the pooling policy, the limits are simply set to infinity.

The simplest kind of pooling policy is one where all dependents pool all of the accounts. This can be accomplished by not creating any subaccounts at all. Instead, the signature of the parent is shared by all participants in the pool. This is likely to be the standard action taken when an application program forks.

To create a dependent, the following operation is performed:

```
function CreateDependent(Customer: in Signature;
                        Limits:   in CurrencyList;
                        Handler:  in Port;
                        Dependent: out Signature)
    return GeneralReturn;
```

The *Customer* parameter identifies the parent. *Limits* is a list of records, one for each account. The first element of each record is the printname of the resource, and the second is the account value. The *Handler* parameter contains send rights on a port. A message will be sent to this port if the new dependent tries to overdraw his account.

No currency is withdrawn from a parent's accounts when a dependent is created. The *CreateDependent* operation merely creates a new signature which can be given restricted rights to withdraw from the parent's accounts. The possible results are:

- | | |
|-----------------------|--|
| <i>Success</i> | The dependent is created. A signature to represent the new dependent customer is returned. |
| <i>Locked</i> | The accounts are locked (see Section 5.5.1). |
| <i>NoSuchCustomer</i> | The signature is not valid. |

4.4.1. Withdrawals On Dependent Accounts

A dependent account has a value that represents how much currency has been taken from the parent account. This may never exceed the limit set for the account. Consider the case of a root customer with one dependent. In this example, there is one resource type, (see Figure 4-1) and one account for each customer.

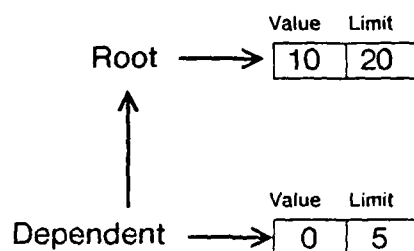


Figure 4-1: A customer's account and dependent's subaccount.

Henceforth, we will not distinguish between customers and accounts, since there is a one-to-one correspondence in this simple example. The root account has the value 10, and a limit of 20. The dependent has a limit of 5 and an initial value of 0. This means the dependent has not withdrawn any currency from the parent (root).

If the dependent withdraws 2, the new state is as in Figure 4-2. Notice that both the dependent and root accounts are debited.

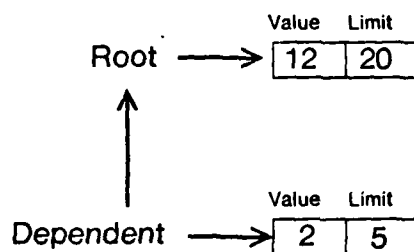


Figure 4-2: The dependent has withdrawn 2.

Now, suppose the dependent creates its own dependent with limit 5. (See Figure 4-3.) The new dependent account has the initial value 0. If the second dependent, called *Dependent2*, were to withdraw 1 from his account the account values of the root, *Dependent*, and

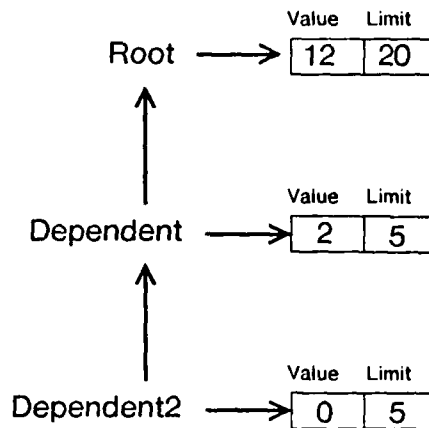


Figure 4-3: Adding another dependent.

Dependent2 would be 13, 3, and 1 respectively. Instead, assume that *Dependent2* tries to withdraw 4 from his account. The new account values would be 16, 6, and 4. This does not exceed the limit of *Dependent2*, but does exceed the limit of the first dependent. The withdrawal is not performed, and an overdraft message is sent to the server attempting the withdrawal. If the withdraw operation's notify parameter is true, a message is also sent to the overdraft handler of the first dependent. Notice that a message is sent to the handler for the first dependent, even though the attempted withdrawal is from an account of *Dependent2*. This happens because the limit of *Dependent2* would not be exceeded by the operation, but the limit of the first dependent (5) would be exceeded by one.

4.4.2. Access To Account Limits

The *GetBalance* operation is also defined for dependent accounts. The balance of an account is the maximum withdrawal that can be made successfully. For integer currency types, a recursive definition of an account balance is:

```

Balance(Account) = if Account is dependent
  then Min(GetLimit(Account) - GetValue(Account),
           Balance(parent's account))
  else GetLimit(Account) - GetValue(Account)
  
```

For set currency type accounts, the definition is:

```

Balance(Account) = if Account is dependent
                    then Intersect(GetLimit(Account),
                                   Balance(parent's account))
                    else GetLimit(Account)

```

In addition to the *GetBalance* operation, *GetLimits* and *SetLimits* operations are defined for subaccounts. The operations are defined as follows:

```

function GetLimits(Customer: in Signature;
                  Limits:   out CurrencyList)
return GeneralReturn;

```

The possible return values are:

Success A list of non-zero or non-empty limits is returned.

Locked The accounts are locked (see Section 5.5.1).

NoSuchCustomer The signature is not valid.

Limits can be changed with the *SetLimits* operation:

```

function SetLimits(Customer: in Signature;
                  Parent:   in Signature;
                  Limits:   in CurrencyList;
                  Notify:   in Boolean)
return GeneralReturn;

```

The *Customer* parameter identifies the customer whose limits are to be changed. The *parent* parameter identifies the customer's parent. Only parents can change limits, so this parameter must be included to authenticate the caller. The *Limits* parameter lists one or more limits to be changed. Customer account limits are changed accordingly. If no account exists for some record in *Limits*, a new account is created with the specified limit. If *SetLimits* attempts to reduce an account limit below the current value, an overdraft error is reported. The possible return values are:

Success The account limits have been set as specified.

NoSuchCustomer The signature is not valid.

Locked The accounts are locked (see Section 5.5.1).

Overdraft At least one of the specified limits is less than a current account value. If notify is true, an overdraft message will also be sent to the overdraft handler for this account.

WrongParent The parent parameter does not match the customer's parent's signature.

4.5. Server Protocols

There is a standard protocol to be observed by servers that use the Banker's accounting services. The object of the protocol is to support deportation and resource revocation as well as to provide uniform server interfaces.

4.5.1. The Server Interface

Servers are ordinarily located through a name server that holds a *public* port through which any process can communicate with the server. To obtain service, a customer process first sends his signature to the public port and requests the allocation of a *private* port. The server allocates a private port for the customer and associates it with the signature. Requests for service are always sent to the private port, which serves to identify the customer.

In some situations, a customer sends a request to a server on behalf of some other customer. It is then desirable to specify a signature with each request, so that a server can use its customer's signatures when acting on their behalf.

A customer must also indicate if he is willing to handle exceptions raised when an account is overdrawn. If so, the server will set the *Notify* parameter to false for withdrawals, and send an error message to the user if an overdraft occurs. Otherwise, *Notify* is set to true, and higher-level exception handling mechanisms are invoked if an overdraft occurs. This will be described below.

The Banker itself is a server, and it adheres to these guidelines for server interfaces. The resources managed by the Banker are customer types and resource types. The operations *CreateResourceType*, *CreateAccount*, and *CreateDependent* all have a *Notify* parameter to specify how to handle overdrawn accounts, though this parameter was omitted from our presentation for clarity.

Furthermore, since the Banker is a server, it can use itself recursively to protect against malicious customers who, for example, might attempt to attack the system by allocating many resource types. A resource type is considered a resource itself, and when a server requests

the Banker to allocate a new resource type, the Banker consults the server's accounts to see if the server is authorized to create a new resource type.

4.5.2. Server Actions

Before a server allocates an accounted resource, it performs a withdraw operation. Ordinarily, the user will not handle overdraft exceptions, so the withdraw operation is invoked with the notify parameter set to true. If *Success* is returned by the Banker, the server continues providing service, but if *OverDraft* is returned, the server suspends all activity associated with that particular customer. No error message is returned to the customer at this point, and the Banker saves the reply port from the withdraw operation. This will be called the *server reply port*. Further actions by the server are discussed below.

Since the notify parameter is true, the Banker will send a notice to some overdraft handler with the following information:

1. The signature of the customer who is overdrawn.
2. A currency list to specify the amount of the overdraft⁷.
3. The server reply port.

When the server deallocates resources, a *Deposit* operation is performed to credit the user's account with the resources that are being freed. With some resources, like CPU time, it does not make sense to credit accounts, and resources are not necessarily conserved. Deposits are ordinarily made only into integer currency type accounts.

4.5.3. The Overdraft Handler

When an overdraft handler receives a notice from the Banker, it can take several actions: it can change the limits on the overdrawn account and ask the server to retry the withdrawal; it can tell the server to refuse the operation; it can deport the customer; or, it can terminate the customer.

⁷The currency list may have more than one currency value if the overdraft is the result of a *SetLimits* operation.

4.5.3.1. Change Limits and Retry

For the first option, the overdraft handler can adjust the account limits by issuing a *SetLimits* operation. The overdraft handler must know the overdrawn customer's parent's signature to do this, so normally, the handler must be the parent. A *Retry* message is then sent to the server reply port. The customer signature is included in this message to identify the customer to the server. When the server receives the message, it repeats the *Withdraw* operation. The overdraft handler may at this time also send a warning or other information directly to the customer.

4.5.3.2. Service Termination

For the second option, in which the handler tells the server to deny further service, a *Refuse* message is sent to the server reply port, containing the customer's signature. The server aborts the service for that customer and returns an appropriate error message to the customer. Notice that only one service is terminated and that this action has the effect of returning the error handling task to the customer.

4.5.3.3. Deportation

The third option, deportation, is transparent to the guest. To deport a customer, a *DeportRequest* message is sent to each of a guest's servers. The message contains a port to which servers send state information associated with the guest. (Deportation is described in Section 5.5.)

4.5.3.4. Customer Termination

The final option is to terminate the entire customer, that is, all services as opposed to some one service. This is accomplished by sending an *AbortRequest* message to each of a guest's servers. Unlike service termination, there is no need for the server to send an error message to the customer, since the customer process will also be terminated. (Termination is described more fully in Section 5.6.)

One could also terminate a customer by killing the associated processes only, since servers must be able to recover from customer deaths anyway. The use of the Banker is cleaner and may be more efficient since servers are located directly rather than through propagation of process death notices.

Furthermore, if a guest gives its server connections to a process on a remote machine, then killing all known guest processes may have no effect on servers. To illustrate this possibility, consider a guest that has requested and obtained a resource, for example access to a mail server. Before the guest is terminated, it sends a port to the mail server over the network to an accomplice on a remote machine. The guest has effectively "laundered" his request so that mail service can no longer be revoked, because killing the guest will still leave the accomplice in possession of a mail server port. However, by using the Banker, the mail server will receive an *AbortRequest* message, and service for the guest's accomplice will be denied. In summary, the Banker allows the Butler to locate and recover all resources allocated on behalf of the guest.

4.5.4. Implicit Service Requests

Processes implicitly invoke the operating system kernel as a server to provide address space and processing time. The kernel interface must be augmented with a call to set the signature on which withdrawals are made. This call must also include a notify parameter to specify how exception handling should be performed. These changes allow the kernel to charge customers for CPU time, virtual memory, and other resources managed by the kernel. A computationally intensive server can charge CPU time to its customer by telling the kernel to use the customer's signature.

4.6. Related Work

The Sue system [Sevcik 72, Atwood 72] uses a mechanism, also called the "Banker", to prevent processes from using more than their allotted resources, and to record billing information on disk storage. The Sue system Banker implements only one level of indirect sponsors (corresponding to our dependent accounts) and is not intended to provide the capability-like protection offered by our Banker. The latter form of protection is provided in the Sue system via capabilities, which contain an optional numeric field for restricting the number of times the capability is used to allocate a resource.

Janson's thesis [Janson 76] describes another quota system used to limit allocation of disk pages and to perform accounting. Quotas are associated with subtrees of a hierarchical file system (directories are interior nodes and files are leaves of the subtrees). This organization is somewhat like the subaccount mechanism in our Banker, in that a user can create a subdirectory (dependent account) and apply a quota (limit).

4.7. Summary

The Banker is useful for a number of reasons:

1. The Banker provides accounting services for other servers. This simplifies the servers, and allows them to share common operations. In addition, it provides an *identification* service in that it maps signatures to accounts. A user does not necessarily need to authenticate himself to each server, since his signature serves as a capability.
2. Identities maintained by the Banker are abstract. The Banker does not associate resources with any specific object such as a process, (human) user, or console as is frequently done in current systems. Thus, the association of resources to objects can be flexibly determined.
3. The Banker has many of the advantages of a capability-based protection system. Signatures are analogous to capabilities, but the operations on signatures are an extension to the normal operations provided on capabilities. Typically, capabilities carry a small set of boolean values indicating rights. Signatures effectively carry accounts that can be large sets of values that are not necessarily boolean. The capability-like aspect of signatures allows users to pass subsets of their rights on to subsystems by creating dependents. Users can provide their own exception handlers to be invoked if a subsystem tries to overdraw an account. Therefore, policy is determined entirely outside the Banker, which simply provides mechanisms for accounting.
4. The Banker contains all of the data structures necessary to map identities to resources. Thus, it is possible, given a signature, to find all of the resources that have been allocated for it. This is useful for recovering resources from a guest.

The next chapter will show *How* Butlers negotiate to determine what rights a guest should have. These rights are then used to establish accounts in the Banker for the guest.

AD-A126 003

RESOURCE SHARING IN A NETWORK OF PERSONAL COMPUTERS(U)
CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER
SCIENCE R B DANNENBERG DEC 82 CMU-CS-82-152

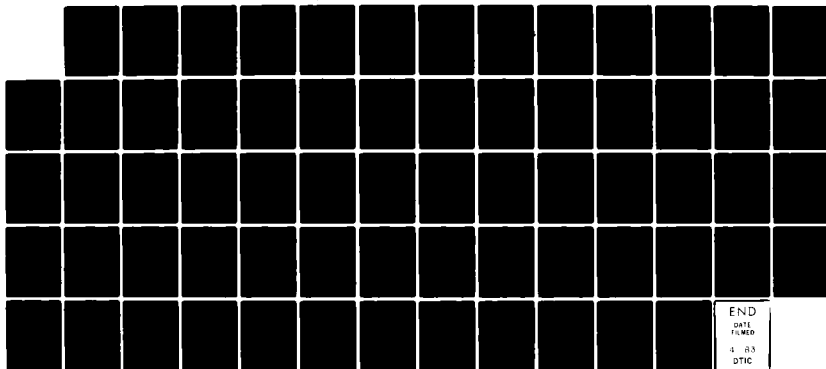
1/2

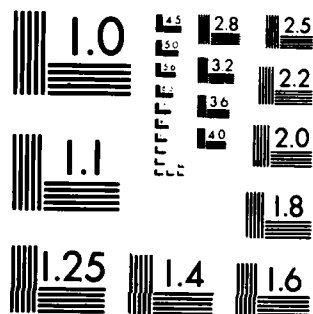
UNCLASSIFIED

F33615-81-K-1539

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

Chapter 5

Negotiation and Revocation

An important function of Butlers is to negotiate the terms under which resource sharing will take place. The agent uses negotiation to inform a potential host of the resources required by the client, and the host uses negotiation to establish limits to the amount of resources a guest can use. It is during negotiation that the host consults current policy and creates capabilities that can be used by the guest to obtain resources. Thus, negotiation includes the process of authorizing a client to use resources on a remote machine.

The primary purpose of negotiation is to insure that the guest has the resources it needs to run successfully to completion. Therefore, the host implicitly agrees not to revoke any resources or rights that are granted to the client during negotiation, except under unusual circumstances. When resources must be revoked, or when a guest exhausts the available resources, some form of recovery action must take place. The Butler design includes three forms of revocation to facilitate the development of distributed programs that can recover from revocation. These are *warning*, *deportation*, and *termination*.

The first section below describes data structures used by Butlers to represent resources and to specify operations. The following section presents the protocol for negotiation between an agent and a host Butler. The host uses a policy database to determine what resources to grant to a guest, and a simple implementation is described. Negotiation is also concerned with how rights are revoked, and the end of this chapter is devoted to how this is accomplished.

5.1. Configuration Specification

Before a remote operation can be invoked, there must be a specification of the requested operation. The agent uses this specification to make a request to a host, and the specification contains at least the name of the operation and the resources required. In general, the agent's request will contain a configuration specification that names a number of system components and specifies their interconnection. In a message-based system like Spice, the configuration specification will name a number of processes that are to be interconnected by ports.

The design of a useful representation for configurations is beyond the scope of this dissertation. Such a design would necessarily consider issues of naming, constraint satisfaction, and human factors. The problem of configuration specification also arises in the context of software development, and many of the issues are considered in Coopriders's thesis [Coopriders 79]. Future systems must address and solve this problem, and the solution should be uniformly applied to software development, the execution of software systems, and the invocation of remote operations.

For the purposes of this dissertation and for preliminary use in the Spice system, a simple representation of configuration has been designed. Our goal is to demonstrate only a representation that is sufficiently powerful to handle most requests, so we will not explore the area of configuration representation in depth. Fortunately, the configuration specification is only loosely coupled to the negotiation process, so the Butler can be easily modified to accommodate changes in the representation of configurations.

A configuration specification must be capable of describing several aspects or dimensions of a configuration. The most important aspect is the operation or service desired. For simplicity, we will assume that the operation is implemented by a server process. The second aspect is the need to pass parameters to the server that implements the operation. Third, resource requirements must be specified, and finally, it may be necessary to specify additional subsystems that are necessary to perform the requested operation. We will address these four aspects below.

5.1.1. Server Specification

The server that provides the desired operation is named by a character string, which is translated by the Butler into a port that represents the server. The translation is facilitated by a local name server, which translates names into ports. If no port corresponds to the name, then the name server may have information that instructs the Butler how to instantiate the appropriate server. In the simplest case, the name server returns a file name that is used to locate code for the server. In the general case, a complete configuration specification is returned.

If we assume that the name server returns a file name, then the Butler creates the desired server by loading and executing the file. The server then sends a port to its parent, the Butler, and the Butler enters the port into the name server for use in future requests. The result is that the string name is mapped to a port name, and the appropriate server process listens for messages sent to that port.

To allow arbitrary application programs to be invoked, one of the possible servers interprets standard user-level operating system commands. To invoke an application program that is not a server, the client invokes the command interpreter server and then gives it the command to execute a program. The server provides the proper environment for the program, loads the program, and executes it.

5.1.2. Parameter Passing

The second problem is the provision of parameter-passing mechanisms in the configuration specification. This problem is solved by splitting invocation into two steps. In the first step, a server is located and a port is returned. In the second step, the desired operation is invoked by sending a request, including parameters, to that port. An alternative would be to include parameters with the configuration specification and to have the Butler send these parameters without interpretation to the server port. This alternative is rejected because it rules out the possibility of invoking a series of operations or engaging in a dialog with the server.

5.1.3. Resource Specification

In Section 2.3.2.2, we developed a representation for resource requests. The important characteristics of this representation are that it associates a value with each of many resources, and the representation includes a specification of how revocation should be handled. A problem with the representation, however, is that the resource types are fixed. This would be acceptable if all machine resources of interest could be identified at an early stage in the system design, but this is not a reasonable assumption. In fact, many resources are likely to be abstract; that is, defined by a server but not corresponding exactly to any physical resource. Therefore, a more flexible representation for resources is necessary. The representation we will choose for resources is the *CurrencyList* type defined in Section 4.2.4. *CurrencyList* replaces the previous definition of *BasicRights*, so type declarations for *BasicRights* and *GuestRights* are:

```
type BasicRights is new CurrencyList;

type GuestRights(Warning: Boolean) is
  record
    InitialRights: BasicRights;
    Deport: Boolean;
    case Warning is
      when True =>
        WarningRights: BasicRights;
      when False => null;
    end case;
  end record;
```

5.1.4. Environment Specification

The fourth problem of configuration specification is to find a suitable representation for the environment in which a program should execute. The solution to this problem is dependent upon how the system of interest represents environments. I will describe one approach which is based on the use of the Spice Environment Manager (SEM) [Ball 82]. The SEM is a specialized database manager that defines the environment of a process. An environment consists of a set of name/value/type triples and possibly a reference to a parent environment (the environment database is tree structured). The SEM is thus a general-purpose mechanism for exchanging values and parameters with a process. A port is associated with each environment, and environments are always accessed via the corresponding port. Each process is given an environment port when the process is created, and the process queries the SEM to read or modify its environment.

For the purposes of configuration specification, we will take a somewhat limited view of what is included in the environment. Our main goal is to make sure the environment includes the resources that are necessary to complete the requested operation; for example, if a compilation requires a file server, the environment must include a port through which the compiler can access the file server. The environment will consist therefore only of server ports that provide access to resources on the host machine.

The servers that are to be included in the environment can be specified in exactly the same manner as the requested operation: a string is used to name the desired server. As before, the Butler either locates a server port through a name server, or it creates the appropriate server by translating the name of the server to a file that contains code for the server. The environment specification must contain a name for each server to be included in the environment. We can now write type definitions for environment specifications:

```
type EnvironmentComponent is
  record
    ServerName: String;
    EnvName: String;
  end record;
```

```
type EnvironmentSpec is array (Integer range <>) of EnvironmentComponent;
```

The environmental aspect of a configuration is specified by a value of type *EnvironmentSpec* which is an array of *EnvironmentComponent* records. Each record names a server (which the Butler will map into a port), and provides a name to associate with the port in the environment. To build an environment, the Butler obtains ports corresponding to each server and enters a name/value/type triple into an environment in the environment manager.

5.1.5. Server Ports

I have purposely omitted several details about the translation of server names to ports. The translation process as described could give the same server port to several guests; however, it is often desirable for a server to allocate a port for each guest rather than receive all service requests on the same port. If a server allocates a separate port for each guest, then revocation is simplified. The use of separate ports also simplifies the authentication of service requests.

5.1.5.1. Public and Private Ports

A server port is typically entered in a name server so that other processes can communicate with the server. This port is called the server's *public* port; access to the public port does not convey any rights to service. To obtain service, a customer sends authorization information to the public port and the server responds by allocating a new *private* port which is associated with the customer. Send rights for this port are then returned to the customer. Requests for service are directed to private ports rather than to the public one.

This organization has several advantages. The use of private ports allows the server to identify the source of a request by the private port on which the request is received. The server does not need to authenticate the incoming request, because only one guest was given the private port, and ports cannot be forged. This is another instance of using ports as capabilities. If notification is sent to the receiver when all send rights for a port have been deleted, as is the case with Accent IPC, then the server can deallocate resources associated with a private port when the customer disappears, even if the customer forgets to explicitly close his connection. Another advantage is that the server can revoke service by destroying a private port, or service can be transferred to another server by moving receive rights for the private port.

5.1.5.2. Name-to-Port Translation

Our translation algorithm must be modified to deal with private ports. The host Butler establishes an account for the guest and locates server public ports as described earlier. For each public port, the Butler sends a message with the guest's signature to obtain a private port. This message must be understood by all servers so that the Butler does not have to have special knowledge of each server and its corresponding message interface. As each server returns a private port, the port is entered into the environment for use by the guest. The name associated with the port in the environment is taken from the *EnvName* field of the *EnvironmentComponent* record that specified the server name.

5.1.6. Representing Configurations

A configuration specification must name an operation, describe an environment, and enumerate required resources. A possible representation for a configuration specification is:

```
type ConfigurationSpec is
  record
    Server: String;
    Environment: EnvironmentSpec;
    Resources: GuestRights;
  end record;
```

The *ConfigurationSpec* provides all of the information required to instantiate a remote operation except for parameters. Once a port to the appropriate server is available, the client invokes one or more operations by sending the operation name and parameters in a message to that port.

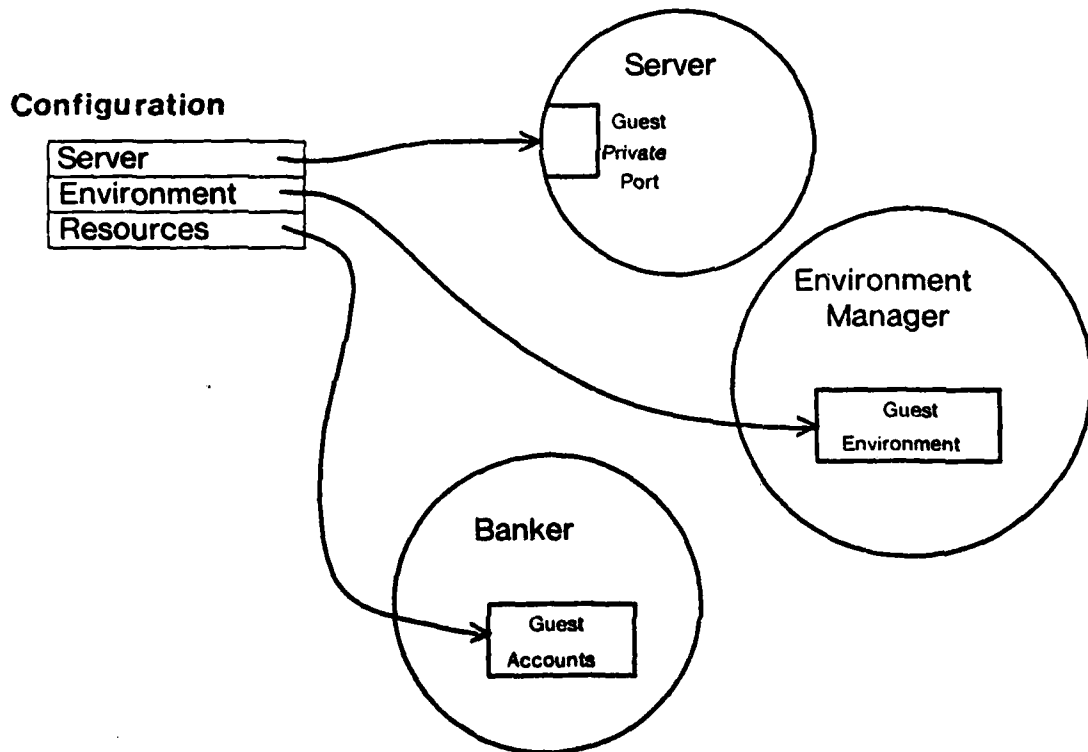


Figure 5-1: A configuration representation.

It is the job of the host Butler to build a *ConfigurationSpec* by creating a configuration.

Figure 5-1 illustrates a guest configuration. Access to the instantiation of a configuration is represented by the following type:

```
type Configuration is  
  record  
    Server: Port;  
    Environment: Port;  
    Resources: Signature;  
  end record;
```

The three fields of the *Configuration* type correspond to instantiations of the fields of the *ConfigurationSpec* type (defined on page 89). The Butler translates a string into a server port as described earlier. The Butler interprets the *EnvironmentSpec* to obtain an environment in the Spice Environment Manager. This environment is represented by the second field of the *Configuration* record, which is a port. The *Resources* field in the *ConfigurationSpec* tells what Banker accounts are necessary for the guest, and the signature for these accounts becomes the last field of type *Configuration*.

In summary, the host Butler accepts a specification for a configuration (type *ConfigurationSpec*). Assuming the configuration is authorized, the Butler instantiates the configuration and builds a configuration representation of type *Configuration*. The representation is returned to the agent Butler, and can be thought of as a capability list, since it contains ports that can be used as tickets to obtain resources and service.

5.2. Negotiation Protocols

So far, we have seen how configurations are specified, and how configuration specifications are translated into configuration representations. We will now turn to the protocols used to invoke a remote operation. Three processes are involved: the client, the agent, and the host. Figure 5-2 diagrams the flow of negotiation messages between these processes. Note that this is a slight elaboration of Figure 2-2. As in the description of the Banker interface, messages will be described as procedure and function calls. Appendix A describes the notation used. The client starts the protocol by making a call on the agent Butler:

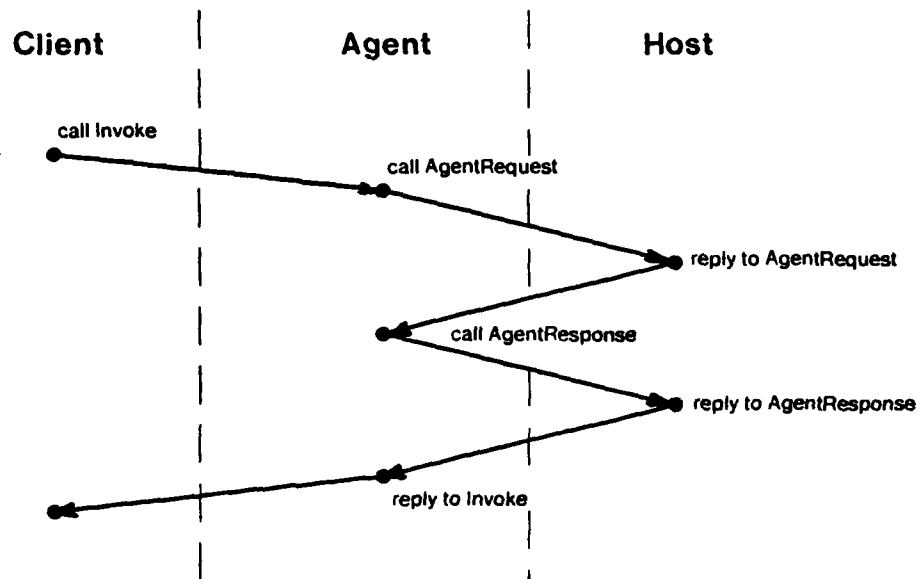


Figure 5-2: Negotiation messages.

5.2.1. Invoke (Client to Agent)

```

function Invoke(Request:      in out ConfigurationSpec;
                  MinRequest:  in   ConfigurationSpec;
                  Where:       in   HostList;
                  TimeLimit:   in   Time;
                  CASPort:     in   Port;
                  WarningPort: in   Port;
                  Guest:       out  Configuration;
                  GuestControl: out  Port;
                  Remoteld:    out  GroupId)
return GeneralReturn;
  
```

The first parameter, *Request*, is a specification of the desired configuration. The parameter *MinRequest* specifies the minimum amount of resources that are adequate to perform the requested operation. *MinRequest* should not include any resources or environment components that are not included in *Request*. The next parameter, *Where*, specifies a list of host Butler names from which resources should be solicited. If the list is empty, the agent Butler searches without constraints. In either case, a central name server is used to find host Butlers. To avoid long searches for relatively quick operations, the parameter *TimeLimit* tells the agent how long to search for resources. The Butler will need to prove the authority of the

client to the host. The *CASPort* parameter is the client's login port (possibly with restrictions applied); the host can use this port to determine the client's authority from the CAS. The *WarningPort* parameter is the destination for revocation warning messages, and *DeportPort* is the destination for deportation messages. Both the *WarningPort* and *DeportPort* parameters may be null, indicating that the Butler should provide default handlers for warning or deportation messages.

Several values are returned by the call. The actual list of resources granted to the client are returned in *Request*. The configuration constructed by the host is returned in *Guest*. The host associates a port with the requested operation, and this port is returned in the parameter *GuestControl*. The client can obtain status information and control the guest via the host Butler by sending messages to this port, and the requests that can be handled by this port are described below. The parameter *Remoteld* is the unique user identifier of the remote Butler. This identifies the person responsible for security at the remote machine.

The value returned by *Invoke* is one of the following:

<i>Success</i>	A host was found that granted at least <i>MinRequest</i> resources. The parameters <i>Request</i> , <i>Guest</i> , <i>GuestControl</i> , and <i>Remoteld</i> are returned.
<i>NoHost</i>	No host was found to provide at least <i>MinRequest</i> resources.
<i>Timeout</i>	The agent was unable to satisfy the request within the time limit.

5.2.2. AgentRequest (Agent to Host)

Upon receiving an *Invoke* message, the agent locates a host and issues the following call:

```
function AgentRequest(AuthHint: in GroupIdList;
                     Request: in ConfigurationSpec;
                     HostPort: out Port;
                     Offer: out ConfigurationSpec)
    return GeneralReturn;
```

The parameter, *AuthHint* is the group membership list of the client, and is derived from the client's CAS port in one of two ways. In most cases the Butler will have handled the login of the client to the CAS and will have obtained and cached the client's group membership list at that time. Otherwise, the Butler will have to send a request to the CAS to get the list. The purpose of *AuthHint* is to avoid going through the authentication protocol for each potential host. If a host is willing to meet the agent's request, the protocol is performed to verify the correctness of the hint.

The next parameter, *Request*, is the configuration requested by the client's *Invoke* message. The host Butler queries its policy database to determine what, if any, of the requested resources can be granted, assuming that *AuthHint* correctly represents the authorization group membership of the client.

The configuration specification returned in *Offer* tells the agent what resources are available. The possible values of *AgentRequest* are:

<i>Success</i>	The host Butler has offered to invoke an operation. <i>HostPort</i> and <i>Offer</i> are returned.
<i>Refuse</i>	The host will not authorize the client to use any resources. <i>HostPort</i> and <i>Offer</i> contain null values.

At the point where an offer is made to the agent, an authenticated connection is required so that the agent can determine who made the offer and the host can determine whom the offer is made. To avoid the cost of the authentication protocol, we will use a technique described in the last paragraph of Section 3.6.3; that is, ports are exchanged with security, and will be authenticated at a later time. The agent's half of the connection is the reply port for the *AgentRequest* message,⁸ corresponding to port P_A in Figure 3-3. The host's half of the connection is returned in the *HostPort* parameter, which corresponds to port P_B in Figure 3-3.

The agent compares the offer, if any, to the client's *MinRequest* to determine whether or not to accept. The agent then replies to the host with the *AgentResponse* call described below.

5.2.3. AgentResponse (Agent to Host)

```
function AgentResponse(Accept:      in Boolean;
                      AgentKey:    in Key;
                      ClientKey:   in Key;
                      DeportationPort: in Port;
                      WarningPort: in Port;
                      Config:      out Configuration)
return GeneralReturn;
```

To reject an offer, the agent sets *Accept* to false. This terminates negotiation with the host. The agent then returns from the client's *Invoke* request with the value *NoHost*, or seeks

⁸ As described in Appendix A, every call message has an implicit reply port.

another host. The next two parameters, *AgentKey* and *ClientKey*, are used in the authentication protocol. The last two parameters, *DeportationPort* and *WarningPort*, are used to handle revocation.

To accept an offer, *Accept* is set to true. The host then performs two instances of the authentication protocol. The first uses *AgentKey* and authenticates the existing channel between the agent and the host. The second uses *ClientKey* and is necessary to obtain a port on which to reply to the *AgentResponse* message. After authentication, the host creates an account, builds an environment, and finds or creates a server as specified in the *Offer* that was returned to the agent through the *AgentRequest* call. Configuration construction is described above in Section 5.1, and the resulting *Configuration* is returned as the last parameter, *Config*. The reply message, containing *Config*, is sent to the port obtained from the authentication protocol with the agent using *ClientKey*.⁹

Meanwhile, the agent Butler has been participating in the two instances of the authentication protocol with the host. When the protocols complete, the agent must be sure that the identities obtained from each protocol instance are consistent. The agent then waits until it receives a reply to its *AgentResponse* message. Next, the agent sends a reply to the *Invoke* operation called by the client.

5.2.4. Additional Operations

The principal operations that are required for negotiation have now been described. Several other operations are provided by Butlers.

⁹It may seem instead that the configuration should be sent to the agent's port, which was authenticated using *AgentKey*. To see how this could lead to trouble, consider the case where a malicious Butler, called X, has just received an *AgentResponse* message from an agent called A. Now, X assumes the role of agent and requests some third Butler (we will call this one V, for victim) to perform some action. Butler X will lie to V, claiming that his client has the same identity as that of A's client. Butler X will of course want to verify the client's identity before giving any rights away, but X cleverly passes along the *ClientKey* obtained from A. Butler V can now successfully perform the authentication protocol using this "stolen" *ClientKey*. Now, if V returns a configuration to X, X will have tricked V into providing a configuration for which X was not authorized. On the other hand, if V returns the configuration to the port obtained using *ClientKey*, the configuration will be sent to A, not X. Butler A can then determine that X violated the negotiation protocol (by giving away *ClientKey*), and the configuration at V can be aborted. This scenario also illustrates that the configuration should not perform any irreversible operations until the client makes a request over an authenticated connection. Furthermore, the agent should verify that the authentication protocols using *AgentKey* and *ClientKey* are in fact performed by the same host.

5.2.4.1. Deportation

The host expects a port from the agent on which deportation messages can be sent. This port will be supplied by the agent unless the client furnishes his own in the *Invoke* operation. The port can be changed to redirect deportation with the following message:

procedure *SetDeportHandler*(*DeportationPort*: in *Port*);

which is issued over the *GuestControl* port.

The client can force deportation using the following call, also issued to the *GuestControl* port:

procedure *DeportStart*;

5.2.4.2. Renegotiation

Additional resources can be requested by sending a *NewRequest* message to the *GuestControl* port:

function *NewRequest*(*Environment*: in out *EnvironmentSpec*;
 Resources: in out *GuestRights*)
 return *GeneralReturn*;

The *Environment* parameter specifies additional environment components desired by the client, and *Resources* specifies additions to the current account limits set for the guest.

To perform this operation, the host consults the policy database and the current account limits. If allowed by the policy, the guest is granted the additional rights.

The possible return values are:

Success	The host has provided the guest with the requested rights.
Offer	The host cannot grant all of the requested rights. The <i>Environment</i> and <i>Resource</i> parameters are set to the maximum subset of the requested rights allowed by the current policy. The client can reissue <i>NewRequest</i> with the reduced rights if desired.
Refuse	No additional rights can be granted.

5.2.4.3. Status

To monitor the progress of a guest, the client can request a status report with the following message:

function *GetStatus*(*Report: out Status*) **return** *GeneralReturn*;

Again, the function is invoked by sending a message to the *GuestControl* port. The only reply (besides *Error*) is:

Success *Report* contains the requested status information.

5.2.5. Alternative Designs

The negotiation protocol as described above places the goal of stable expectations over the goal of optimal resource allocation. In fact, no attempt is made to find the "best" site for a given guest. Instead, the protocol is designed to find only a host that will provide the necessary resources. Negotiation also tells the host what resources the guest expects. I feel that this protocol is appropriate for a system in which control is distributed and nodes are autonomous personal computers.

Other protocols have been developed that assume greater cooperation between machines and less machine autonomy. Examples are a protocol developed for DCS [Farber 73] and the Contract Net Protocol [Smith 80]. Both of these systems broadcast resource requests for which potential hosts submit bids.

There are many minor differences between DCS, Contract Net, and Butler protocols. The DCS and Butler protocols require the potential host to respond immediately to requests, but in Contract Nets, only idle nodes submit bids, and the node may wait for further task announcements before making a bid. Butler offers, and Contract Net bids are binding, but DCS bids can be retracted. Contract Net nodes can overallocate resources by bidding on several tasks, but Butlers are expected to honor their offers. Only Butlers acknowledge rejected offers (bids). Most of these statements about Contract Nets are not invariably true, since there are a number of optimized protocols for handling special cases.

Each of these systems was designed for a slightly different purpose. DCS was intended to be a reliable, shared, general-purpose computer system. Contract Nets was meant to be a distributed problem-solving system, and the Butler is intended to facilitate sharing in a

network of autonomous nodes. Very little experience has been obtained in this area, so experimentation with the Butler and other protocols is warranted.

5.3. Policy Database

This section describes a simple policy database to be used by a machine owner to specify what resources are to be made available to guests. When the host Butler receives an *AgentRequest* message, it consults the policy database to determine what resources to offer the agent. Recall from Section 2.4 that the policy database must implement the function:

Policy: GroupIdList x Locality x Occupancy → Rights

Data structures that are used in the policy database and its interface are first described. Next, the operations implemented by the database are described. We will *not* design or describe a suitable user interface to create and manage the database.

5.3.1. Data Structures

Rights are represented as currency lists. The database is conceptually an unordered set of 4-tuples that have the following components:

1. A *user* field of type *GroupId*.
2. A *locality* field, which contains one of (*Local*, *Remote*, *DoNotCare*).
3. An *occupancy* field, which contains one of (*Occupant*, *Guest*, *DoNotCare*).
4. A *rights* field, which is of type *CurrencyList*.

5.3.2. Butler Access

The Butler accesses the database with the call:

```
function GetPolicy(IdList: in   GroupIdList;
                  Loc:   in   Locality;
                  Occ:   in   Occupancy;
                  Rights: out CurrencyList)
return GeneralReturn;
```

where *IdList* is an array of user and group ID's, *Loc* is one of (*Local*, *Remote*), and *Occ* is one of (*Occupant*, *Guest*). Multiple ID's are given because the user may be a member of a number of groups, each of which can have different rights. In Spice, group membership can be

determined from a central authorization server, which maps group ID's into lists of group ID's by computing the reflexive transitive closure of the group membership relation. Other systems support similar notions, and we have seen how this information can be obtained as part of the authentication protocol. The value returned from the database manager is the union of all rights obtainable by any entity identified in *IdList*.

The database manager saves the port used to reply to *GetPolicy*. If any changes occur to the database, a notification is sent to that port using the following message:

procedure *ChangeNotice*;

This procedure is a warning that the database has changed, and previous policy information may no longer be valid. A reduction of rights can thus be detected by the Butler without polling the database.

5.3.3. Owner Access

Authorization to change the policy data is stored in the database itself. A machine owner can change the database using the following operations:

```
function ExtendRights(ID:      in GroupId;
                      Loc:      in Locality;
                      Occ:      in Occupancy;
                      Rights: in CurrencyList)
return GeneralReturn;
```

```
function DeleteRights(ID:  in GroupId;
                      Loc:  in Locality)
                      Occ: in Occupancy;
return GeneralReturn;
```

ExtendRights adds the rights given by *Rights* to the tuple whose first three fields are *ID*, *Loc*, and *Occ*. A new tuple is created if none currently exists. The possible return values are:

Success	The operation was performed.
Unauthorized	The user is not authorized to change the database.

DeleteRights removes any tuples that match *ID*, *Loc*, and *Occ*. Locality and occupancy values of *DoNotCare* may be specified to match any locality or occupancy value. The possible return values are:

Success	The operation was performed.
----------------	------------------------------

Unauthorized The user is not authorized to change the database.

NotFound No matching tuples were found.

5.3.4. Discussion

Certainly, a more elaborate policy database could be designed. For example, one might wish to make policy vary as a function of time or the current processing load. In this section, I have only attempted to demonstrate a possible design and illustrate how the Butler interfaces to it. Because the policy mechanisms have been cleanly separated from the mechanisms provided by the Butler, one could give users their choice of several styles of policy database without changing the rest of the Butler implementation.

5.4. Warning

To provide the greatest amount of flexibility in handling revocation, the client or guest can provide an application-specific handler that is notified by the host when the guest exceeds its resource limits. When the host receives an overdraft message from the Banker, it constructs a warning message containing the amount of resources that the guest has overdrawn and the current resource limits. The warning is sent to the port specified in the *AgentResponse* message (see Section 5.2.3). Note that a warning is only sent if the *Warning* flag in *GuestRights* is true. (*GuestRights*, defined on page 86, is a component of a *ConfigurationSpec*, defined on page 89, which is the type of the *Request* parameter to *Invoke*, defined on page 91).

The overdraft condition can be either the result of a server making a withdrawal or the host restricting account limits to reflect a policy change. The warning message is:

procedure *Warning*(*Overdraft*: in *CurrencyList*;
 Limit: in *CurrencyList*);

When the host sends the warning message, the guest's current balance is augmented by the guest's warning rights. This gives the guest some additional resources to use in its recovery from revocation.

As an example of the use of warnings, consider a computationally expensive program that can be checkpointed rapidly. If this program were to be run remotely, it could be designed so

that upon receiving a warning message, the program sends its checkpoint information to a supervising process on the user's machine.

The ability of the guest or client to supply application-dependent handlers is both an advantage and a disadvantage. While warnings give the greatest flexibility to the client, they also require that recovery handlers be programmed for each application. Furthermore, the host has to trust the guest to heed the warning. In the next section, we discuss another revocation mechanism, called deportation, that is transparent to the guest, and does not require cooperation between the guest and host.

5.5. Deportation

The goal of deportation is to remove a guest from a machine, called the *source* machine, and to reconstruct the guest at another *target* machine in a way that is transparent to the guest. By "guest", I mean a configuration that is instantiated by the Butler on behalf of some client, as described previously. Deportation is only performed if (1) the guest attempts to overdraw an account, (2) the *Deport* flag was set in the *GuestRights* returned from the *Invoke* operation, and (3) either *Warning* was not requested or the guest's warning rights were exhausted. A guest contains state, uses resources, and includes not only processes but connections to servers and state information maintained by servers. To deport a guest, the guest's state must be encoded and separated from physical resources so that the resources may be reclaimed. For the purpose of revoking resource rights, it is important to be able to identify the resources that are used by a given configuration, including all of the servers that have allocated resources on behalf of the guest. A few operations are implemented by the Banker to facilitate the location of these servers, and will be described in this section.

Deportation can be separated into several steps. The first step is to locate the state information that represents the guest. Second, the guest's state must be translated to a form that is suitable to be transferred to another machine. The last step is the reconstruction of the guest on a remote machine. These steps are described in detail below.

5.5.1. State Location

All state associated with a guest is managed by server processes. As explained in Section 5.1.5.2, service is granted to a guest by sending a standard message with the guest's signature to a server's public port. The message requests that the server allocate a private port for the guest. After allocating the port, the server sends a *PleaseNotify* message to the Banker. The message contains send rights for the private port and instructs the Banker to remember the port and associate it with the supplied signature. As we shall see, the Banker can be instructed later to send these ports to the Butler so that the Butler can locate all of a guest's state. The structure of the *PleaseNotify* message is given by:

```
function PleaseNotify(PrivatePort: in Port;  
                        Guest:      in Signature)  
  return GeneralReturn;
```

The possible return values are:

Success The operation is complete.

To deport a guest, a message must be sent to each server containing part of the guest's state. The servers of interest are those that previously registered a port using the *PleaseNotify* message. The registered ports are retrieved in a two-step operation as follows.

In the first step, the guest's accounts and dependent accounts in the Banker are locked to prevent synchronization errors between other servers that share the guest's signature. In the second step, all ports registered with *PleaseNotify* and associated with the guest's signature or dependents are retrieved from the Banker. The following function performs these operations:

```
type PrivatePortRecord is  
  record  
    Sig: Signature;  
    PrivatePort: Port;  
  end record;
```

```
type PrivatePortList is array (Integer: range <>) of PrivatePortRecord;
```

```
function LockAndRetrieve(Guest:   in Signature;  
                        Parent: in Signature;  
                        PortList: out PrivatePortList)  
  return GeneralReturn;
```

Guest is the signature of the guest. *Parent* is the signature of the guest's parent. To limit

the power of malicious servers, only the parent has the right to retrieve a guest's private server ports from the Banker, and *Parent* serves to authorize this operation¹⁰. If the call succeeds, *PortList* is returned containing a list of pairs of ports. The first element of each pair is a signature, and the second is a private port that some server associated with the signature. These are exactly the parameters necessary to invoke a *DeportRequest* operation.

To perform *LockAndRetrieve*, the Banker first sets a lock on the guest's account and on the transitive closure of the guest's dependent accounts. The lock prevents any further operations using the guest's signature or any of its dependent's signatures. The Banker then assembles the ports associated with these signatures (as a result of *PleaseNotify* operations), and returns them to the Butler. The possible return values are:

<i>Success</i>	The operation succeeded, and <i>PortList</i> was returned.
<i>Unauthorized</i>	The <i>Parent</i> parameter is not the signature of <i>Guest's</i> parent. The operation was not performed.

If a process attempts any operation provided by the Banker, such as *Withdraw* or *PleaseNotify*, after a signature has been locked, then the Banker does not perform the operation and returns *Locked*. A server should assume that a deportation is in progress and wait for a *DeportRequest* message.

5.5.2. Deport Request

We have seen how a guest's state may be located by having each server register private ports with the Banker. Servers themselves are responsible for encoding the guest's state when requested by a *DeportRequest* message.

A server must always be prepared to accept a deport request message on a private port, and the same form of message is sent to all servers:

procedure *DeportRequest*(*Guest*: in Signature; *WhereTo*: in Port)

The parameter *Guest* provides the signature with which the private port was registered in an

¹⁰ However, if the parent and one of its dependents both give their signatures to a malicious server, the server could, by trial and error, construct a valid *LockAndRetrieve* message, so the *Parent* parameter does not provide a truly secure authentication of the parent. To achieve complete security, the Banker could, for each dependent, issue the parent a special port on which to invoke *LockAndRetrieve*. I feel that this extra effort is not worth the additional security, given the number of other ways a malicious server can wreak havoc upon a guest.

earlier *PleaseNotify* message. If a server allocates a new private port with each guest, the parameter is not strictly necessary, but it is conceivable that a server might use the same port for more than one client, in which case the signature is necessary.

The second parameter, *WhereTo*, specifies a port to which the guest's state is to be sent. Upon receiving the *DeportRequest*, the server encodes the guest's state and sends it as a message to this port. The encoding operation is described below.

Notice that any process with the guest's signature and send rights to the guest's private port can request a server to deport a guest. In particular, the guest itself can arrange to deport all or part of itself. The more interesting case, however, is deportation without any specific preparation or active participation by the guest itself. This capability allows a host Butler to deport a guest without detailed knowledge of the guest's configuration, and it also allows deportation of untrusted configurations. Since in general, the host knows nothing about the guest, it is not sufficient to send a "please deport thyself" message to the guest. What if the message is ignored? If the Butler can locate the guest's servers, then a deportation can be ordered directly without relying on a guest's cooperation. Of course, the Butler still has to rely on servers to carry out the deportation.

5.5.3. State Encoding and Decoding

When a server receives a *DeportRequest* it assembles a server-dependent *deportation* message, which encodes the current server state associated with the guest. This message will be sent to a *deportation server* which will reconstruct an instantiation of the guest on another machine. The job of the deportation server is to look at a header section of each message to determine what kind of server generated the message. Then, a corresponding server is located or constructed at the target machine. The message is forwarded to this server, which is expected to be able to interpret the message and reconstruct the guest's state. The deportation server should not need information specific to servers because that would require modifications to the deportation server whenever a new server is created.

A deportation message has two parts. The first part is a header that names a server. The server specification is another form of configuration specification. For the purposes of this dissertation, a simple character string will suffice. The second part of the message is everything that follows the header. The format of this part is server-dependent, and is

determined by the designer of the server. The only requirement is that the format must allow a server to encode the guest's state and allow the receiving server to decode the state in a manner that is transparent to the guest.

5.5.3.1. A Server Model

To illustrate the way deportation messages are constructed, we will examine a simple abstract model of a server process, and see how a guest's state can be represented. From the viewpoint of the guest, the server behaves as indicated in Figure 5-3.

```

loop
    receive M from Pg
    (Sg, r) ← f(Sg, M)
    send r
end loop

```

Figure 5-3: Abstract state-transition server model.

In this model, P_g is the guest's private server port, and S_g is the state of the guest maintained by the server. When the guest sends a message M to the server, the server evaluates $f(S_g, M)$ to determine the next state and a set of output messages r . The server is thus described by a finite-state machine, where inputs and outputs are messages. This will be called the *abstract state-transition model*.

It is important to note that the model is only an abstraction of the server. The implementation of the server might look more like Figure 5-4.

```

0)   loop
1)       receive M
2)       g ← M.FromPort
3)       (C, Sg, r) ← f(C, Sg, M)
4)       send r
5)   end loop

```

Figure 5-4: Model of a server implementation.

In this figure, the fact that the server deals with multiple guests is made explicit. After receiving a message (line 1), the server determines the private port from which it came (line 2). The service function, f , is used to change the state of the corresponding guest (line 3) as before, but additional state, C , is also changed. C represents state that is shared by all guests. Although this information is hidden from the abstraction seen by guests (as in Figure 5-3), it may be necessary for an implementation. For example, resources may be allocated from a common pool described by C .

Given an implementation like that in Figure 5-4, it is not obvious how to extract a guest's state. Obviously, the state includes S_g and P_g , but C must also be examined and probably modified when a guest is deported. C must be left in a consistent state to avoid adversely affecting service to the remaining guests. Another problem is that the state S_g may not be meaningful outside of the context of a particular server instance: for example, it may contain absolute addresses as pointers, etc. Even if S_g were meaningful outside of the server, it might be very difficult for the target server to check the consistency of S_g , and importing random state information could be very dangerous to the target server.

The solution to these problems is to use the abstract state-transition model rather than the actual server implementation to specify the guest's state. The deporting server translates the guest's real or implementation state into a representation that corresponds closely to the guest's abstract state. The point of this encoding is to remove implementation-specific information from the guest's state and to encode a high-level representation of the guest that can be easily and safely imported by another server.

When a guest is imported, the importing server must be very careful to check all state information for consistency. The information is not trustworthy, and the state is more complicated than typical server requests, which must also be carefully screened. One way of simplifying the problem is to decompose the import operation into a number of lower-level operations that can be invoked through the ordinary server interface. This is only possible if none of the guest's abstract state is hidden by the server interface, which is a still higher level of abstraction [Parnas 75]. If each lower-level operation is checked by the server, then the server is protected against forged state information.

A study that is relevant to this recommendation on server interface design is by Kapur and Mandayam [Kapur 80]. In this study, three kinds of operation sets for data abstractions are defined: *expressively incomplete*, *expressively complete*, and *expressively rich*. The first kind of set does not have sufficient operations to translate the value of the abstract data into some other form, say integers, and back again. An expressively complete operation set permits the translation, but not necessarily in a practical sense. Finally, an expressively rich operation set allows the values of the type to be translated to other types in a practical sense. As an example, in the current Accent kernel, to discover the pages of an address space that represent valid memory, a process must inspect each of the 2^{23} possible pages. The interface is complete, but not rich because it is possible but not practical to translate the address space into another form (for instance, network messages).

5.5.4. Supervision

To make sure that a guest is deported expediently, all guest state should be sent from servers to the host. The host can then forward deportation messages to the deportation port furnished by the agent. The host should place a time limit on the delivery of messages so that they are not just transferred to the network server where they continue to use local virtual memory.

The host must also be careful if network servers implement a "lazy evaluation" message-passing protocol, where data is not sent until it is actually needed at the destination. Some cooperation with the network server is again necessary to prevent deportation messages from occupying local virtual address space for an arbitrary time.

5.5.5. Example

To illustrate these concepts, we will consider how the Spice Environment Manager (SEM) can deport guest environments. First, the abstract state-transition model of the SEM and a possible implementation of the model are described. Then we will discuss how guests are deported and imported. For this example, a simplified version of the SEM is considered to avoid describing irrelevant details.

5.5.5.1. Model of the Spice Environment Manager

The state S_g of a guest g consists of a set of name/type/value triples. Messages from the user can add a triple to the environment, delete a triple, store a value into the triple with a given name, and retrieve the value of a triple with a given name. We will not concern ourselves here with parent environments or other SEM operations.

5.5.5.2. Implementation

A possible implementation of the SEM uses a binary tree to represent each environment. Each node holds a triple and two pointers to other nodes. A hash table is used to map private port names to a binary tree representing an environment.

5.5.5.3. Deportation

To deport a guest, the SEM must generate a representation of the guest's abstract state, which is a private port and a set of triples. A representation for the triples can be generated easily by walking the binary tree and writing a list of suitably encoded triples into the deportation message. Notice that no information about the binary tree structure is deported, nor is any information relating to the hash table deported. These implementation structures have nothing to do with the guest's abstract state.

5.5.5.4. Importation

When the target server receives the deportation message, it first enters the private port into its hash table and initializes the hash table entry to the empty environment. Then, for each triple in the deportation message, the server invokes its own *InsertTriple* operation. This operation checks the triple for such things as legal name syntax and a match between the type specification and value. The operation also checks with the Banker to make sure the guest environment is not so large as to exceed an account limit.

By decomposing importation into a number of *InsertTriple* operation, we can have a greater confidence that the server state cannot be corrupted by a forged deportation message. Since *InsertTriple* is already part of the SEM, the extra implementation effort to perform importation is minimized.

5.5.6. Discussion

The goal of deportation is to provide a mechanism for recovering from the revocation of resources in a manner that is transparent to the guest. Deportation is an operation that may not always succeed. Some servers do not fit the abstract state-transition model, and state information may not always be machine-independent. In the example above, the problem of hierarchical environments was purposefully avoided. If several guests share access to a top-level environment, it may not be possible to deport a guest transparently. Another example is the deportation of a guest with a file open on the local disk. If the file is simply a cached copy of a file known globally, there is no problem, but if the file is meant to be kept only locally, the guest cannot be deported from the local file server. There is no general solution to these and similar problems. Server designers should try to make deportation succeed in most cases, and provide a clear specification of the conditions that will cause deportation to fail.

Finally, it should be remembered that deportation is only one of several available recovery techniques. The guest is not forced to use it, and a more general technique based on revocation handlers provided by the guest is available.

5.6. Guest Termination

If warning and deportation were not requested, or if they fail, the host can revoke resources by terminating the guest. The location of the guest's state is determined exactly as in deportation, using *LockAndRetrieve*. The host then sends the following message to each server containing some of the guest's state:

procedure *AbortRequest*(Guest: in *Signature*);

The message is sent to the guest's private server port. When a server receives this message, it first verifies that the signature matches the one presented when the private port was allocated. The server then deallocates the private port and the associated guest's state information.

5.7. Summary

In this chapter, we have focused on the process of borrowing and giving up shared resources. It is here that protection, autonomy, and sharing strongly interact.

The representation of resource requirements is essential to the task of negotiation, and a simple scheme of configuration specification was described for this purpose. The proposed representation does not solve all of the problems of configuration specifications, however, and this is an area that requires further investigation.

Negotiation is performed in order to establish an agreement between the agent and host Butlers on the amount of shared resources and the terms under which resources may be revoked. Authentication is used so that if the agreement is violated, the exploited party can identify the offender.

Negotiation also establishes what actions will be taken when a guest exhausts its resources or when a policy change dictates that a guest cannot continue executing with its present resources. The programmer has several options that simplify the task of writing reliable software. A *warning* is the most flexible option, but it requires application-specific handlers to

be written. *Deportation* is a higher-level option that is application-independent, but perhaps less efficient and certainly less flexible. *Termination* is simple, but may make recovery in a distributed program more difficult. These options can be used hierarchically, that is, the first may be attempted, and if it fails, the second is tried, and so on.

Chapter 6

A User Interface

One of the goals of the Butler design is the support of load leveling. If a user is using most of the computation power of his machine while other machines are idle, it may be desirable to run some tasks on a remote machine. To make this form of load sharing practical, we must design a system that allows a user, working at the operating system command level, to easily run a program on a remote machine. Furthermore, the environment on the remote machine should be sufficiently close to the local environment that no special coding or recompilation is necessary to run a program remotely.

This chapter addresses the problem of designing a user interface to the Butler that allows the user to execute programs remotely. Even without this user interface, the Butler would be valuable for use with distributed parallel programs and programs for exchanging information, as described in Chapter 1. However, support for load leveling is important, and the requirements of transparency must be considered carefully.

6.1. Introduction

Two approaches can be taken in describing a user interface for the Butler. The first is to describe various aspects of the interface in general terms to avoid specifics that might not be applicable to every system. The second approach is to present a more detailed description of an interface as it would appear in a specific system. I have chosen this second approach because it affords the opportunity to look at problems encountered in a real system. Actually, so much background information would be required to describe the Spice operating system environment, that the user interface cannot be described in great detail, and only the main technical problems are presented. The goal of this chapter is to outline the implementation of a user interface and to convince the reader that a successful interface can be built.

Before describing how to execute programs remotely, we must first see how programs are executed locally. Then, we will see how an equivalent execution environment can be constructed on a remote machine with the help of the Butler.

6.2. Local Program Execution

In Spice, a program is instantiated by loading a file containing the code for the program and forking the loader process to create a separate thread of control and a separate address space. A set of ports is passed to the new program, including an environment port and a signature. The environment port is a capability that authorizes access to an environment maintained by the Spice Environment Manager (SEM) [Ball 82]. The signature is a capability for the resources available to the new program, as represented in the Banker.

6.2.1. The Environment Manager

In previous chapters, the SEM has been described as a database of name/type/value triples. In addition to the database component, the SEM contains another component called the *forms interpreter*. The purpose of the forms interpreter is to provide a standard user interface to application programs and to service an interactive display window.

The forms interpreter uses a template called a *form* to specify the user interface of a program. The form describes the components that are expected to be in the environment database, and specifies how database components are to be displayed. For example, an integer in the environment might be displayed as text, as a bar graph, or some other graphical display. The form also specifies which components of the database can be updated by the user. Some components may only be written by the application program, some only by the user, and some may be written by both. The form includes the name of the application program to which it corresponds.

Programs are normally invoked by sending a form to the forms interpreter. The forms interpreter creates an environment, obtains and initializes a display window, loads the application program, and starts its execution. The normal mode of interaction with the display is through the forms interpreter. When a program needs an input parameter, a request is sent to the environment port for the parameter. If the parameter is not yet defined, the forms interpreter is notified. The forms interpreter then prompts the user for the required value. The

value is entered by the user and stored into the environment, and the SEM returns the value to the application.

6.2.1.1. Environment Hierarchy

The environment database consists of hierarchically organized environments. Recall that values are accessed by name; if a program tries to read the value corresponding to a name from its environment and there is no entry for that name, then the environment manager looks in the parent environment. If the name is still not found, the search proceeds to the next parent, and so on. This allows a parent process to make its environment available to one or more subsystems without actually copying values into subenvironments.

6.2.2. The Terminal Manager

The terminal manager is a server process that manages human input and output devices such as the keyboard, pointing device, and display. A process communicates with the terminal manager through Accent IPC messages.

The terminal manager interface is designed to hide many of the specific characteristics of the physical display and keyboard. For example, display coordinates are described in terms of an abstract coordinate space rather than in terms of physical screen pixels. The terminal manager interface has another advantage in that it also hides the location of the physical display and keyboard. Since all input and output operations are invoked via messages, a program and its terminal can easily be located on separate machines.

In Spice, an interface process called Canvas already fills most of the requirements for the terminal manager. The major component not present is a window manager that can allocate screen space at the request of a process. In Canvas, a process can only subdivide windows that it already has. On the other hand, the window manager treats the screen as a resource and uses the Banker to determine whether a process is allowed to allocate a window and receive input from the keyboard or other devices.

6.2.3. Interconnection

The typical configuration of an application program, its environment, and the window manager is illustrated in Figure 6-1. Initially, the forms interpreter has a port through which it can send messages to the terminal manager, and the shell or system command interpreter has a port through which it can send messages to the forms interpreter.

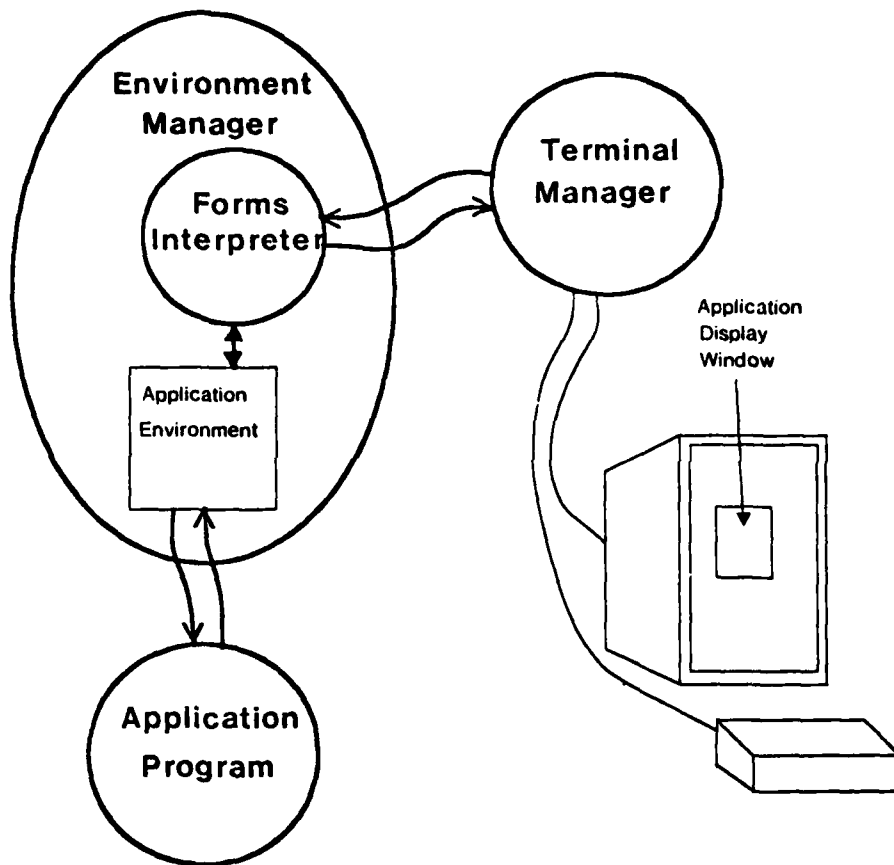


Figure 6-1: Port connections associated with an application program.

The shell invokes an application by sending a form (or perhaps the name of a file that contains a form) to the forms interpreter. The shell also sends a signature to provide access to resources for the application program. The forms interpreter reads the form to obtain a specification for the required display window, and then sends a message to the terminal manager to allocate a window. The forms interpreter initializes the window and an

environment according to the form, and starts the execution of the application program. A port to the environment is passed to the application program when it is started.

6.3. Remote Program Execution

Our problem is now to show how this sort of configuration can be invoked from a local machine and instantiated remotely. The local terminal manager should be used so that the user can interact with the remote program in the same way he would interact if the program were executed locally. All other components of the program configuration should be instantiated on the remote machine.

Remote program execution is supported by a local process called the *client server*, which interfaces the user to the agent Butler. The client server uses the agent Butler to locate resources at a remote machine. The client server then communicates with a forms interpreter at the remote machine to invoke the desired application program.

6.3.1. The Client Server

As indicated in earlier chapters, the interface to the Butler is not trivial, and the casual user will need some assistance even to run simple applications on a remote machine. The job of the client server is to insulate the user from most of the details by supplying default values where possible, and a user-oriented command interface to obtain the remaining information. The client server is invoked by the shell and uses the environment and terminal managers just like any other application to interact with the user. (Alternatively, the client server functions could be incorporated into the shell, if desired.)

The client server gets information from the user and constructs a configuration specification that names the forms interpreter as the required server. Assuming the Butler can find a suitable host and return a configuration to the client server, the client server must now send local terminal manager port to the remote forms interpreter. To authorize use of the local terminal, the client server also sends a signature to the forms interpreter. Now the client server can start the application requested by the user by sending a form to the remote forms interpreter.

6.3.2. The Forms Interpreter

The forms interpreter can now be described more comprehensively. The forms interpreter is a server whose job is to create an environment and an interactive display window for a program, to start the program, and to serve as an interface between the window and the environment. Like other servers, the forms interpreter normally has a public port entered in the name server so that the Butler and other programs can locate it and obtain private ports.

Ordinarily, the forms interpreter will use the local terminal manager to interact with the user, but an alternate terminal manager can be selected by sending its port and a signature authorizing its use to the forms interpreter. In this way, a client server can divert a remote forms interpreter's terminal input and output to the local machine.

The forms interpreter handles several message types that invoke programs. The standard message contains the name of a file that contains the form to be used. The form in turn contains the *file name of the program to be executed*. In some cases, the form and possibly the program will not be available directly as files. For example, the desired form and program may be on a remote machine. In these cases, the form and program may be included in the message.

6.3.3. Environment Protection

It was mentioned earlier that the hierarchical structure of environments helps a parent process to provide an environment to subsystems, since the environment manager will automatically look in a parent environment if the required information is not found in the current environment. This can be a problem if the parent does not want the subsystem to have access to its data. A simple solution is to allow the parent to disable all access by a subsystem when the subenvironment is created. This might be appropriate for guest processes that might otherwise make random probes for values such as ports and other capabilities.

More general approaches are possible. Entries in an environment could be individually marked as accessible or non-accessible to environments at lower levels. Access lists could also be associated with entries, or the environment manager could even be replaced by a general purpose database. Experience is needed to determine whether any of these more elaborate schemes are necessary. The simple approach of the previous paragraph is probably adequate.

6.3.4. Example

To illustrate how remote invocation is performed, we will consider a user who wishes to perform a remote compilation. To simplify our presentation, we will assume that a typescript interface to the forms interpreter is used, although a menu- or editor-based interface could also be used. The user first tells his forms interpreter to run the client server program. The client server prompts the user for the name of the program to be executed, and allows the user to specify the resources he needs and his choice of host machines. For this example, we assume that the user has simple requirements and takes the defaults provided by the client server. The display might look like the following (commands typed by our user are in *italics*):

```
>clientserver
ClientServer V. 0.0
>>runremote
command? compile demo.ada
host? any
resources [<CR> for defaults]?
```

If the client server is invoked as a parallel task, the user can perform other activities at his local machine while waiting for the compilation to finish.

The client server constructs an *Invoke* message (described in Section 5.2.1) that names the forms interpreter as the desired server. The message is sent to the local agent Butler which locates a host and returns a configuration to the client server.

At this point, the remote forms interpreter has no access to the local machine, so it cannot communicate with the local terminal manager. The client server uses the client's signature to obtain a private port to the local terminal manager and sends this port to the remote environment manager.

The client server can now send a form and a program (or their file names) to the forms interpreter. The signature provided in the guest's configuration accompanies the message to authorize the resources that will be used by the compiler. The resulting interconnection of client server, terminal manager, forms interpreter, and compiler appears in Figure 6-2.

The client server implements two other operations:

<i>status</i>	Print the current status of the remote task.
<i>abort</i>	Halt the execution of the remote task.

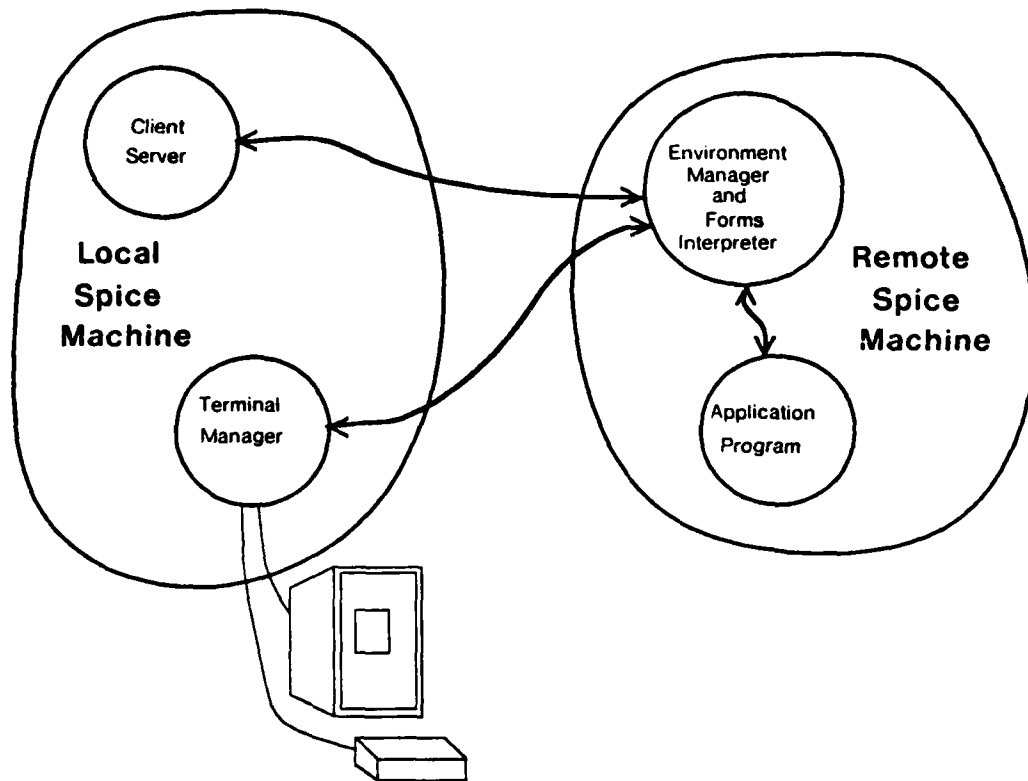


Figure 6-2: Executing a remote application program.

6.4. Summary

A convenient user interface is essential to make resource sharing for load leveling practical. A program like the client server can perform most of the work of invoking a remote operation in simple cases.

It is important to structure the system so that ordinary programs can be executed remotely without reprogramming. In particular, a message-based IPC facility can be used to hide the location of a process from its terminal and other resources.

The environment manager must be designed carefully to support the client server. It must be possible for the environment manager to direct input and output to several terminal managers to handle multiple user locations. Also, environments must be protected from the users of subenvironments.

Chapter 7

Evaluation and Conclusion

We begin this chapter with a preliminary evaluation of the design presented in the previous chapters. Included in this evaluation is a description of a partial implementation of the Butler. The second part of this chapter summarizes what we have learned, presents some concluding remarks, and describes some important areas for future research.

7.1. Evaluation

The design presented in the previous chapters has been developed with the goal of supporting resource sharing in an environment consisting of autonomous personal machines, several trusted central facilities, and a high-speed interconnecting network. The principal concern and overriding philosophy has been to provide as much protection and autonomous control of machines as possible, even in cases where simpler control or less protection would lead to a more efficient system. On the other hand, the design does strive for efficiency whenever possible; for example, the Butler is generally used only at the beginning of a remote operation. Thus, an evaluation of this design should consider its level of support for sharing, support for autonomy, and efficiency. In addition, an evaluation should assess the system's ease of use, since it is only valuable if people use it. In the following sections, we will consider each of these four properties in turn.

7.1.1. Support for Sharing

The Butler design is fairly conservative in its support for sharing. It allows great generality in the sorts of operations it can invoke, but this is achieved by avoiding any special purpose functions, such as atomic transactions, which might in fact be quite useful. Consequently, the Butler must be augmented by programs that actually perform the desired operations. This leads to a more modular system, but it may make the programmer work harder to construct a

resource-sharing application. An exception to this philosophy is the Butler's support for *recovery from revocation*, which is provided because the Butler's recovery mechanisms would be much more difficult to implement at the level of the application program.

On the other hand, there is little in the Butler design that forms an obstacle to the construction of resource-sharing applications. Such applications are free to use any existing server, or to create new processes. Furthermore, once a configuration is instantiated, the Butler places no restrictions on the way in which the application's components communicate or invoke internal operations. A variety of potential applications was discussed in Chapter 2. The Butler design also supports resource sharing by protecting machine owners from malicious borrowers. Without this protection, owners would be reluctant to allow shared access to their machines. To summarize, the Butler supports sharing by providing primitives to invoke remote programs or servers, and by protecting resource sharers. While this support seems to be adequate, experience with a full implementation and a community of users will be required to determine if the Butler supports sharing in a practical sense.

7.1.2. Support for Autonomy

The Butler design takes an aggressive approach to the support of autonomy, and much of the design is concerned with mechanisms that enable a machine owner to control and protect access to his machine. Several components of the design work together to provide substantial protection for resource sharers, and in the following paragraphs we will evaluate these components in terms of the kind of protection provided, and the extent to which autonomy is supported.

The first component we will consider is secure network messages. Using the protocol presented in Chapter 3, any two network servers can obtain a secure channel for communication. The only requirement is the presence of a trusted intermediary, in this case, the Central Authorization Server (CAS). In particular, the network servers need not trust any other system components such as the network itself, other network servers, or inter-network gateways. Autonomy is supported by the facts that an independent and secure channel is established between each pair of communicating machines, and that machines only need to trust the physically secure CAS.

The second component is a protocol for establishing a secure and authenticated

connection based on ports and IPC messages. This protocol is based on the security of underlying network messages, and has similar characteristics with respect to autonomy, except we must remember that ports are implemented by (and depend upon the security of) the operating system. A user should therefore be careful not to give his rights (represented by his CAS port or his password) to an untrustworthy operating system.

The third protection component is the Banker, which is used to keep track of resource utilization after a guest has been instantiated. An independent Banker process executes on each machine, so the Banker certainly supports the concept of autonomy. Furthermore, the accounts that enable a configuration to obtain resources are determined by local machine policy that is controlled by the machine owner. The Banker is a part of the local operating system, and the security of the Banker does not depend upon any processes outside of the operation system, again in support of autonomy.

Finally, we come to the Butler itself. Like the Banker, a separate Butler process executes in each machine, and the host Butler does not depend upon the correct functioning of any other Butler to provide protection for its owner. The agent Butler, however, cannot guarantee that a guest will not be exploited by a malicious host, but it can at least make the identity of any host it deals with known to the user. In this way, social mechanisms can be used to discourage maliciousness. In either case, the user must only trust in his local operating system and in the security of the CAS. The policy that controls sharing on a machine is determined completely by its owner, and no global operating system or higher authority is used to control sharing; therefore, machines are highly autonomous.

7.1.3. Efficiency

One must be cautious in evaluating the efficiency of the Butler design, since the important question is not "how fast can a user invoke operation X on a remote machine?", but "is the overhead of the Butler justified by its support for security, autonomy, and programmer convenience?". The latter question can be answered only through experience with an implementation; but on the other hand, it would be foolish to implement a Butler without carefully examining the issues of efficiency. In this section, we will consider the work required to perform the basic functions of the Butler and Banker. These functions are negotiation, banking, and deportation.

7.1.3.1. Negotiation

From the negotiation protocol described in Chapter 5, it can be seen that a fairly large number of messages is issued and that little computation is involved in negotiation, with the possible exception of accessing the policy database. We will assume that the cost of negotiation is dominated by inter-machine messages, since it seems likely that this will be the case.

The cost of sending IPC messages between machines in turn depends on how these messages are mapped onto underlying network-level messages, which must include acknowledgements, retransmissions, flow control messages, and port status information. To avoid considering these unknowns, which are largely implementation-dependent, we will simply look at the number of inter-machine messages at the IPC level, and count the number of IPC ports delivered.¹¹

Starting with the *AgentRequest* message (see Section 5.2), and ending with the reply to that message, there are 12 inter-machine IPC messages. This figure includes 8 messages required for two instances of the connection protocol of Section 3.6. In addition, there are 15 ports sent between machines. The actual cost of these operations is not presently known, but it should be easy to measure the cost of the connection protocol when the Spice Authentication Server is implemented.

Once negotiation is complete, it is necessary for the client to invoke an operation using the server, signature, and environment ports returned by the host Butler. This may actually cost more than negotiation, since programs and data may need to be transferred over the network to the remote machine. The actual cost depends upon the nature of the remote operation and on the amount of data that must be transferred. In order to get a better idea of these costs for real applications, I have implemented a prototype Butler and used it to obtain estimates of the cost of executing an existing distributed signal-processing program. The results of this experiment will be presented in Section 7.2.

¹¹ Sending an IPC port implies extra work for the network server; see Section 3.5.

7.1.3.2. Banking

We will now consider the cost of the Banker and its protocols. The use of the Banker implies some initial cost to set up accounts and to notify the Banker of the servers associated with a signature. There is also an incremental cost for each withdraw operation.

To create a customer account, the Butler performs a *CreateDependent* operation and obtains a signature. The Banker need only generate the signature and record the corresponding limits and handler as supplied by the Butler, so this is a simple operation. Then, for each server in the guest configuration, the Butler sends a message containing the signature. Each server responds by allocating a private port for the guest and then sending a *PleaseNotify* message to the Banker. When the Banker replies, the server sends the guest's private port to the Butler. These are simple operations, so the cost will probably be dominated by the cost of sending the messages. There are two messages (for the *CreateDependent* operation) plus four per server to obtain and register private ports. The cost to invoke a trivial operation with one argument via (two) messages using the current implementations of the Accent kernel and the Matchmaker server-interface generator is about 11ms, including run-time type checking of the argument.

Once the initial guest configuration is built, there is the additional cost of withdraw operations whenever a server allocates resources for the guest. Computationally, a withdrawal requires finding an entry in a sparse table (indexed by signature and resourceid), and performing several addition and compare operations. This is simple enough that most of the cost will be in the two messages required to invoke the operation and receive a reply. For cases where frequent withdrawals are too costly, it is possible for the server to reduce this overhead by withdrawing resources in larger blocks and keeping track of fine-grain resource usage locally. To summarize, the cost of using the Banker is roughly the cost of four messages per server connection, plus two messages for each withdrawal operation. All of these are intra-machine messages, so no network overhead is involved.

7.1.3.3. Deportation

We will now consider the cost associated with deportation. Most of the cost will probably be that of transferring state information, but let us first look at the number of intra-machine messages required.

The host Butler begins by sending a *LockAndRetrieve* message to the Banker to obtain a list of the guest's private ports. Next, the host sends a *DeportRequest* message to each private port and waits for a message from each server containing the guest's state. Thus, to retrieve the guest's state, it takes two messages to retrieve the private ports, plus two more for each server. The state must then be transferred to the target machine and decoded. The decoding process takes at least one message to forward the state, and two messages for the server to perform a *PleaseNotify* operating at the new Banker. In addition to intra-machine messages, deportation includes the process of encoding a guest's state into messages, transferring the state over the network, and decoding the state at a new server. The cost of these operations depends upon how state is represented internally and externally, the cost of consistency checking and transformation from one representation to another, and the total amount of state information.

The cost of deportation was the subject of another experiment, in which process deportation was implemented and measured. The results were used to determine the cost of encoding a process into messages and then decoding the messages back into an executing process, and the cost of deporting a guest from one machine to another has been estimated. This experiment is described below in Section 7.2.

7.1.4. Ease of Use

The final area in which we will examine the Butler design is ease of use. The Butler design supports both application programs and interactive use, although emphasis is placed on the former. Application programs are supported in a number of ways. First, the agent Butler has a simple interface that hides the necessary negotiation and authentication protocols to obtain resources. The Butler design does not specify the representation of configuration specifications (although a simple one is suggested), so a representation that is compatible with the rest of the system can be used, leading to a reduction in the number of system data types and concepts. The Butler also supports application programs by providing several mechanisms for revocation. The options allow the application to use its own handlers or to rely entirely on the Butler to revoke rights. Finally, it should be noted that the use of messages and ports to hide process location is a considerable help toward writing distributed programs, although this feature is not specific to the Butler design.

The basic Butler can be augmented with servers to support direct user invocation of remote

programs and interaction with them as described in Chapter 6. The user interface allows programs to be executed remotely without any changes or recompilation, and this is facilitated by the transparency of port location. The interface can be designed to look like the standard system command interpreter for ease of use.

7.1.5. Evaluation Summary

We have examined the Butler design from the standpoints of support for sharing, autonomy, efficiency, and ease of use. To support sharing, the Butler provides primitives to borrow resources and to handle revocation. The Butler gives clients direct access to servers so as not to overly constrain the way resource-sharing programs are constructed, and the Butler also encourages sharing with a well-protected environment. Autonomy is supported by avoiding any dependence or trust in other personal machines for protection or resource allocation decisions. From the standpoint of efficiency, we examined negotiation, banking, and deportation. While the initial negotiation and banking operations are fairly expensive, the only cost incurred while executing a guest is in Banker withdrawal operations, but this cost can be held to an acceptable level by withdrawing resources in suitably large blocks. The cost of deportation is likely to be dominated by the cost of encoding, transferring, and decoding state, not by the messages required to locate and notify servers. Finally, the Butler supports application programs by providing a simple invocation interface, and a flexible means of handling revocation. The location transparency provided by messages and ports also simplifies the programming task and makes it possible to build a powerful user interface.

In order to evaluate the design more completely, several experiments could be performed without undertaking a full implementation. One such experiment is to measure the process-to-process message delay, where the message must go through network servers as described in Chapter 3. I do not know of any existing systems that can do this very efficiently, but work is currently underway to implement an efficient network server for the CMU Spice system, and message delays of 10 to 20ms are anticipated. Some experimentation with various implementations of passing ports over a network is also warranted, since this appears to be a frequent and potentially costly operation. In addition, the cost of authentication can be measured when the Spice Authentication Server is implemented. Experiments to provide estimates of the performance of invocation and deportation have already been performed, and are described in the following section.

7.2. A Prototype Butler

Several aspects of the design presented in this thesis have been implemented in the form of a prototype Butler. The primary goal of this implementation is to get some idea of the overhead involved in remote invocation and deportation in the context of a network of personal computers. It should be emphasized that the results obtained cannot be expected to completely validate or invalidate the design; however, the results can serve to indicate problem areas or to demonstrate that certain aspects of the design are feasible.

Although the prototype is not intended to provide the full functionality of the Butler, it does support a real distributed processing application, to be described below. The prototype implementation is limited to the areas of remote invocation and deportation, because these are areas where performance is an important factor, and because these areas seemed feasible to investigate, given the current state of implementation of the Spice system.

7.2.1. Methodology

The prototype was constructed to support an existing distributed processing application through the provision of a remote invocation operation and to implement the deportation of processes. In each case, the prototype was instrumented to measure the cost of the operation in terms of actual processing time, and also in more abstract terms to achieve some degree of technology independence in the results.

For remote invocation, the cost can be described in terms of two components: the cost of messages between Butlers to effect the invocation, and the cost of transferring files and data to the remote machine. (Authentication was not implemented.) The cost of messages between Butlers was measured directly in terms of actual execution time, and also in terms of the number and size of messages. To measure the cost of network data transfer, an existing file-transfer program was used since a satisfactory network server is not currently available.

The prototype Butler also implements deportation. The cost of deportation can be expressed in terms of three components: the cost of extracting state from a source process, the cost of transferring state, and the cost of installing state in a target process. Furthermore, the state consists of a port space, an address space, and a micro-interpreter register state. The cost of extracting, transferring, and installing each of these three components was

measured separately, and cost is expressed in terms of both CPU time and the size and number of object moved.

7.2.2. The Application

The application selected is a prototype for a distributed program that tracks object locations in real time, using acoustical data from a set of microphones. The program uses multiple processes to perform Fast Fourier Transform (FFT)¹² operations on several streams of data. In this version, the results are simply plotted; however, the results will be used ultimately to locate the signal source. This program was adapted for Spice by Jon Webb from a program written by David Hornig.

The structure of the program is simple. A process, called *SigGen*, reads signal data and distributes it to a number of processes that are instances of the program *SigProc*. In the original program, each *SigProc* process performs an FFT operation on its signal and plots the result in a local display window. Each process is started manually, and *SigGen* finds the other processes through a name server. The application was originally developed as a collection of processes on a single machine, and would not run without modification on a collection of machines, since no provisions were made to connect the *SigGen* process to *SigProc* processes except through the use of a local name server process.

7.2.3. Structure of the Prototype

The facilities required to adapt the signal-processing program to multiple-machine execution are implemented in the Butler prototype. In fact, the prototype structure allows the signal-processing program to be executed on multiple machines with no modifications whatsoever. In this section, we will look at the structure of the prototype and see how it can support a distributed application program. In the following section, we will describe the deportation facility.

The prototype is implemented as two programs, a host and an agent. When executed, the

¹²The FFT is an operation that transforms a representation of signal strength as a function of time to one of signal strength as a function of frequency. This is useful in a variety of applications involving signal analysis. The time complexity of the FFT algorithm is order exactly $n \log n$, where n is the number of sample points representing the signal in both the time and frequency domains. In the present application, 1024 integer points are transformed.

host registers its name with a name server and waits for a message from some agent. A *RunRemote* message, when sent to the host will cause the host to execute a program. The *RunRemote* message contains a program name, a command line, and a set of ports. These ports are to be passed to the executed program, forming its execution environment (an environment manager as described in Chapter 6 is not yet implemented), so the program can be connected to arbitrary servers as specified by the agent. If any of the ports are null, the host substitutes connections to servers on its own machine.

The host creates a process to execute the specified program, and then sends a reply to the *RunRemote* message containing the *kernel* and *data* ports of the new process. The kernel port can be used to directly manipulate the process; for example, a debugger can use the kernel port to suspend the corresponding process and read or write its memory. The data port is used to send messages to the new process.

The second component of the prototype is a program that serves as an agent. The program includes a simple text command interpreter and provides some of the functions of the client server described in Chapter 6. Some of the user-level commands are now described.

The *FindHost* command searches for a host with a specified name, and a variant of that command will read a list of potential hosts from a file and search for them. Assuming that a host has been found, the *RunRemote* command can be used to send a *RunRemote* message to a specified host. This command prompts the user for the program name and command line to be sent in the message; thus, the agent provides a user interface that is compatible with the existing command interpreter. In addition, the agent allows the user to specify whether the program will receive local or remote file and name server connections. Finally, the agent allocates a local display window and includes the corresponding ports in the *RunRemote* message.

Other commands are provided to suspend or resume a remote process, and to print its status. The command interpreter can also be instructed to read commands from a file, which helps to automate the construction of distributed programs in which several processes must be started on various machines.

From this description, we can see that the Butler prototype supports the signal-processing application in several ways. First, the Butler allows the application to be run on multiple

machines with no modification to the single-machine version. This is possible because the Butler can create processes with port connections to foreign processes in place of the normal port connections to local processes. Also, the command interpreter in the agent can be used to automate the configuration of a distributed computation.¹³ In addition, the Butler allows the distributed program to be configured such that input and output requests are directed to the user's terminal, which may not be at the site that originates the requests.

7.2.4. Deportation

The Butler prototype has also been used to investigate the cost of deporting processes. The basic approach to deportation is to extract the state of a suspended guest process piece by piece, and to insert the state into a newly created and suspended target process. At the end of deportation, the target process will be logically identical to the deported guest, and may be resumed to continue execution.

The state of the guest process consists of three components: the address space, the port space, and a register state.¹⁴ We will look at the deportation of each of these components in turn.

The extraction of address space from a process is simplified by the address space manipulation primitives in the Accent kernel. The operation *ReadProcessMemory* is used to move a block of memory from the guest process to an area in the Butler's address space; because of the internal representation of address spaces, this operation is efficient even for large blocks of addresses that may have "holes" of unallocated virtual memory. Since the Butler's address space is no larger than that of the guest, the address space of the guest must be copied in several chunks. Each chunk is sent as a separate message to the target Butler, which inserts the chunks into the target process using the Accent operation *WriteProcessMemory*.

Ports are extracted from the guest one at a time and sent to the target Butler. The

¹³In a full scale implementation, the command interpreter should be a part of the shell or client server rather than a part of the agent Butler. The important point is that the agent in combination with some form of command language can be used to automate the construction of a distributed configuration.

¹⁴The register state contains the state of the micro-interpretor, including the microprogram counter, hardware expression stack, registers, and microprogram return address stack.

operation *GetPortStatus* is used to search the port name space of the guest to find valid port names. Then, the *ExtractPort* operation is used to move each guest port to the port space of the agent Butler. The agent then uses the *InsertPort* operation to install the ports in the target process's port space.

Finally, register values are read one word at a time using the Accent operation *Examine*. The register values are written into the target process using the operation *Deposit*.

The last operation is to extract the guest's kernel port, causing the guest to be terminated.¹⁵ The kernel port is installed as the kernel port of the target process, and deportation is complete. The agent Butler can then resume the target process.

7.2.5. Invocation Measurements

The Butler prototype was instrumented to measure its performance. The first set of measurements is related to the cost of invoking a program on a remote machine; the second set measures deportation performance, and will be discussed in the next section. The cost of remote invocation is the sum of the Butler execution cost and the cost of transferring local data to a remote machine. The Butler execution times are summarized in Table 7-1; the figures represent the execution time to invoke an instance of SigProc on the local machine using the Butler and also using the standard command interpreter, called *Shell*. In both cases, process creation is performed by sending a message, *PMCreateProcess*, to a process known as the process manager, whose execution time is listed separately. Also listed is the time to load and initialize SigProc.

Butler Invocation		Shell Invocation	
Agent Butler	0.43 s	Shell	0.30 s
Host Butler	0.04		
PMCreateProcess	0.45	PMCreateProcess	0.30
SigProc	<u>4.5</u>	SigProc	<u>3.35</u>
total	5.42	total	3.95

Table 7-1: Execution time to invoke and load SigProc using the Butler and Shell.

The Butler was instrumented to measure total elapsed time as well as execution time. The

¹⁵ No process can be allowed to continue to exist without a kernel port, since a process can only be terminated by sending a message to its kernel port.

total elapsed time (on a single machine) to invoke *RunRemote* and get a reply is 1.9s. Of this, 1.4s are spent invoking *PMCreateProcess*, which includes the creation of a new process, but not the time to load the code for *SigProc*.

The measurements indicate that the invocation time is currently dominated by the time to load a program. The execution time of the Butler (0.47s) is comparable to that of the Shell (0.30s).

The other cost of remote invocation is the cost of sending programs and data from the local machine to a remote one. This time was estimated using the data rate of an existing file transfer program, CFTP, which also runs on the Perq computer. CFTP transfers about 10K bytes per second, so roughly 7 seconds are required to send the 71.2K bytes of code that make up *SigProc*. This assumes that none of the code is available at the remote machine, but in fact, most of the *SigProc* code is standard system code for input, output, interfaces to standard servers, string manipulation, real arithmetic, and debugging. Only about 15K bytes are specific to *SigProc*, so perhaps as little as 1.5s would be required to transfer *SigProc* to a remote machine. This requires that the loader be able to use local copies of code segments whenever possible. Faster transfers may also result when copies of the same program are sent to several machines, since the virtual memory system will act as a cache, reducing the number of secondary storage accesses necessary to read the code segments. Alternatively, if the code segments are on one or more file-server machines, the remote programs can be loaded without placing a load on the local machine.

Once loaded, the execution time of *SigProc* is 6.9s, which is greater than the total transfer time only if the remote machine can load standard code-segment files from the local secondary storage, or if caching increases the data transfer rate. On the other hand, once a *SigProc* process is created, it can be used to analyze many collections of data.

7.2.6. Deportation Measurements

The next set of measurements deals with the cost of deportation. These measurements are designed to indicate under what conditions deportation is an economical recovery technique. As with the invocation measurements, the cost of deportation can be divided into several disjoint components. These are: the cost of extracting the state of a process, the cost of transferring that state, and the cost of installing the state in a target process. The cost can be

further subdivided according to the three components of a process: its address space, its register set, and its port space. In addition, *one process is created and one is destroyed* when a process is deported. Table 7-2 summarizes the direct measurements of the deportation of a SigProc process.

Extract State	
Address Space	4.4 s
Registers	0.4
Port Space	2.7
Install State	
Address Space	2.7
Registers	0.4
Port Space	0.3
Process Creation	1.6
Process Destruction	<u>1.4</u>
total	13.9

Table 7-2: Elapsed time measurements of the deportation of SigProc.

These figures represent elapsed time, including delays for paging. The cost of moving the address space includes the cost of building a virtual memory description tree consisting of about 160 nodes, each representing a "chunk" of contiguous valid addresses. No memory is physically copied, however. There were also 35 registers and 26 ports to be moved. The time to extract ports (2.7s) is somewhat misleading because it includes the cost of searching the entire 8-bit port address space to locate only 26 ports; a kernel operation that returned a list of valid port names would reduce this time to about 0.3s. The times for moving ports and registers could be further reduced by implementing kernel operations that extract and install collections rather than single items, thus reducing the overhead of many procedure calls, messages, and context switches.

The cost of transferring process state between machines can be estimated as before. The amount of state is approximated by the code size (71.2K bytes) plus the global data size (9K bytes), or about 80K bytes. Using our previous estimate of 10K bytes per second transfer rate gives approximately 8 seconds to transfer all of the process state. As before, this time could be reduced if the system could recognize that most or all of the code segments already exist on the target machine.

7.2.7. Discussion

From the measurements of invocation overhead, it can be seen that the Butler mechanism adds relatively little cost to the total cost of loading and executing a program in the current Spice system. These measurements should only be taken as rough estimates; however, since very little effort has been invested in optimizing the system for faster execution.

An important lesson learned from the measurements is that even for a computationally intensive program like SigProc, the code transfer time may dominate the cost of remote execution. On the other hand, several factors tend to reduce the cost of transferring code. First, large amounts of code are commonly used utilities, and are likely to be present at the remote machine. It is therefore important that the code of a program be composed of separately loadable segments, so that locally available code can be combined with code from a remote machine. In Spice, this optimization has been carried even further, so that when pages of code are shared, only one copy is kept in physical memory, resulting in improved virtual memory system performance. Thus, commonly used code segments may even be present in primary memory at the remote machine. The second factor is that code may be available from another machine, such as a file server. This frees the invoking machine's local disk to do other useful work and may reduce the time to transfer data if the file server has higher performance than the file system of the invoking machine. Third, the properties of virtual memory may allow copies of a program to be sent to several machines for little more than the cost of reading the first copy from secondary storage. Finally, it should be noted that a distributed program will have additional primary and secondary storage as well as additional processing resources. In some cases, these resources will improve the performance of a program by more than the amount estimated on the basis of execution time alone.

While many factors will tend to improve performance beyond that which was measured, the use of additional protection mechanisms will tend to degrade performance, and the prototype does not implement any of these mechanisms. The additional protection mechanism overhead was described in Section 7.1.3.2.

From the deportation measurements, we can estimate the total time to move a SigProc process from machine to machine. Let us assume that state extraction and installation proceed in parallel, but that state transfer does not. This gives about 15.5s to deport a process, and there may be additional time to read swapped pages from secondary storage.

We can conclude that in the case of SigProc, it would be more efficient to terminate the remote process and start another one than to deport it. However, the application must be written to handle the termination of instances of SigProc, because simply restarting the entire configuration of SigGen and several instances of SigProc takes well over 15.5s.

7.2.8. What Was Learned

Several lessons were learned in the construction of the prototype Butler. First, the use of multiple processes communicating through messages was found to be a powerful technique for distributed processing applications. This is not really a new lesson (for example, see the report on AMPL [Dannenberg 81]), but it is important nonetheless. In particular, the use of ports to provide a communication path to a server, to function as a capability for service, and to name the client is very useful. For example, processes are manipulated by sending messages to their kernel ports. The kernel port provides a path to the kernel which performs the operation, grants permission to perform the operation, and also indicates to what process the operation should be applied. Similarly, terminal input and output is handled by sending messages. Ports provide a path to the terminal manager, authorize a process to produce output, and indicate on which window output should be displayed. Ports have proved to be a very flexible mechanism for structuring a system of communicating processes.

Another lesson supports the recommendations on server design to facilitate deportation (see Section 5.5.3): whenever possible, the server interface should contain operations to extract and reconstruct the complete abstract state maintained by the server. Accent violated this principle in its original port space manipulation primitives, which were designed to allow debuggers to conveniently tap into a communication path. To support deportation, Accent was extended with three new primitives: *GetPortStatus*, *ExtractPort*, and *InsertPort*. These are simpler and more general than the original operations, which will be removed.

The implementation of the prototype Butler consists of about 2200 lines of code in an extended version of Pascal, and it took between one and two man-months for the author to implement; the code length does not include standard packages that are used for interfacing to the kernel and servers. A Butler that provides authentication, protection, and policy administration would be much larger than the prototype.

From the measurements, we learned that the overhead of the Butler in invoking a remote

program is comparable to the overhead of the current system's command interpreter and that the major source of overhead is in transferring files across the network. It is therefore important that the system take advantage of the fact that a large part of a program may already be present at the remote machine. With this optimization, which is part of the Spice system design, the signal-processing application would profit from distribution. Negotiation and particularly the authentication protocols will add additional overhead; however, if we assume an overhead of 20ms per network message, this overhead is only 240ms, which is small compared to other factors.

Deportation has also been demonstrated, and again, the measured performance indicates that data transfer accounts for most of the deportation time. If code accounts for most of the state of a process, and if deportation can be optimized to take advantage of code that is already available at the target machine, then deportation will cost little more than invocation of the same program. On the other hand, if these constraints are not met, then deportation may only be desirable as a convenience, or when checkpointing or other means of handling revocation would be complicated or expensive.

7.3. Conclusions

This research has developed in roughly three phases. In the first phase, it was recognized that resource sharing is important in a personal computer environment, and the principal problems of sharing were identified. It was discovered that autonomy has a profound effect on the structure of a network of personal computers. Machine owners should be able to control the use of their machines, including how they are shared. Another important consideration, related to autonomy, is that personal machines are not physically protected, which complicates the problem of maintaining security when machines are shared. Our goal was to provide protection equivalent to that of a conventional time-shared system, and to allow as much sharing as machine owners are willing to permit.

The Butler concept was developed to provide a foundation from which these problems could be approached. The Butler has ultimate control over the resources of a machine, and acts on behalf of the machine owner to supervise the borrowing of resources. The Butler concept inherently supports autonomy since a Butler is associated with each machine. In addition, the Butler concept also seemed like a good basis for protection mechanisms since it places a single process in charge of the users of a machine.

In the second phase, the Butler concept was developed into a well-defined operating system component for the support of resource sharing. The Butler maintains a global view of the machine it controls, enabling it to administer policies set by the owner; and because the Butler is a trusted piece of software, it is the logical place to perform authentication protocols which are necessary to identify users of malicious software. The Butler also controls the revocation of resources.

Although originally conceived as a mechanism to run programs on remote machines, the Butler has been generalized as a mechanism to invoke remote operations, where the operation can be performed by an existing server process, a newly instantiated server, a configuration of servers, or a specified program. It is intended that the Butler use whatever representation of configurations is used for other purposes in the system, and it is the flexibility of the configuration specification language that ultimately determines the kinds of operations that the Butler can invoke.

As the requirements of the Butler became more specific, it was realized that the Butler actually should perform two roles. The first role, called the *host*, is concerned with protecting the local machine from malicious users. As a host, the Butler places limits on the resources available to guests, and uses an encryption-based protocol to authenticate the identity of a guest before granting any resources. The second role, called the *agent*, is a server that hides much of the host Butler protocol from clients that desire to invoke remote operations. The agent locates a host, authenticates itself and the client to the host, and negotiates with the host to obtain resources and invoke the desired operation.

In the process of designing the Butler, it became necessary to augment the Butler with an accounting system called the Banker. The Banker is the means by which the policy, as determined by the Butler, is made known to servers. Before granting resources to a process, servers consult the Banker to determine if the process has permission to use the requested resources. Because the Banker is strictly an accounting mechanism, it turns out to be quite simple, but because it is protected by a secure operating system, the Banker is also quite powerful. One of the important properties of the Banker is that any program can become a server, define new resources, and use the accounting mechanisms of the Banker.

The third phase has been the implementation of a prototype Butler. The prototype demonstrates that a Butler-like program can be used to support a distributed application

program. The prototype deportation mechanism has demonstrated the deportation of a process, even when the process is in the midst of executing application-specific microcode. Furthermore, deportation is entirely transparent to the application. Instrumentation of the prototype has shown that the overhead of the Butler to invoke a program on the local machine is comparable to the overhead of using the command interpreter to perform the equivalent operation. The results indicate that the main overhead of invoking a remote program will be the cost of transferring data across the network to the remote machine.

While the chosen application demonstrates the usefulness of the Butler for distributed processing, the Butler is intended to support other styles of resource sharing as well. While the prototype has not been used for data-sharing or load-sharing applications, we sketched how the Butler supports these applications in Chapter 2 and described the user interface necessary for load sharing in Chapter 6.

I conclude that the Butler concept is a useful one, in that it provides solutions to a number of resource sharing problems, and it supports a variety of applications. The Banker is a powerful and general facility for restricting the rights of processes. I have demonstrated the Butler concept with a prototype and provided a preliminary evaluation; however the final evaluation of the Butler must await experience with a more complete implementation.

7.4. Future Directions

I have been careful to use the term "prototype" to describe the present implementation. The next step is to implement a "real" Butler and Banker for the Spice system. The Butler will use the authentication server and name server components of the Spice file system, and will perform negotiation and revocation as described in this dissertation.

It would be interesting to investigate practical methods of protection under different assumptions about the sophistication of users and the value of information maintained on machines. For example, many (if not most) existing personal computer systems provide little or no protection through the encryption of data. In relatively "friendly" environments such as an office or research institute, it may be possible to achieve a suitable level of protection with simpler and more convenient techniques than those described in this dissertation. This might lead to a greater degree of sharing and simpler protocols.

In Chapter 5, a simple representation of configuration is defined; however, we did not address all of the problems of configuration specification. Better representations are needed for assisting software development, and when an improved representation is designed, it can be integrated into the Butler.

7.5. Contribution to Computer Science

This dissertation recognizes the special consideration that must be given to resource sharing in a network of autonomous machines. It is particularly important to consider the issue of protection, since the assumption of physical security used to design secure time-sharing systems is not necessarily applicable to personal computers. In addition, new techniques are required to support sharing when machines are autonomous, and the distributed nature of a network of personal computers must also be accommodated.

I have identified a number of requirements for a system that supports resource sharing in a network of personal computers and presented a design for a system that meets these requirements. The requirements include the ability to invoke remote operations according to a configuration specification, secure authentication and authorization of resource sharers, and support for the autonomy of both resource borrowers and resource lenders.

The information in this dissertation can be used to construct networks that support resource sharing without sacrificing user autonomy. In particular, this dissertation presents a comprehensive and coherent design for a resource-sharing facility called the Butler. The design will serve both as a basis for implementation and as a point of departure for the design and evaluation of alternatives.

At a more abstract level, this dissertation tells how to construct an execution environment that conforms to a set of policies and is protected against exploitation. This is a necessary step toward more powerful and more secure computer systems based on personal computers and local area networks.

References

- [Accetta 80] Mike Accetta, George Robertson, M. Satyanarayanan, Mary Thompson.
The design of a network based central file system.
Technical Report CMU-CS-80-134, Carnegie-Mellon University, August, 1980.
- [Almes 80] Guy T. Almes.
Eden: Research in Integrated Distributed Computing.
In Workshop on Fundamental Issues in Distributed Computing, pages 1-5.
December, 1980.
- [Atwood 72] J. W. Atwood, B. L. Clark, M. S. Grushcow, R. C. Holt, J. J. Horning, K. C. Sevcik, and D. Tschritzis.
Project SUE Status Report.
Technical Report, University of Toronto Computer Systems Research Group, April, 1972.
no longer in print.
- [Ball 81] J. Eugene Ball.
personal communication.
- [Ball 82] J. Eugene Ball, Mario R. Barbacci, Scott E. Fahlman, Samuel P. Harbison, Peter G. Hibbard, Richard F. Rashid, George G. Robertson, and Guy L. Steele Jr.
The Spice Project.
In 1980-1981 Computer Science Research Review, pages 49-77.
Department of Computer Science, Carnegie-Mellon University, 1982.
- [Bartlett 81] Joel Bartlett.
A NonStopTM Kernel.
In Proceedings of the Eighth Symposium on Operating Systems Principles,
pages 22-29. ACM, December, 1981.
Also published as SIGOPS 15(5).
- [Baskett 77] F. Baskett, J. H. Howard, and J. T. Montague.
Task Communication in DEMOS.
In Proceedings of the Sixth Symposium on Operating Systems Principles,
pages 16-18. November, 1977.

- [Casey 81] L. M. Casey.
Decentralized Scheduling.
The Australian Computer Journal 13(2):58-63, May, 1981.
- [Cooprider 79] Lee W. Cooprider.
The Representation of Families of Software Systems.
PhD thesis, Carnegie-Mellon University, April, 1979.
- [Cosell 75] B. P. Cosell, P. R. Johnson, J. H. Malman, R. E. Schantz, J. Sussman, R. H. Thomas, and D. C. Walden.
An Operational System for Computer Resource Sharing.
In *Proceedings of the Fifth Symposium on Operating System Principles*,
pages 75-81. ACM, November, 1975.
published as SIGOPS Operating Systems Review 9 (5).
- [Daniels 82] Dean Daniels.
Query Compilation in a Distributed Database System.
Research Report RJ3423 (40689) 3/22/82, IBM, 1982.
- [Dannenberg 81] Roger B. Dannenberg.
AMPL: Design, Implementation, and Evaluation of A Multiprocessing Language.
Technical Report CMU-CS-82-116, Carnegie-Mellon University, March, 1981.
- [Dannenberg 82] Roger B. Dannenberg.
Resource Sharing In A Network Of Personal Computers.
PhD thesis, Carnegie-Mellon University, 1982.
- [Davies 81] Donald W. Davies.
Lecture Notes in Computer Science. Volume 105: *Protection*.
Springer-Verlag, New York, 1981, pages 211-245.
- [DoD 80] *Reference Manual for the Ada Programming Language*
United States Department of Defense, 1980.
- [Eade 77] D. J. Eade, P. Homan, and J. H. Jones.
CICS/VS and its role in Systems Network Architecture.
IBM Systems Journal 16(3):258-286, 1977.
- [Farber 73] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowe.
The distributed computing system.
In *COMPCON Spring 73*, pages 31-34. IEEE, February, 1973.
- [Forsdick 78] Harry C. Forsdick, Richard E. Schantz, and Robert H. Thomas.
Operating Systems for Computer Networks.
Computer 11(1):48-57, January, 1978.

- [Gifford 81] David K. Gifford.
Cryptographic Sealing for Information Secrecy and Authentication.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*.
December, 1981.
- [IBM 79] *Customer Information Control System/Virtual Storage (CICS/VS) Version 1, Release 4, Introduction to Program Logic*
IBM, 1979.
- [Janson 76] Philippe A. Janson.
Using Type Extension to Organize Virtual Memory Mechanisms.
PhD thesis, Massachusetts Institute of Technology, 1976.
- [Jones 82] Mike Jones, Richard F. Rashid, and Mary Thompson.
Sesame: The Spice File System
Department of Computer Science, Carnegie-Mellon University, 1982.
Spice Document S140.
- [Kapur 80] Deepak Kapur, and Srivas Mandayam.
Expressiveness of the Operation Set of A Data Abstraction.
In *Seventh Annual ACM Symposium on the Principles of Programming Languages*, pages 139-153. January, 1980.
- [Kent 80] Stephen T. Kent.
Protecting Externally Supplied Software in Small Computers.
PhD thesis, MIT, September, 1980.
- [Kent 81] Stephen T. Kent.
Security in Computer Networks.
In Franklin F. Kuo (editor), *Protocols and Techniques for Data Communication Networks*, chapter 9. Prentice-Hall, New Jersey, 1981.
- [Linde 75] Richard R. Linde.
Operating System Penetration.
In *Proceedings of the National Computer Conference*, pages 351-360.
AFIPS, 1975.
- [Lindsay 80] Bruce Lindsay.
Object Naming and Catalog Management for a Distributed Database Manager.
Research Report RJ2914 (36689) 8/29/80, IBM, 1980.
- [Liskov 82] Barbara Liskov, Robert Scheiffler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 7-19. January, 1982.

- [Millstein 77] R. E. Millstein.
The National Software Works: a distributed processing system.
In *Proceedings of the ACM Conference*, pages 44-52. Association for
Computing Machinery, October, 1977.
- [Moorer 77] James A. Moorer.
Signal Processing Aspects of Computer Music - A Survey.
Computer Music Journal 1(1), February, 1977.
Also published in *Proceedings of the IEEE*, July 1977.
- [Nassi 82] I. Nassi.
The Liberty Net: An Architectural Overview.
In *COMPCON, 1982 Fall*. IEEE Computer Society, 1982.
to appear.
- [NBS 77] National Bureau of Standards.
Data Encryption Standard.
Technical Report, Federal Information Processing Standards, 1977.
Publ. 46.
- [Needham 78] R. M. Needham and M. D. Schroeder.
Using Encryption for Authentication in Large Networks of Computers.
Communications of the ACM 21(12):993-8, December, 1978.
- [Nelson 81] Bruce Jay Nelson.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, 1981.
- [Organick 72] Elliott I. Organick.
The Multics System: An Examination of Its Structure.
The M.I.T. Press, Cambridge, Mass, 1972.
- [Parnas 75] David L. Parnas and Daniel P. Siewiorek.
Use of the concept of transparency in the design of hierarchically
structured systems.
Communications of the ACM 18(7):401-408, July, 1975.
- [Perq 81] *Perq Software Reference Manual*
Three Rivers Computer Corporation, Pittsburgh, Pennsylvania, 1981.
- [Popek 79] Gerald J. Popek and Charles S. Kline.
Encryption and secure computer networks.
ACM Computing Surveys 11(4):331-356, December, 1979.
- [Rashid 81] Richard Rashid, George Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*,
pages 64-75. December, 1981.

- [Rashid 82] Richard F. Rashid.
Accent Kernel Interface Manual
Department of Computer Science, Carnegie-Mellon University, 1982.
- [Saltzer 75] Saltzer and Schroeder.
The protection of information in computer systems.
Proceedings of the IEEE 63(9):1278-1308, September, 1975.
- [Sevcik 72] K. C. Sevcik, J. W. Atwood, M. S. Grushcow, R. C. Holt, J. J. Horning, and D. Tsichritzis.
Project SUE as a learning experience.
In *Fall Joint Computer Conference*, 1972, pages 331-338. AFIPS, 1972.
- [Shoch 81] John F. Shoch.
personal communication.
- [Shoch 82] John F. Shoch and Jon A. Hupp.
Notes on the "Worm" programs - early experience with a distributed computation.
Communications of the ACM 25(3):172-180, March, 1982.
- [Smith 80] Reid G. Smith.
The Contract Net Protocol: high-level communication and control in a distributed problem solver.
IEEE Transactions on Computers C29(12):1104-1113, December, 1980.
- [Spector 82] Alfred Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
Communications of the ACM 25(4):246-260, April, 1982.
- [Svobodova 79] Liba Svobodova, Barbara Liskov, and David Clark.
Distributed computer systems: structure and semantics.
Technical Report MIT/LCS/TR-215, Massachusetts Institute of Technology, March, 1979.
- [Tanenbaum 81] Andrew S. Tanenbaum.
Network Protocols.
Computing Surveys 13(4):453-489, December, 1981.
- [Thacker 82] C. P. Thacker, E. M. McCreight, B. W. Lampson, and D. R. Boggs.
Alto: A personal computer.
In Siewiorek, Bell, and Newell (editors), *Computer Structures: Principles and Examples*, 2nd edition, pages 549-572. McGraw-Hill, New York, 1982.
- [Thomas 73] Robert H. Thomas.
A Resource Sharing Executive for the ARPANET.
In *Proceedings of the National Computer Conference*, pages 155-163. AFIPS, June, 1973.

- [Thomas 75] Robert H. Thomas.
JSYS Traps - A TENEX Mechanism For Encapsulation Of User Processes.
In *Proceedings Of The National Computer Conference*, pages 351-360.
May, 1975.
- [Wilkes 79] M. V. Wilkes and D. J. Wheeler.
The Cambridge Digital Communications Ring.
In *Proceedings of the Local Area Communications Network Symposium*.
National Bureau of Standards, May, 1979.
- [Wright 82] Keith Wright.
Matchmaker: A remote procedure call generator
Department of Computer Science, Carnegie-Mellon University, 1982.
Spice Document S129.
- [Wulf 74] W. A. Wulf, et. al.
Hydra: The Kernel of a Multiprocessor Operating System.
Communications of the ACM 17(6):337-345, June, 1974.

Appendix A

Message Specifications

This appendix explains the notation used to describe message interfaces. Although messages are often thought of as simply carriers of data, their most common use for our purposes is to invoke remote operations. This usage is reflected in our syntax, which is based on the Ada [DoD 80] syntax for subprogram specifications.

The complete syntax is given below, using the syntax notation of the Ada reference manual [DoD 80]:

```
message_specification ::=
    procedure identifier [ formal_part ]
    | function identifier [ formal_part ] return subtype_indication

formal_part ::= (parameter_declaration { ; parameter_declaration })

parameter_declaration ::= identifier_list: mode subtype_indication

identifier_list ::= identifier { , identifier }

mode ::= [ in ] | out | in out
```

The meaning of a message specification is straightforward. Normally, the specification describes two messages, an *invocation* and a *reply*. The invocation message consists of the following:

1. An operation code that corresponds to the procedure or function identifier. This tells the receiver what operation to perform, and serves as a tag to identify the message format.
2. A list of parameters: every parameter with mode *in* or *in out* is included in the message in the order implied by the specification. Parameters are passed by value.
3. A sequence number to be returned in the reply message.

4. A reply port.

Since every message includes a destination and a reply port, these are not included explicitly in the specification syntax.

It is assumed that the receiver of the invocation message makes available an operation defined as a subprogram whose formal parameter list matches that of the message specification, and whose name corresponds to the operation code. The receiver of the invocation message performs the indicated operation using the supplied parameters and then constructs a reply message with the following components:

1. A reply code that corresponds to the procedure or function identifier. Like the operation code in the invocation message, this component identifies the message format.
2. A list of parameters: for reply messages, every **out** or **in out** parameter is included. For **function** specifications, the return value is treated as an additional **out** parameter.
3. A sequence number: this is a copy of the number used in the invocation message. Its purpose is to allow the invoker to match the reply with the invocation.

Normally, message interfaces use synchronous communication, that is, the invoker waits for a reply before proceeding; however, no reply is sent when a procedure is invoked with only **in** parameters.

Most of the message interfaces described in this dissertation are functions that return type *GeneralReturn*, which is a subtype whose base type is *Integer*. This format is used to implement a simple form of exceptional condition handling where the returned value is either *Success*, or a code indicating an exception. Warning: although not mentioned explicitly in every message interface specification, the return code *Error* is a standard non-specific exception code that may be returned by any operation.

This notation is based on that used in the Accent Kernel Interface Manual [Rashid 82] and on the design of Matchmaker, the Spice message interface generator [Wright 82].

Appendix B
Another Perspective On
Protection And Autonomy

Butler Blues

as improvised by Craig Madge, November 1981

Lyrics by Roger Dannenberg and Frances Krouse

medium blues in F

My ba- by's got a but-ler, she keeps him there to
guard the door. Say my ba-by's got a but-ler, she
keeps him there to guard the door. I went
down to see her, but now I can't get in no more.
Don't you know it gets me lone-ly, lone-ly right down to my
shoes? Don't you know it get's me lone-
-ly, lone-ly right down to my little old shoes? My ba-
- by she don't want me, an' I just got them
Butler Blues.

DATE
ILME