A Butler Process for Resource Sharing on Spice Machines

ROGER B. DANNENBERG and PETER G. HIBBARD
Carnegie-Mellon University

A network of personal computers may contain a large amount of distributed computing resources. For a number of reasons it is desirable to share these resources, but sharing is complicated by issues of security and autonomy. A process known as the *Butler* addresses these problems and provides support for resource sharing. The Butler relies upon a capability-based accounting system called the *Banker* to monitor the use of local resources.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *network operating systems*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.4.3 [**Information Systems Applications**]: Communications Applications

General Terms: Design, Security

Additional Key Words and Phrases: Autonomy, resource sharing, office automation, personal computers, process migration, negotiation

1. INTRODUCTION

Large-scale integration makes it economically attractive to replace time-shared computer systems with networks of personal computers. A personal computer, because it is dedicated to a single user, can support high-bandwidth, low-latency input and output far better than can a time-shared system, and potentially it can provide its users with better control over the computing environment.

However, there are several disadvantages to distributing resources on a personal computer network. For example, a user may need to access data that is only available on a remote machine, but security may dictate that the data cannot be transferred to any other machine, thus forcing the remote processor to be used to access the data. Another disadvantage is that the physical distribution of

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

resources may not match the distribution of the demands for service, which may result in some machines being idle while others are overloaded. Finally, even though a personal computer may have significant computational capabilities, its power will be less than that of a large mainframe computer. As a consequence, even though a network may collectively have tremendous computing power, there may be programs that are inappropriate for personal computers because of the amount of computation involved. All of these problems can be alleviated by resource sharing.

In this paper we describe a technique for sharing resources on the Spice personal computing environment [2] under development at the Carnegie-Mellon University Computer Science Department. A process known as the *Butler* provides the mechanisms to support resource sharing, with the assistance of a capability-based accounting system caller the *Banker*. We first describe the issues and give several examples, then describe the Butler and the Banker, and finally show how resource sharing can be negotiated.

2. RESOURCE-SHARING ISSUES

Before the problems of sharing resources in a personal computing network can be fully appreciated, we must first examine some of the characteristics of these networks. We are particularly interested in the issues of security and protection, both of hardware and software, the autonomy of the personal computer, and the policies and mechanisms for resource sharing.

2.1. Protection and Security

Most main-frame computers are physically secure (or at least they are assumed to be). Only authorized, trusted personnel are allowed access to the physical machine, and users' access to the information stored on the machine is through an operating system interface that is able to protect the information. The current state of the art in time-sharing system design [20, 28] allows users to be highly selective in granting 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.

This situation is to be contrasted with that for a network of personal computers. A personal computer is generally located at the point of use, in an office or home. Provided that the owner is the only user of the machine, it can be considered physically secure against everyone except the owner; in addition, no special software protection is required, since there is no sharing. If, however, personal computers are shared among several users, protection and security become issues. For example, if a corporation's accounting department wants to grant limited access to accounting information to the marketing department, some form of security is necessary. The problem is perhaps even more critical if information or resource sharing crosses corporation boundaries, say between a manufacturer and a distributor. We will see that the levels of protection that can be provided will depend upon the assumptions made about the physical security of the machines.

2.2 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 main advantages for the user. Autonomy implies that users can control the load on their machines and thus form some stable expectations of the computing power their machines will provide. Users of time-sharing systems, on the other hand, cannot form a stable expectation of the response time and available computing power, since the service level depends on the load, which is outside the user's ability to control. The second advantage of autonomy is that users are free to run whatever programs, operating systems, or microcode they wish. This is especially important in a research-oriented environment such as that supported by Spice. In an office information system, autonomy implies that machines and resources can be administered at the individual or department level.

The property of autonomy and the desire for stable expectations may at first seem to be incompatible with resource sharing. There are several reasons for believing that this is not necessarily the case. First, in a network of personal computers, one can expect many machines to be idle. When a machine is idle, there is no reason (aside from protection issues) for assuming that a user would not want to share idle resources. Second, users may wish to cooperate by sharing resources.

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 or her machine is shared. For human engineering reasons, the user should be able to create *policies* that constrain sharing. These policies are then administered by some component of the operating system.

2.3. Mechanisms and Policies

Any facility to support resource sharing should provide *mechanisms* for sharing without dictating how these mechanisms should be used. For example, the owner of a machine should be able to decide not to share the machine at certain times. This control is provided through *policies* that dictate how sharing may take place. By decoupling policies from mechanisms, means are provided to modify the behavior of the system by providing new policies—a simpler task than altering the mechanisms.

3. EXAMPLES

In order to illustrate some of these points, we give some examples of resource sharing. These problems will serve to motivate the solutions that are presented later in the paper. We start by giving some terminology.

Any sharing of resources 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. (The term *machine* will be used

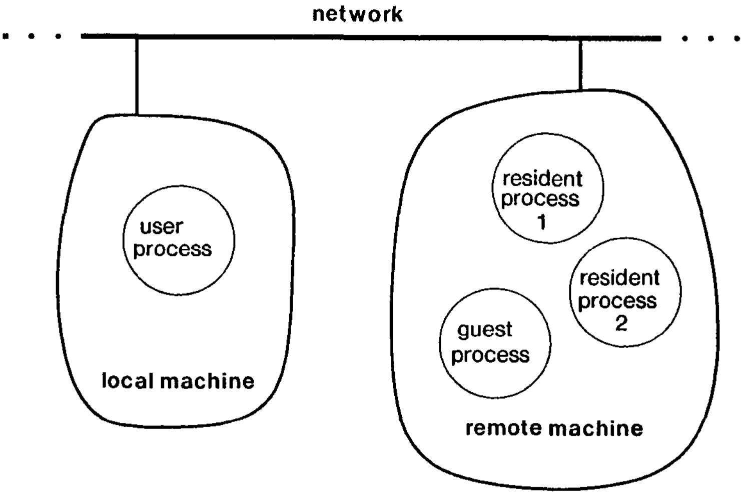

Fig. 1. Terminology for resource sharing.

rather loosely to mean not only hardware but also microcode 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 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.

3.1. 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 are private and the owner does not trust other machines. However, the owner would like to allow limited, controlled access to the database so that colleagues can make appointments and schedule meetings. A similar situation might occur if a department wanted to release some but not all of its records to another department in a corporation. To avoid giving users direct access to the database, the owner writes a program for making appointments that will run on his or her machine and handle user requests. In effect, the owner has used the physical security of the machine as a basis for building a protected subsystem [24].

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. Some means of authenticating a user's identity is therefore necessary. Finally, the owner may want to control when users are allowed to use his or her machine or restrict the priority of some users. The last problem again illustrates the concept of autonomy. An important problem is the difficulty of making any guarantees to a guest if he or she ultimately has no control over the machine.

3.2. Remote Program Execution

Consider the case of a user who has just edited a large document and wants to format it for printing. It is desired to use another machine for this task so as not to degrade performance on the local machine. The first problem is to find a machine having some idle resources that can be borrowed. To do this, the network is used to locate and interrogate remote machines. Some form of negotiation can then take place to determine if the remote machine is willing to perform the computation. There are several issues to be addressed in this negotiation:

(1) *The Sharing Policy.* The remote machine must know what resources are available for sharing. The owner of the remote machine may wish to make different resources available to different users. Also, the amount of available resources may be a function of the current state of the machine. Normally, the owner would want residents to have priority over guests.

(2) *Configuration Specification.* Similarly, the user needs to know what resources are required to execute the guest. In this case the user only wants resources to run the format program. 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 in turn specifies the necessary resources and 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. The needs of the guest must be presented and compared with the resources that are available.

Assuming the remote machine agrees, a formatter has to be invoked, sources need to be retrieved over the network, and the resulting formatted document finally returned to the user or sent to a printer. Given enough idle resources, many such jobs could be performed in parallel on a number of machines.

3.3. Distributed Program

In this example the user wants to construct a large program that executes in parallel on many machines. The ability to share these resources will make additional applications feasible, since the computing power available through sharing will always be greater than that available on a single machine. Examples of applications that can benefit from this type of sharing are computer graphics, transaction processing systems, database systems, image and signal processing, design-rule checkers for computer-aided-design systems, and simulation of physical systems. One of the problems faced by the designer of such a program is how to handle machine failures and resource revocations. This problem did not exist in the previous example because the user could simply restart the remote job if it failed for some reason. In the current example users will need to handle many exceptional conditions to avoid restarting their programs. Since many machines are being used, the user is more likely to encounter problems.

This example raises two problems. First, the programmer needs a model of machine failure and resource revocation. Second, the programmer needs assistance in recovering from the revocation of resources. We discuss resource revocation in Section 7.1.

3.4. Other Problems

There are some problems not already mentioned in these examples. For instance, in the formatter example, what happens if the user's source file exhausts the resources of the host's machine? In this case there are some *ad hoc* solutions that could be built into the formatting program. On the other hand, it may not be wise to trust a program to control its use of resources carefully. Futhermore, this solution is not general enough; it cannot handle situations where the guest's program is untrusted.

Suppose that the user is malicious and has the goal of controlling or at least crashing the system on the remote machine, and assume furthermore 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 penetrate directly 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 will be able create new processes, the file system will be able to access the disk directly, etc. 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. 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. Our solution to this problem is presented in Section 6.

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 Section 5.4.

4. THE ARCHITECTURE OF THE BUTLER

Our approach to resource sharing is based on the concept of the Butler:[1] a process that controls access to a machine by enforcing a resource-sharing policy, and provides protection. The design of the Butler is strongly influenced by the autonomy premise and the goal of stable expectations; hence, it falls somewhere between the extremes of a distributed operating system adhering to global strategies, and a computer network of autonomous nodes with no resource sharing.

[1] The Butler is so named becaused it manages a personal machine on behalf of the owner in a way which 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 an entity separate 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 required the reimplementation of software that is critical for protection, then errors could arise. Under these circumstances 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 best regarded 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. With this in mind, the Butler should be regarded not so much as a program, but rather as an abstract specification of protocols and expected behaviors. However, given the difficulties of implementing a program like the Butler, we do not expect there will be more than one version in the Spice environment.

4.1. General Design

The purpose of the Butler is to allow clients to invoke operations on remote machines. In addition to providing the capability of simply running a program remotely, the Butler allows a client to invoke arbitrary services, such as the program for making appointments referred to in Section 3.1. In this way, the Butler supports the sharing of information as well as resources.

The Butler provides the user with a high level of support that addresses all of the problems raised in the examples given earlier. The Butler, therefore, is a rather "heavyweight" mechanism. It is intended that the Butler be used to initiate operations and then intervene only when necessary. Once the operation is in progress, the Butler adds little or no overhead. There remains a certain amount of overhead for protection, but this overhead will be present in any design.

At least two machines, the *local machine* and the *remote machine* are involved in sharing (see Figure 2). On the local machine, the program that needs to borrow resources relies on the local Butler to borrow those resources. In this role, the Butler is called an *agent.* The agent communications with the Butler on the remote machine which acts as a *host.* The host creates guests on behalf of the client. As before, any program that is sharing the remote machine (including another guest) is called a *resident.*

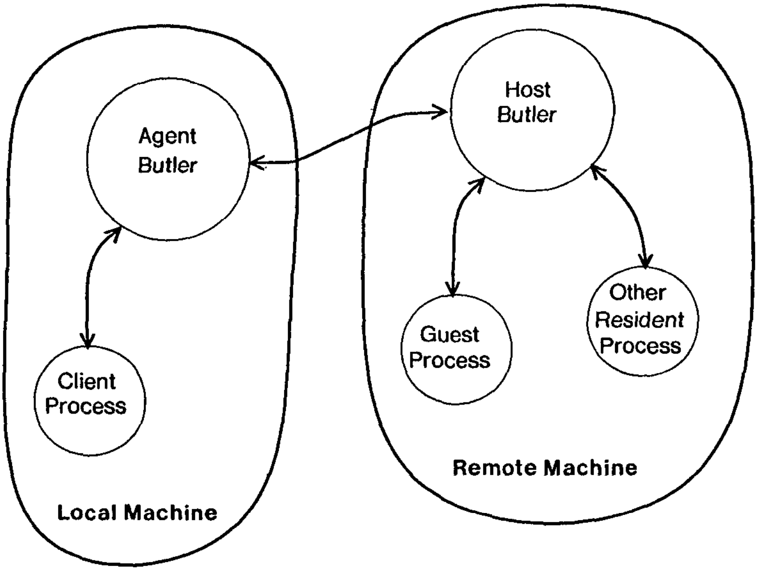

Fig. 2. Relationship between client, agent, host, guest, and resident.

4.2. 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.

If the resources are not available on one host, the agent looks for another host. Once a suitable host has been found, the identity of the host is authenticated. (Authentication could be performed before the negotiation takes place, but this would make the search for a host slower, because 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 (see Section 9).

4.3. The Butler as Host

The job of the host is to lend 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 it can, 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.

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 Butler handles the exception. The host's action can take one of several forms, depending on what action was requested at the initial negotiation.

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.

5. PROTECTION

In the previous section we described how the Butler facilitates resource sharing. In this section we examine how the Butler can provide protection for the client, agent, host, guest, and residents. Since the Butler relies on the security of lower levels of software, we must first state a few assumptions about security at the level below the Butler.

We assume that a user can load a secure Spice operating system, which allows multiple processes to run in separate virtual address spaces. We also assume that the user has the capability of verifying at load time that his or her 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.

5.1. Protecting the Client

If we assume that the client has a benevolent agent, (i.e., 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. In addition, the Spice file system [1] has a mechanism whereby the client can grant limited access rights to the guest. 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.

5.2. 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 in Section 6.

5.3. Protecting the Butler

The host protects itself from guests using the same mechanisms as those that 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. The Butler treats all incoming messages as suggestions and acts upon them only when the suggestions are consistent with local (trustworthy) data.

5.4. 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 we describe the authentication scheme used in Spice to discourage such behavior. A few stronger schemes, which require stronger assumptions, are then presented.

5.4.1 *Authentication.* 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. In the Spice system, a machine owner who allows a guest to borrow resources is responsible for executing a certified copy of the Spice 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 Authorization 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.) The CAS, acting as a trusted third party, uses the authenticated channels that were established by the first protocol. Space does not allow a full description of the second protocol, but it is similar in concept to the secret key exchange protocol described by Popek and Kline [21]. Details may be found in Dannenberg [7, 8].

5.4.2. *Stronger Schemes.* 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. It is necessary to assume that 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 [15]. All of these schemes rely on physical protection in one form or another. We do not plan to use extensive physical protection for Spice machines.

5.5. Summary of Protection

An important job of the Butler is to provide protection for resource sharers. The Butler relies on a secure operating system implementing processes with separate protected address spaces. However, the operating system must be extended with the Butler to deal with protection problems that involve multiple machines. The most important problem is the protection of the guest. In Spice, authentication is used to discourage malicious behavior that compromises a guest's security, but stronger techniques are possible if physical protection can be guaranteed. The Butler must also protect residents from the guest. The most important problem here is keeping track of resources given to the guest. The next section introduces the Banker process which solves this problem.

6. THE BANKER

The Banker is used for protecting and tracking resources. It represents a solution to the problem of laundered requests in which a guest coerces a server with greater privileges to behave maliciously. The problem of laundered requests also appears when we wish to revoke rights from a guest. Simply migrating or aborting the guest process may not recover many resources if the guest has employed local servers. Consider the following. The 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 is that the host has lost track of the fact that access to the file system and its resources are associated with the guest and should therefore be revoked.

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

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 account names to accounts. A user does not necessarily need to be authenticated to each server, since the user's account name 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. Account names 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. Account names, however, carry accounts, which can be large sets of values and are not necessarily Boolean. The capabilitylike aspect of account names 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. 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 and servers. Thus, it is possible, given an account name, to find all of the servers that have made withdrawals from that account. This is useful for recovering resources from a guest, because ordinarily these servers are the ones that have allocated resources to the guest.

So far we have seen how the Butler can share resources and protect them using assistance from the Banker. In the next section we take a closer look at how Butlers specify resources and how an agent negotiates with a host to obtain them.

7. NEGOTIATION OF RESOURCES

The resources controlled by the Butler are abstractions of different aspects of the physical machine. For example, priority is a resource, and access to the local file system is a resource. Guests must obtain *rights* that authorize them to use resources. In order to give rights to guests without losing control over resources, most rights are *revocable.*

The guest's rights and the method of revocation are established by negotiation. Negotiation is important because the client wants to run processes remotely with the guarantee that the processes will be able to obtain the resources they require. In the event that something goes wrong, the client would like to know in advance how each exceptional condition will be handled.

7.1. Revocation of Resources

Several methods are used to handle resource revocation. Together, these methods form a hierarchy of recovery procedures.

7.1.1. *Warning.* The first recovery 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 five additional seconds of CPU time, and ten additional disk pages along with a warning message. The ten disk pages would be added to the current allocation, regardless of the initial negotiations. 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.

7.1.2. *Deportation.* The second method is called *deportation* and is discussed further in Section 9. The goal of deportation is to provide a mechanism whereby the host can reclaim resources without harming the guest and 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.

7.1.3. *Abortion.* The third method of handling revocation is to abort the guest process and send an explanation to the agent. This method is invoked if all else fails or if higher level revocation-handling mechanisms are not requested. There is also a fourth possibility: If the host machine crashes, no notice can be sent by the host, but the agent is informed of the crash by the network server after a timeout period.

7.2. Resource Specification

In order to talk about negotiation, we must first present a more concrete description of how resources are specified.

7.2.1. 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. Other resources relate to services, such as access to the local file system. Resources can also refer to revocation; for example, a warning before revoking any rights is considered a resource.

7.2.2. Resource Data Types. All resources not related to revocation are grouped into a *BasicRights* data type. *BasicRights* is a collection of values representing either numerical limits (how many of resource X) or Boolean decisions[2] (the right to perform operation X).

GuestRights is a data type that fully specifies a guest's rights, including revocation. One field of *GuestRights* is a value of type *BasicRights*. Two additional fields specify if warning or deportation is to be performed. If both rights exist, warning will be attempted first. 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* is also included to specify the increment of resources required to handle the warning. Possible Ada [10] type specifications for the data structures discussed thus far are given in Figure 3.

7.3. 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, so the second value of type *GuestRights* expresses the minimum acceptable amount of resources.

[2] We 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 BASIC_RIGHTS is
   record
      DISK_PAGES        : INTEGER;
      NUMPROCESSES      : INTEGER;
      PRIORITY          : INTEGER;
      MICROCODE_ACCESS  : BOOLEAN;
      .
      .
      .
   end record;
type GUEST_RIGHTS(WARNING: BOOLEAN) is
   record
      INITIAL_RIGHTS  : BASIC_RIGHTS;
      DEPORT          : BOOLEAN;
      case WARNING is
         when TRUE  =>
            WARNING_RIGHTS : BASIC_RIGHTS;
         when FALSE => null;
      end case;
   end record;
```

Fig. 3. Ada type specifications.

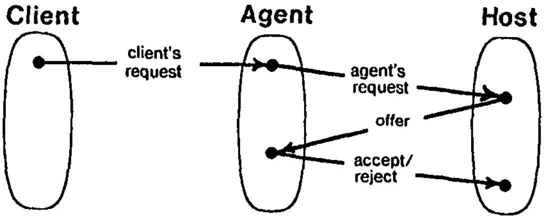

Fig. 4. Negotiation between an agent and host.

The negotiation process is outlined in Figure 4. 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.

7.4. Host Search Strategies

The client has the option of passing a *host search list* to the agent to specify which hosts to ask for resources. In the absence of a search list, the Butler finds 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.

Otherwise, the Butler does not attempt 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. It may be desirable for groups to optimize resource usage of the machines under their jurisdiction, and the Butler design allows for this.

8. POLICY

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

8.1. Terminology

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 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:

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

The rights may also be a function of the current machine state, the time of day, etc. 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.

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

9. AN EXAMPLE

In this example we see how a guest's rights are limited by mechanisms in the Banker, ultimately leading to deportation of the guest. For our example, we consider a signal-processing task used for locating and tracking objects acoustically. The sound from the tracked object is sensed by an array of microphones, and the resulting signals are digitized. The microphone signals are then cross-correlated. Ideally, each cross-correlation yields a sharp peak at a point indicating the time delay due to microphone positions. Triangulation can then be used to locate the object. Since many independent cross-correlations are performed, the cross-correlation step of this algorithm can easily be distributed.

In a typical configuration, a control program will assume the role of the client and request its agent to find some idle machines to run cross-correlation processes. Let us say the client specifies 500 disk pages and 100 CPU seconds in its request to the agent. The operation requested is to execute a cross-correlation process to be supplied by the client. The agent then searches for a host that will grant the requested resources. When contacted by the agent, a host consults its policy database to determine what rights are available for this user. After negotiating a set of *GuestRights,* the guest is created by the host Butler. At this time, the host creates an account for the new guest, and places limits on how much of each currency can be withdrawn from that account. The currency corresponds directly to the resources specified in the *GuestRights* record. In this case, the Banker is authorized to draw up to 500 disk pages and 100 CPU seconds for the guest. The name of the account is given to the guest, which would include this name in any request to a local server. In this example, the only service comes from the kernel, which provides virtual memory and processing time.

Now suppose the guest tries to allocate more resources than allowed by the currency in its account. Some server will attempt a withdrawal, and the Banker will discover that the guest has insufficient funds. A message to that effect is then sent to the server (in this case it goes to the kernel). Another message is sent to the establisher of the account (in this case the Butler) with a notification that includes the name of the server and guest.

Previous negotiation will have determined the method of handling the guest. In this example, let us assume that the guest is to be deported. The Butler sends a *deport* message to the Banker, specifying the account of the guest. The Banker then looks at the guest's account and notifies the appropriate servers that the guest is to be deported.[3] Notice that in this step all accounted resources allocated by the guest are located. The servers respond by packaging state information corresponding to the guest and sending it to the Butler. The address space of the guest, including registers and other processor states, combined with state information from various servers, makes up the complete state of the guest (see Figure 5).

The guest's state is sent to some other Butler, the identity of which was established during negotiation. This Butler interprets the state to determine what server connections are necessary, and state information prepared by the original servers is forwarded to new servers. Processes that were in use by the guest are "reincarnated" by the new Butler, and execution is resumed. The Spice operating system kernel, Accent [23], allows message ports to be moved, so communication paths between the client and guest are not broken by deportation.

10. RELATED WORK

A large amount of work has been done regarding security. Techniques for security in computer systems are surveyed by Saltzer and Schroeder [24], and encryption techniques is presented by Needham and Schroeder [19], Popek and Kline [21], and Davies [9]. Much less research has been done on topics that relate directly to the problems of resource sharing discussed in this paper. Svobodova et al. [27] consider a distributed system composed of autonomous nodes, but focus on language primitives for distributed applications rather than on the sharing of resources in the manner we have considered. Further work is reported by Liskov and Scheifler [17]. Other researchers have investigated applications of resource sharing [25], scheduling problems [3, 12, 26], and configuration control [5] in a personal computer network, but protection issues are generally not addressed. Distributed systems that run on collections of autonomous but trustworthy time-

[3] The Banker maintains a mapping from currency types to servers. Given an account, the Banker can locate servers that have made withdrawals from that account.

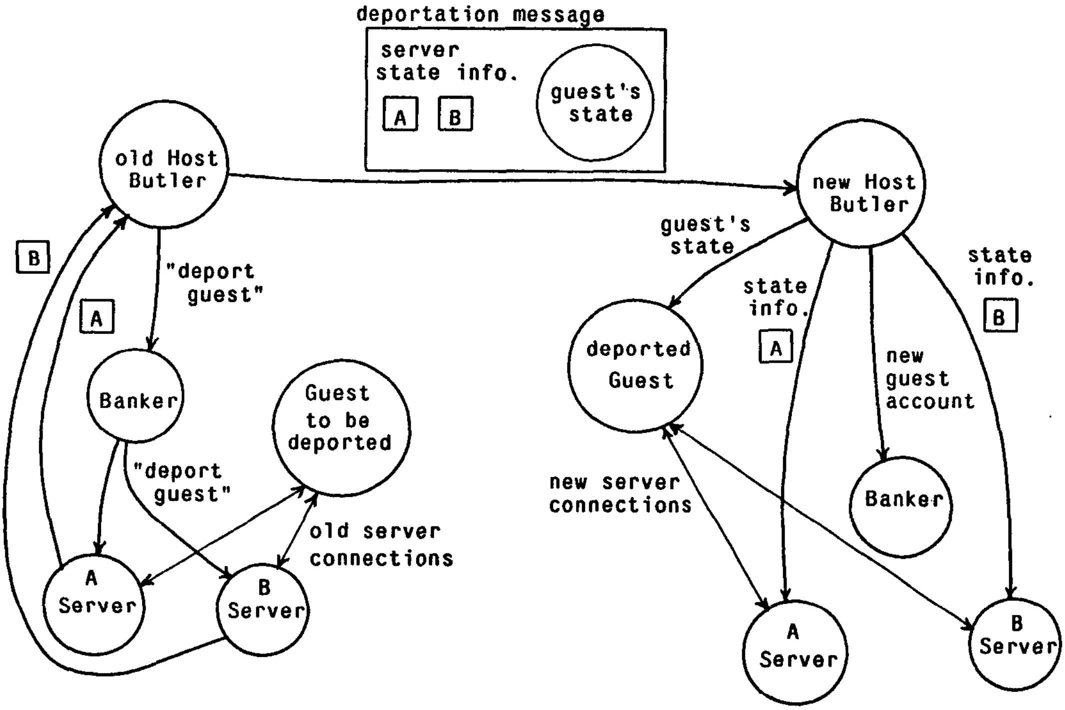

Fig. 5. Deportation of a guest process.

shared systems have also been constructed [6, 4, 11, 13, 16, 18]. Process migration in DEMOS/MP, similar to the deportation operation, is described by Powell and Miller [22].

11. CONCLUSIONS

The Butler is the result of an investigation of resource sharing on a network of personal computers. Its design is strongly influenced by concerns about autonomy and protection, both of which have led us to an approach where control is distributed and the policy of any machine is dictated by its owner. The system support for this approach is similarly distributed; each machine executes an independent Butler. When Butlers interact, they use authentication and negotiation to avoid blindly trusting another machine. At all times a Butler enforces its owner's policies. To do this, the Butler relies on lower level mechanisms which are a part of Spice, including a secure operating system kernel, a secure message passing system, network encryption, and the Banker.

At present a prototype Butler has been implemented to demonstrate deportation and to obtain some performance estimates [7]. The Banker has not been implemented, and the Central Authentication Server, a component of the Sesame file system [14] will soon be released for general use.

REFERENCES

1. ACCETTA, M., ROBERTSON, G., SATYANARAYANAN, M., AND THOMPSON, M. The design of a network based central file system. Tech. Rep. CMU-CS-80-134, Carnegie-Mellon Univ., Pittsburgh, Pa., Aug. 1980.
2. BALL, J.E., BARBACCI, M.R., FAHLMAN, S.E., HARBISON, S.P., HIBBARD, P.G., RASHID, R.F., ROBERTSON, G.G., AND STEELE, G.L. JR. The Spice Project. In *1980–1981 Computer Science Research Review.* Carnegie-Mellon Univ., Pittsburgh, Pa., 1982, pp. 49–77.
3. CASEY, L.M. Decentralized scheduling. *Australian Comput. J. 13,* 2 (May 1981), 58–63.
4. COSELL, B.P., JOHNSON, P.R., MALMAN, J.H., SCHANTZ, R.E., SUSSMAN, J., THOMAS, R.H., AND WALDEN, D.C. An operational system for computer resource sharing. In *Proceedings of the 5th Symposium on Operating System Principles* (Nov. 1975). ACM, New York, pp. 75–81 (published as *SIGOPS Operating Syst. Rev. 9,* 5).
5. CRAFT, D.H. Resource management in a decentralized system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Oct. 1983). ACM, New York, pp. 11–19.
6. DANIELS, D. Query compilation in a distributed database system. Res. Rep. RJ3423, IBM, San Jose, Calif., 1982.
7. DANNENBERG, R.B. Resource sharing in a network of personal computers. Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1982.
8. DANNENBERG, R.B. Protection for communication and sharing in a personal computer network. In *Proceedings of 5th International Conference on Distributed Computing Systems* (May 1985). IEEE, New York, pp. 88–98.
9. DAVIES, D.W. Lecture Notes in Computer Science. V Protection. In *Distributed Systems—Architecture and Implementation.* Lecture Notes in Computer Science, vol. 105. Springer-Verlag, New York, 1981, pp. 211–245.
10. DOD. *Reference Manual for the Ada Programming Language.* United States Department of Defense, 1980.
11. FORSDICK, H.C., SCHANTZ, R.E., AND THOMAS, R.H. Operating systems for computer networks. *Comput. 11,* 1 (Jan. 1978), 48–57.
12. HORNIG, D. Automatic partitioning and scheduling on a network of personal computers. Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1984.
13. IBM. *Customer Information Control System/Virtual Storage (CICS/VS) Version* 1, *Release* 4, *Introduction to Program Logic.* IBM, 1979.
14. JONES, M., RASHID, R.F., AND THOMPSON, M. Sesame: The Spice file system. Spice Document S140, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1982.
15. KENT, S.T. Protecting externally supplied software in small computers. Ph.D. dissertation, M.I.T., Cambridge, Mass., Sept. 1980.
16. LINDSAY, B. Object naming and catalog management for a distributed database manager. Res. Rep. RJ2914, IBM, San Jose, Calif., 1980.
17. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. In *9th Annual ACM Symposium on Principles of Programming Languages* (Jan. 1982). ACM, New York, pp. 7–19.
18. MILLSTEIN., R.E., The National Software Works: A distributed processing system. In *Proceedings of the ACM Conference* (Oct. 1977). ACM, New York, pp. 44–52.
19. NEEDHAM, R.M., AND SCHROEDER, M.D. Using encryption for authentication in large networks of computers. *Commun. ACM 21,* 12 (Dec. 1978), 993–938.
20. ORGANICK, E.I. *The Multics System: An Examination of Its Structure.* M.I.T. Press, Cambridge, Mass, 1972.
21. POPEK, G.J., AND KLINE, C.S. Encryption and secure computer networks. *ACM Comput. Surv. 11,* 4 (Dec. 1979), 331–356.
22. POWELL, M.L., AND MILLER, B.P. Process migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Oct. 1983). ACM, New York, pp. 110–119.
23. RASHID, R., AND ROBERTSON, G. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Dec. 1981), ACM, New York, pp. 64–75.

24. SALTZER, J.H., AND SCHROEDER, M.D. The protection of information in computer systems. *Proc. IEEE 63*, 9 (Sept. 1975), 1278–1308.

25. SHOCH, J.F., AND HUPP, J.A. Notes on the "Worm" programs—Early experience with a distributed computation. *Commun. ACM 25*, 3 (Mar. 1982), 172–180.

26. SMITH, R.G. The Contract Net Protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comput. C29*, 12 (Dec. 1980), 1104–1113.

27. SVOBODOVA, L., LISKOV, B., AND CLARK, D. Distributed computer systems: Structure and semantics. Tech. Rep. MIT/LCS/TR-215, M.I.T., Cambridge, Mass., Mar. 1979.

28. WULF, W.A., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C., AND POLLACK, F. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM 17*, 6 (June 1974), 337–345.

Received August 1984; revised April 1985