Protection for Communication and Sharing in A Personal Computer Network

Roger B. Dannenberg

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

The autonomy offered to users by most personal computer systems creates new problems for communication and sharing. Protection is required if personal computer networks are to attain the same level of user cooperation as is now possible in time-shared systems. A design is presented for protected communication that relys upon a secure high-level interprocess communication (IPC) mechanism and its extension across machine boundaries using encryption techniques. Protocols and support for authentication (identity certification) and authorization (rights certification) can be built upon the underlying IPC facility. To support resource sharing, an accounting process called the Banker is used to maintain a record of a borrower's resource utilization, and a sharing supervisor, called the Butler, is used to administer a locally determined sharing policy.

1. Introduction

In recent years, personal computer networks have become an attractive alternative to time-sharing systems. A personal computer, in contrast to a time-sharing system, gives its user a more stable and predictable set of computing resources. In addition, a network of personal computers offers the advantages of economy, support for highly interactive and real-time programs, graceful degradation in the presence of machine failures, and the ability of the system to grow incrementally.

Time-sharing systems, however, have some advantages that should not be overlooked. In particular, time-sharing systems provide users with a protected environment in which data and programs can be shared according to certain restrictions, and in which users can communicate through unforgeable messages. Furthermore, users can share expensive resources such as file storage systems, printers, and special-purpose processors.

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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Ideally, we would like the benefits of a personal computer network without sacrificing the good features of time-sharing. However, protection techniques developed for physically secure time-shared computers do not necessarily apply to networks of personal computers. An examination of the protection characteristics and general philosophy of personal computers suggests that, unlike time-sharing systems, they should be considered independent and autonomous. In this light, methods for secure communication, authentication, and authorization are designed. New strategies are also presented that provide protection for sharing resources.

The mechanisms described here were designed in the context of the Spice personal computing environment[1] at Carnegie-Mellon University. The reader is warned that the present tense is used throughout to give a clear view of the overall system design, but that some of the mechanisms described are not operational. A description of the present state of Spice is given in the last section.

2. Autonomy

Autonomy is an important factor in a personal computer network. Autonomy in this context refers to the degree of control exerted over a machine by its user. In the case of time-sharing systems, autonomy is low because the system makes resource-allocation decisions and prevents the user from arbitrarily accessing stored information. Although the protection and control mechanisms are usually implemented in software, there also must be physical barriers that prevent users from tampering directly with the machine. In contrast, personal computers are usually located at the site of the user -- at his home or in his office -- so that physical isolation of the machine from its user is not so simple.

CH2149-3/85/0000/0088$01.00©1985 IEEE
From PROCEEDINGS of the 5th International Conference on DISTRIBUTED COMPUTING SYSTEMS, May 13-17, 1985.

Consequently, most personal computers are unprotected and can be arbitrarily programmed by their users. However, even if a personal machine could be protected from its owner, this might not be desirable. Giving users total control over their machines allows them to use a variety of operating systems, write their own microcode, and test experimental system software. Also, users must be allowed to make local resource-allocation policies in order to achieve the predictability of performance that the personal computing approach has to offer. Thus, it is undesirable, if not impractical, to view personal computers as protected machines running trusted software. Rather, a personal computer and its user should be regarded as autonomous with respect to other machines and users. With this approach, users are not only enabled, but encouraged to take complete control of their personal machines. New techniques are now required to support secure communication, authentication, authorization, and resource sharing.

In principle, existing techniques such as those proposed by Needham and Schroeder[2] could be used to achieve many of the desirable protection properties mentioned above. In practice, however, these techniques are difficult to use in a layered system because they require cautious programs to perform all encription and key exchange protocols at the application program level. This implies that either (1) processes encrypt data even when they are on the same machine, or (2) the message facility must make the network visible to communicating processes, losing some of the advantage of a layered system. Alternatively, all security protocols could be hidden at the network transport layer, but this has the disadvantage that authentication and authorization are made on the basis of a machine identity rather than on a per-process basis. None of these alternatives are desirable. The primary goal of the present study is to develop protocols that provide flexible and secure authentication and authorization at the application program level, while restricting all encription to the network transport layer, thus keeping the network transparent to application programs.

Before describing the results of this exploration, I will briefly describe the Spice interprocess communication facility (Section 3) and a protocol to make the IPC secure (Section 4). Then, higher level protocols for authentication (Section 5), authorization (Section 6) and resource sharing (Section 7) are described.

3. Interprocess Communication

In this section, we will examine the interprocess communication facility provided by the Spice system. This facility is the foundation for the protection mechanisms to be described in later sections, and is based on abstract objects called *ports* and supported by the Spice operating system kernel, Accent[3]. A simplified list of operations provided on port objects is:

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

Port objects are protected by the kernel, and are referenced indirectly through port names that are local to each process. For each process, Accent maintains a table of correspondences between local port names and ports. This level of indirection prevents processes from forging port tokens and avoids naming conflicts.

A port is made accessible to another process by sending the name of the port in a message. 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 being *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. *Important:* in this paper, we will use the expression: "port Y is sent to some port X" to mean "*send* rights for port Y are included in a message and sent to port X". Although it is also possible to send *receive* rights, it will not be necessary here. The reader is referred to the description of Accent IPC by Rashid and Robertson[3] for more details.

Because a process cannot forge a port, ports are used extensively as capabilities within the Spice system. If a server wishes to grant certain rights to a client, it can send a port representing those rights to the client. Later, the client can present the port to the server as a token representing his rights. Just as with capabilities, the client could give its rights to another process by sending the port in a message.

Since ports cannot be forged and they provide message passing operations, ports can also be viewed as a high-level abstraction of secure connections. Below, we will see uses of ports as both capabilities and secure connections. In general, it is *not* useful to think of ports as names. In the Spice system, a *name server* process is often used to map string names into ports when a client wishes to make initial contact a server or some other process.

3.1. Network Servers

Message passing can be extended to the network by introducing *network server* processes[3]. 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).

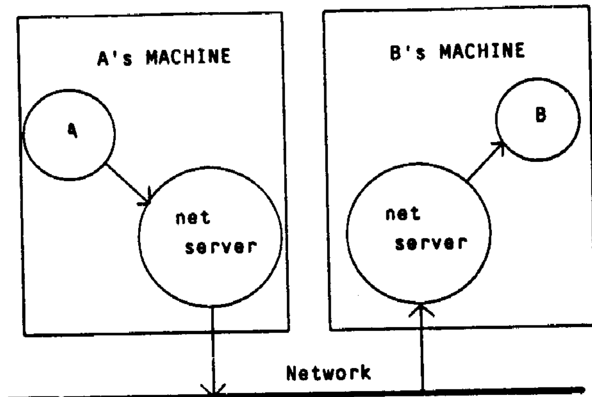

Figure 3-1: The use of network servers to achieve transparent intermachine communication.

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 port P_A. Instead, a network server process receives the message. Next, the port on which the network server receives the message is used to find a virtual circuit[4] on which to forward the message. The message at this point might 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 mapped into 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.

Ports can also be passed over the network. To do this, the network servers translate ports to network virtual circuits and back to ports. Thus, network servers transparently extend the kernel's IPC facility to the network.

Because all interprocess communication in Spice is based on this IPC mechanism, we have based our security measures on ports as well. While Accent IPC is secure within a machine, additional measures are required to obtain secure network communication. These measures could be handled individually by application programs; however, hiding the security measures in the network servers maintains the transparency of the network and simplifies application programs. It might be argued that trusting security to the network server makes programs less secure, but ultimately, the application program itself relies on the security of the underlying operating system. Therefore, there is no loss in security by trusting the network server.

4. Secure Network Communication

4.1. Encryption

This section is included for the reader who is unfamiliar with encryption techniques, which enable secure communication over an unprotected network. For the purposes of this paper, we will assume the use of conventional encryption algorithms such as DES[5]. Conventional encryption uses two functions:

Encode(*Message*, *Key*) **return** *Message*
Decode(*Message*, *Key*) **return** *Message*

where the key is of fixed length (usually 56 or 64 bits), and messages are of arbitrary length. The functions *Encode* and *Decode* are publicly known, and have the property that given an encoded message without the corresponding key, it is not feasible to recover the unencoded form. This property means that if the key is kept secret between two parties, then they can exchange messages that cannot be decoded by a third party. As an abbreviation, $\{D\}^K$ will be used in place of *Encode*(D, K), since it is implied that the same encryption function is always used.

Another class of algorithms, called *public-key* encryption systems[6] offers some advantages over conventional encryption. However, for the sake of simplicity, we will consider only the latter. Furthermore, the DES algorithm has been implemented on a single integrated circuit, so it seems likely that conventional encryption hardware will be available on personal computers before we see support for public-keys.

Encryption is useful not only to prevent the release of information to a third party, but also to prevent a third party from inserting messages into a stream of encrypted ones. To prevent forgeries, each encrypted message must have redundant information to indicate that it was actually encrypted with the expected key. (Otherwise, a random stream of bits would be indistinguishable from a legitimate message.) In addition, a technique such as sequence numbering must be used to avoid the possibility

of replayed messages. Popek[6], and Kent[7], and Voydock[8] give excellent descriptions of these techniques.

4.2. Key Exchange

Before one machine can communicate with another, the two machines must share a secret encryption key. This is a problem because the machines cannot exchange secret information before they agree upon the key, yet the key is necessary to exchange secret information. This circle of dependencies is broken by relying upon a trusted intermediary, which we will call the Central Authentication Server, or CAS. A secure communication channel, henceforth called a *connection*, is created between a pair of machines in two steps. First, each machine must create a connection to the CAS. Then, the CAS is used to forward an encryption key from one machine to another.

To connect to the CAS, a user enters his name and a secret key, K, known only to the user and the CAS. The user's machine then generates a random key R and sends the following message to the CAS:

$$name, \{name, R\}^{K} \rightarrow CAS$$

The CAS uses *name* to find K, and then decrypts the rest of the message, including R, which is used in all subsequent communication with the CAS. This scheme results in a secure and authenticated connection with the CAS, because only the user and CAS originally have copies of K. Therefore, the CAS knows that the machine identified by *name* must have R, and the user knows that only the CAS can decode R.

To connect to another machine, the CAS is used to forward an encryption key. Suppose machine A wants to communicate with machine B. First, machine A constructs a random key K_{AB}. It then sends the following message to the CAS:

$$\{K_{AB}, name_A, name_B\}^{R_A} \rightarrow CAS$$

where $name_A$ is the name of A, $name_B$ is the name of B, and R_A is the key used in A's connection with the CAS. Now, the CAS sends the following to machine B:

$$\{K_{AB}, name_A\}^{R_B} \rightarrow B$$

The CAS then destroys its copy of K_{AB}. Now, A and B both have K_{AB} and can use it to communicate securely. The connection is authenticated as well as secure, because A trusts the CAS to forward the key to B, and B trusts that the key originally came from A. In practice, messages must also contain serial numbers or use some other means of avoiding the possibility of replayed messages. This protocol is based on one described by Donald Davies[9]. It differs from that of Needham and Schroeder in that the CAS forwards information directly to B rather than returning information to A, and pre-existing connections (the ones encrypted by R_A and R_B) are used to eliminate some handshaking.

4.3. Summary

We saw in Section 3 how the Accent interprocess communication (IPC) facility can make the network transparent to processes, and in this section, we have shown that network servers can communicate securely. In the next three sections, we will assume a secure IPC facility and show how protection can be obtained using IPC port and message operations.

5. Authentication

Given the protocols above for secure communication between network servers, we can design authentication protocols at the Accent IPC level of abstraction. When an encrypted channel is established between a machine and the CAS, the network servers involved generate a virtual circuit and associate it with an IPC-level port connection from the personal machine's operating system to the CAS. This is called the *machine-to-CAS* port and is used by the operating system to communicate with the CAS at the Accent IPC level

The machine-to-CAS port does *not* represent the user, however. The Spice system distinguishes between users and their machines: there may be several users on a single machine, or a user may use several machines. The *machine identity* represents the person who initializes the machine and its operating system. This person is implicitly responsible for deeds (or misdeeds) performed by the machine. In contrast, a *user identity* represents a person who uses the machine to run application programs. He might or might not have the same identity as that of the machine, and is generally prevented by the operating system from doing any harm.

The Spice operating system requires that all users of the local machine authenticate themselves by logging in (if they use the keyboard, screen, etc.) or by using an authentication protocol if they access the machine via the network. The login procedure will be described first.

To log into a machine, the user enters his name and secret key as described in Section 4.2. This information is sent to the machine-to-CAS port in a *Login* message. Upon receipt of this message, the CAS checks its table to see that the user has supplied the correct key. Then, the CAS allocates an *authentication* port and returns it to a reply port specified in the *Login* message. If the key does not match, an error message is returned, and the user is denied access to the machine. Otherwise, the local operating system creates a command-interpreter process

for the user. (This login procedure is performed automatically after the machine is initialized (see Section 4.2) so that the user who is also the machine owner need not enter the same information twice.)

A common operation is for a user to request a service, such as printing a file, where the user must authenticate himself to a server on another machine. A simple (but, as we shall see, not very safe) way to accomplish this is for the user to send his authentication port to the server. The CAS is augmented with an operation of the following form:[1]

Verify1(port) **return** *UserID*

The operation is invoked by sending a message to the sender's authentication port. The argument is an authentication port to be verified, and the result is the user identity that corresponds to the argument. To see how this form of authentication can be used, suppose a client wants to print a file, given a port to a printing server. The client sends a file and his authentication port to the server (see Figure 5-1). The server then sends a *Verify1* message to the CAS. This message is sent to the server's authentication port to make sure it will be received by the CAS, and contains the port to be verified. The CAS then replies to the server with the name (if any) associated with the port.

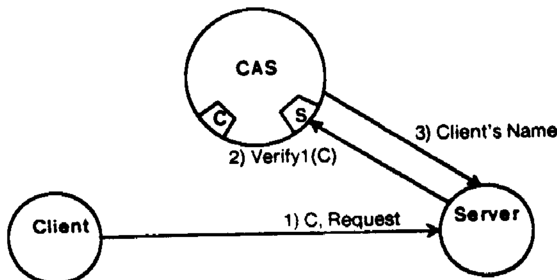

Figure 5-1: Simple authentication protocol.

There is a problem with this scheme: the server might not be secure and trustworthy. Since possession of the client's authentication port gives the server the ability to masquerade as the client, a malicious or careless server could jeopardize the client's security.

To avoid this problem, the identity of the client must be conveyed without giving away the ability to masquerade as the client. This is done by adding an operation called *Register* to the CAS:

Register(port)

which is invoked by sending a message to an authentication port. The message contains send rights for a port that is to be associated with the identity of the sender.[2] A companion operation:

Verify2(port) **return** *UserID*

can be used to determine the identity (if any) associated with a port.

Let us return to the previous example and see how our client can authenticate himself using these new operations (see Figure 5-2). First, the client allocates a port, X, and registers it with the CAS using the *Register* operation. Then the client sends X to the server. Now, the server can find the identity of the client with receive rights for X by sending a *Verify2* message to the CAS. Assuming the server is willing to print a file for the client (who is now authenticated by the result of *Verify2*), the server allocates a port, Y, and sends its receive rights to the client via X. The server also sends its name to confirm the name of the owner of port Y. Finally, the client sends his file to port Y. When a file is received on Y, the server knows that it comes from the client because the CAS claims that X belongs to the client, and Y was sent to X; therefore, only the client can send to port Y. This assumes that the client does not ever give away his authentication port (so no one else can register a port in the client's name) and that the client does not give away X (so that the server's message only goes to the client) or Y (so that only the client can send to the server).

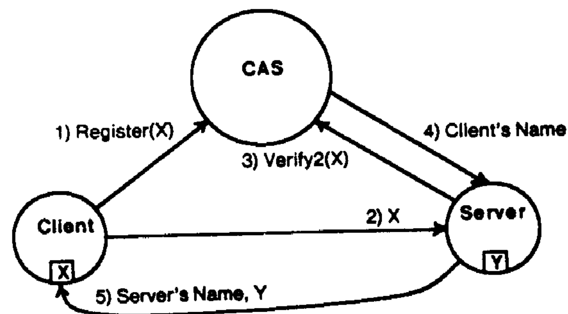

Figure 5-2: Improved authentication protocol.

To clarify the security properties of this protocol,

[1] We will use an Algol-like notation to describe server interfaces. Interfaces are actually implemented using one message to invoke the operation and supply parameters, and another message to return the results.

[2] The identity of the sender is known to the CAS because the message is sent to user's authentication port.

suppose a server attempts to masquerade as his client by passing along the registered port to yet another server, which we will call "victim". The server will attempt to obtain service from the victim by posing as our original client. As before, the client sends the server his registered port X, but this time the server sends X to the victim, claiming to be the client (see Figure 5-3). The victim will verify that X corresponds to the client, allocate a private port Z, and send Z and his name ("victim") to port X. Until now, neither the victim nor the client has noticed anything wrong, but observe that the printing server has now been bypassed, and will not even be able to intercept further messages. When the client receives the victim's message with the identifier "victim" rather than "printing server", the client detects the error and aborts the attempt to obtain service. At no point does the printing server have access to Z; therefore, it cannot forge any service requests to the victim.

There is one remaining problem with this protocol: a server might lie about its identity. For example, a malicious program might claim to be the printing server in order to deceive users into sending sensitive data. Two approaches can be taken to avoid this problem. First, a protected name server can provide a directory service to obtain server ports. The name server uses the authentication protocols already described to prevent unauthorized directory updates. This leaves us with the

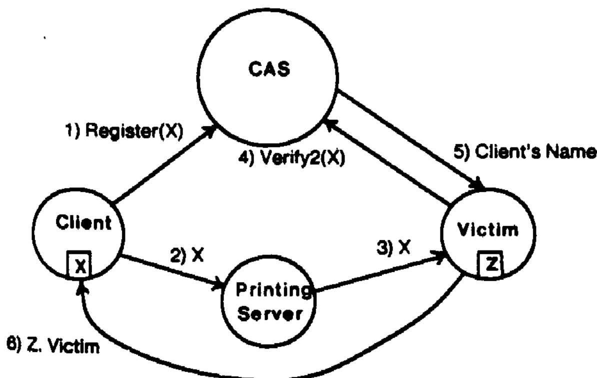

Figure 5-3: The printing server attempts to masquerade as the client.

problem that a malicious program could pretend to be the name server. This situation can be avoided by combining the name server and CAS[10], but the Spice designers wanted to keep the CAS as simple as possible and to provide a more general mechanism for authenticating servers. Therefore, a bilateral authentication protocol was designed to allow two processes to exchange ports such that each process also receives the identity of the other.

The client begins the bilateral authentication protocol as before by allocating a port and sending it to the CAS in a *Register* message. (See Figure 5-4.) This port, call it X, is also sent to the server. The server allocates a port, Y, and sends X and Y to the CAS using yet another version of *Verify*:

Verify3(*ClientPort, ServerPort*) **return** *UserID*

For this operation, the CAS looks for the identity associated with *ClientPort* and sends it to the server as the result of the *Verify3* operation. The CAS also finds the identity of the server (as determined by the authentication port on which the *Verify3* message is received) and sends it with the *ServerPort* to the client. Now the Client has a port associated with the server, and the server has a port associated with the client. The correctness of this protocol is based on the fact that both the client and the server authenticate themselves through their respective authentication ports and the trusted CAS.

Note that in this protocol, two *users* exchange identities and set up a secure communication channel through *ports*, whereas in Section 4.2, two *machines* (network servers) exchange identities and set up a secure communication channel based on *encryption*. Five messages are required at the Accent IPC level, and this

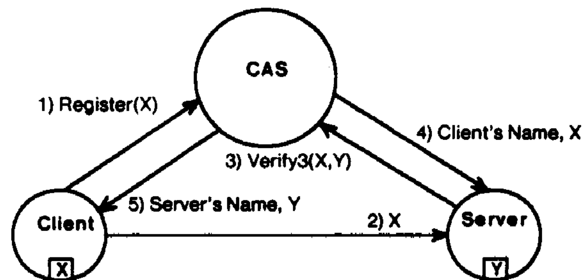

Figure 5-4: Bilateral authentication protocol.

will normally map onto five messages between network servers if secure connections have already been established between each pair of network servers. If not, then at least two additional messages must be sent to pass an encryption key from the Client's network server through the CAS to the Server's network server. At the opposite extreme, the Client and Server may be on the same machine. For this case, an implementation of the CAS in which every processor runs a local authorization server process (this organization has not been discussed here) could result in *no* network messages and *no* encryption. This is achieved with no loss of security because trusting a local authorization server is no worse than trusting the local kernel.

We have seen how a user (or his program) can certify the identity of some other user or program. In some instances, this is all the information necessary; for example, authentication is sufficient to prevent forged signatures in electronic mail. In other cases, authentication is the first step in determining what rights to grant to a client. This process, which we call *authorization*, is discussed in the next section.

6. Authorization

To support authorization, the CAS maintains *access groups* which represent subsets of users with a common set of rights. For example, members of the Spice project are included in the access group called *Spice-Project*. Access groups provide a convenient mechanism to grant or revoke rights to a collection of users; for example, one can grant file-access rights to the group *Spice-Project* without naming the members of that group explicitly. We must now have a secure way for servers to determine the access group memberships of a given user.

When a user logs in, the CAS finds the access groups of which the user is a member and associates them with the user's authentication port. In addition to returning a user's identity, the *Verify* operations return the user's access group membership list. It is up to the server to define what rights to associate with each access group.

Authorization is further refined by allowing the user to obtain an authentication port with a restricted membership list. The CAS operation *Restrict*:

Restrict(*DeleteList*) **return** *AuthPort*

is invoked by sending a list of access groups to be removed from the user's current membership list. The CAS associates the restricted list with a new authentication port that is returned to the user. Note that this does not permanently change any membership relations; it only changes the list that will be returned by a *Verify* operation on a port registered with a given authentication port. Thus, a user can be selective about the authentication rights he distributes.

To illustrate the value of restricted membership lists, suppose the system administrator wants to execute a program that he does not trust. Since the system administrator has many privileges, it would be unwise to give his authentication port to the program. Instead, he uses *Restrict* to obtain an authentication port with only ordinary user privileges. The untrusted program is given this restricted port to limit the damage that it can cause.

We have now seen a number of mechanisms that can be used to authorize the use of network resources such as printers, databases, file servers, and special-purpose processors. In Spice, we are also interested in supporting the sharing of personal machines among the user population. In the next section, facilities are discussed which were designed to cope with the problems of sharing autonomous machines.

7. Resource Sharing

There are several reasons for sharing personal machines. First, the user might be able to borrow an idle machine to reduce the load on his own. Users might also want to share machines as a means of sharing data. For example, it might be desirable to execute a database manager remotely to access remote data. Finally, the total processing power in a network of computers will be much greater than that of a single machine for many applications, so the additional resources available through sharing might make new applications feasible. Sharing of personal machines is thus useful for load leveling in the presence of idle machines, for information sharing, and for computational parallelism.

There has been little previous work concerned with the sharing of autonomous machines. Shoch and Hupp[11] describe the "worm" programs, which are distributed programs implemented on a network of Xerox Alto computers. No protection was provided, although some interesting applications were generated. Liskov has investigated programming language facilities to support distributed programs[12], but does not address the lower-level issues of obtaining access to or borrowing resources. Some of the issues of sharing data and programs are addressed by the National Software Works (NSW) project[13, 14]; however, the NSW is a logically centralized operating system, and machines in the NSW are not autonomous. RSEXEC[15, 16, 14] comes closest to our goals of supporting sharing on a network of autonomous machines. RSEXEC is implemented on the TENEX operating system and ARPA network, and it attempts to extend TENEX to the network environment in a transparent way. Since TENEX systems run on protected time-shared computers, RSEXEC emphasizes autonomy for reliability rather than for its implications for protection. Powell and Miller[17] describe process migration in DEMOS/MP, a system in which kernels trust each other. The authors discuss the possibility of mutually suspicious processors, but no protection mechanisms are suggested except the possibility of refusing to grant resources to a potential client. Hornig's Stardust system[18] is an experimental system running in the Spice environment for distributed parallel computation.

In the Spice system, sharing is supported in several ways. A host machine can protect itself by granting a limited set of rights to a guest. This insures that the host can remain autonomous. In addition, the resource borrower, or client, can obtain assurance that he will not be exploited if he grants some of his rights to a remote process. Finally, the programmer or user is not burdened with intricate protocols to achieve sharing.

Two subsystems, the *Banker* and the *Butler*[19], have been designed to support sharing. The Banker helps a host system to set and enforce limits on the use of resources by a guest, while the Butler administers the resource-sharing policy at the host machine and performs negotiation on behalf of clients who want to borrow resources. The Banker and Butler are described below.

7.1. The Banker

A common method of attacking a system is to exploit a privileged program that performs inadequate checking before carrying out a request[20]. For example, while a user program might not be able to attack a system directly, it might be able to use the file system to exhaust the available disk space and achieve the same goal. We refer to this as the problem of *laundered requests* because the identity of a request is made to appear "clean" by passing the request through a system program. This problem can be solved by keeping track of the origin of each request for service or resources.

The Banker helps to solve the problem of laundered requests by providing a secure accounting facility for the local machine, and by implementing a mechanism for intra-machine naming and authorization. A separate Banker runs on each machine and maintains *accounts*, which are lists of resources and associated quantities. These quantities are the *resource value*, giving the number of resource units allocated, and the *resource limit*, giving the maximum number permitted by the account. An unforgeable *signature* is used as the external representation of an account, and each user of the machine has an account and an associated signature, which is presented to servers to obtain resources.

Accounts can be created in two ways. The initial account is created by the Banker for the local operating system. As each server is initialized, it identifies to the Banker the type and number of resources that are made available by that server. The Banker credits the operating system account with these resources. The kernel, which manages physical and virtual memory, ports, and messages must also use the Banker for accounting, and these resources are initially credited to the operating system account. All other accounts are created as *dependents* of existing accounts. For example, when a user logs in, the operating system creates a dependent account representing a subset of the resources available to the operating system. The signature for the dependent account is then given to the user. In general, any process with a signature can create dependent accounts and pass the corresponding signatures to subsystems.

An account can have several dependents, each one including a set of *limits* indicating an upper bound on the number of resources the dependent can take from the parent account. By setting the limits to infinity, the parent allows dependents to withdraw as many of the parent's resources as necessary; alternatively, the parent can divide his resources among dependents by setting desired limits. For any given resource, the effective limit is defined recursively by:

limit(dependent) = min(*limit specified by parent for dependent, limit(parent)*)

To obtain resources, a process presents its signature to a server as a representation of authorization to receive resources. Next, the server takes the signature and presents it to the Banker with a request to withdraw the necessary resources. Then, the Banker replies with either *Success* or *Overdraft* to indicate whether or not the account contains the required number of resource units.

This description of the Banker has omitted some details to simplify the explanation. In order to prevent any process with a signature from making withdrawals on arbitrary resources, the Banker creates an unforgeable token for each resource as a capability to withdraw that resource from an account. This capability must be presented along with the signature to perform a withdrawal. Also, the Banker allows the parent to associate an *overdraft handler* port with each dependent account. If a dependent attempts to overdraw his account, the Banker sends a message describing the event to the overdraft handler port. The parent can also ask the Banker for a port corresponding to the server for a given resource. Using these primitives, protocols that deal with exhausted resources can be devised, and some of these are described in the next section. Finally, the Banker provides additional operations to check the balance of an account, to determine the total of withdrawals, and to allow servers to deposit resources when they are no longer needed by a client.

The Banker provides a link between a process that must control resource allocation (the owner of a parent account) and server processes that directly manage resources. Without the Banker, resources would have to be managed on a server-by-server basis. In the next section, we will see how the Banker is used by the Butler to enforce resource-sharing policies.

7.2. The Butler

As with the Banker, there is one Butler process per machine. The function of the Butler is quite similar to that of Craft's Resource Manager[21], although the Butler focuses more on issues of autonomy and protection. The Butler serves two roles in the Spice system. As a *host*, the Butler grants access to its machine according to a policy established by the machine owner. Rather than managing resources directly, the Butler relies upon the Banker to communicate the policy, and upon servers to enforce it. As an *agent*, the Butler functions to locate potential host Butlers and to negotiate to obtain resources for a client. Let us look at these roles in greater detail.

7.2.1. The Host Butler

The job of the host Butler is to loan resources while protecting the interests and autonomy of the machine owner. The owner communicates a sharing policy to the host Butler through a policy database. The database implements a mapping from user attributes to rights, where user attributes include identity and locality, and the rights have a direct correspondence to accounts in the Banker. An owner can also deny sharing entirely by entering an *exclusive* mode without modifying the database.

Normally, a host Butler is asked by an agent Butler to loan resources. The host then uses the bilateral authentication protocol to determine the identities of both the agent and his client. The client's identity is used in consulting the policy database to determine the client's rights on the host's machine.

Bilateral authentication is also used so that each party involved will know the identities of the others. This knowledge is useful if users are to detect malicious behavior and prevent its recurrence. Because machines are autonomous, a malicious machine owner can easily exploit a guest by circumventing the operating system's protection mechanisms. Similarly, if a guest is given the right to load microcode or use device drivers directly, then the guest can easily exploit the machine owner. Maliciousness can be discouraged here, as in society, by identifying and punishing wrongdoers. Thus, protection in this case is not absolute, but is based on social mechanisms.

If the host Butler agrees to furnish the requested resources, it builds a configuration consisting of one or more servers and a signature representing an account with the Banker. Access to the configuration is then returned to the agent. Because the Butler can use the Banker to limit the resources that servers will grant to a client, there is no need to create special servers for guest processes. This completes the host Butler's normal involvement with the agent. The host stands by, however, in case the client attempts to exceed the limits placed on its resource utilization. This situation would be detected by the Banker, which would send an overdraft notice to the host Butler.

To handle an overdraft, the host performs one or more of three recovery actions, according to earlier negotiation. The first possible action is to send a *warning* to a port supplied by the client. This allows application-specific actions, such as checkpointing, to be performed. If the warning fails to achieve the required reduction in resource utilization, then the host might attempt to *deport* the configuration by packaging the associated state information and sending it in messages to a port specified by the client. Finally, if warning and deportation fail, the host can *terminate* the configuration by notifying all of the participating servers. The Banker supports deportation and termination of configurations by locating these servers for the Butler.

7.2.2. The Agent Butler

The agent Butler is used to insulate the client from the details of the negotiation and authentication protocols. The client expresses his request to the agent as a configuration specification containing the rights *preferred* by the client and another configuration specification containing the rights *required* by the client. The preferred rights are forwarded to potential hosts, which reply with a specification of what rights they can offer. The agent then compares each host's reply with the client's required rights and decides to accept or reject the offer. To reduce the cost of negotiation, authentication protocols are only performed when an agent intends to accept an offer.

The Butler uses a name server to locate resources for its clients. Because of the constraints imposed by autonomy, there is no attempt to perform any sort of global optimization of resource allocation. This would require cooperation between Butlers that might make them vulnerable to malicious users.

8. Summary

As personal computers become more common, we will see an increasing need for communication and sharing among users. The mechanisms that have been described here will allow users of the Spice system to communicate and share resources with a high degree of protection and autonomy. At the heart of the system is the Accent IPC mechanism and the use of encryption to extend this high-level communication facility across machine boundaries in a secure manner. Several protocols rely upon this facility and a trusted intermediary to perform authentication and authorization. These protocols are interesting in that they rely solely upon the security properties of the IPC facility. Thus, network

transparency can be maintained, and authorization does not require a particular form or uniform implementation of encryption across all machines.

To protect and facilitate resource sharing, we introduced two processes called the Banker and the Butler. The Banker helps to solve the problem of laundered requests by providing a mechanism for keeping track of the resources used by a guest. The Butler is responsible for administering the machine owner's policies for sharing. It does this by negotiation with potential borrowers and creating the requested configurations when permitted.

9. Conclusion

The most important aspect of this work is the recognition that traditional security methods are inappropriate for networks of personal computers. It is only a matter of time before these networks move out of "friendly" research and office environments and into the general public where competing interests of users will demand careful attention to security. The general theme of this work has been to design an overall architecture supporting communication and sharing rather than to develop specific techniques. One result of this study has been the realization that the general problem is quite difficult. A practical solution will consist of at least the following: (1) a careful design of the underlying communication and security mechanisms, (2) a set of high-level protocols for application-level communication and sharing, and (3) a careful implementation of all servers to avoid loopholes through which applications can obtain unauthorized privileges. I believe the design described in this paper makes a significant step toward the realization of a personal computer network that supports a high degree of communication, sharing, and security. In particular, the technique of layered security protocols seems to have advantages in a layered system, and warrants further study.

At the present time, the Spice interprocess communication facility is operational, providing transparent inter-machine communication across a local area network. The IPC facility on a single machine is secure, but encryption is not likely to be provided for secure inter-machine communication on the present hardware. In spite of the lack of security, a number of distributed programs have been written and are in routine use for sending mail, version control, and graphics. The CAS is actually part of a larger system, called Sesame[22], which will provide naming, authentication, authorization, data storage and retrieval services. A preliminary version of Sesame is in use, and the full implementation of Sesame is in progress. An initial release of Sesame that includes a CAS and support for authentication should be made by the time of this publication. A prototype Butler was implemented for experimental purposes[19].

10. Acknowledgments

I would like to thank Peter Hibbard, Jim Morris, Rick Rashid, and Alfred Spector for advice and comments. To a large extent, this paper integrates the good ideas of many members of the Spice Project, and would not have been possible without their efforts.

References

1. 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*, Department of Computer Science, Carnegie-Mellon University, 1982, pp. 49-77.

2. R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, Vol. 21, No. 12, December 1978, pp. 993-8.

3. Richard Rashid, George Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 64-75.

4. Andrew S. Tanenbaum, "Network Protocols," *Computing Surveys*, Vol. 13, No. 4, December 1981, pp. 453-489.

5. National Bureau of Standards, "Data Encryption Standard," Tech. report, Federal Information Processing Standards, 1977, Publ. 46

6. Gerald J. Popek and Charles S. Kline, "Encryption and secure computer networks," *ACM Computing Surveys*, Vol. 11, No. 4, December 1979, pp. 331-356.

7. Stephen T. Kent, "Security in Computer Networks," in *Protocols and Techniques for Data Communication Networks*, Franklin F. Kuo, ed., Prentice-Hall, New Jersey, 1981, ch. 9.

8. Victor L. Voydock and Stephen T. Kent, "Security Mechanisms in High-Level Network Protocols," *ACM Computing Surveys*, Vol. 15, No. 2, June 1983, pp. 135-171.

9. Donald W. Davies, *Protection*, Springer-Verlag, New York, Lecture Notes in Computer Science Vol. 105, 1981, pp. 211-245ch. 10.

10. I. Nassi, "The Liberty Net: An Architectural Overview," *COMPCON, 1982 Fall*, IEEE Computer Society, 1982, to appear

11. John F. Shoch and Jon A. Hupp, "Notes on the "Worm" programs – early experience with a distributed computation," *Communications of the ACM*, Vol. 25, No. 3, March 1982, pp. 172-180.

12. Barbara Liskov, Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982, pp. 7-19.

13. R. E. Millstein, "The National Software Works: a distributed processing system," *Proceedings of the ACM Conference*, Association for Computing Machinery, October 1977, pp. 44-52.

14. Harry C. Forsdick, Richard E. Schantz, and Robert H. Thomas, "Operating Systems for Computer Networks," *Computer*, Vol. 11, No. 1, January 1978, pp. 48-57.

15. Robert H. Thomas, "A Resource Sharing Executive for the ARPANET," *Proceedings of the National Computer Conference*, AFIPS, June 1973, pp. 155-163.

16. 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," *Proceedings of the Fifth Symposium on Operating System Principles*, ACM, November 1975, pp. 75-81, published as SIGOPS Operating Systems Review 9 (5)

17. Michael L. Powell and Barton P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, October 1983, pp. 110-119.

18. David Hornig, *Automatic Partitioning and Scheduling on a Network of Personal Computers*, PhD dissertation, Carnegie-Mellon University, 1984.

19. Roger B. Dannenberg, *Resource Sharing In A Network Of Personal Computers*, PhD dissertation, Carnegie-Mellon University, 1982.

20. Richard R. Linde, "Operating System Penetration," *Proceedings of the National Computer Conference*, AFIPS, 1975, pp. 351-360.

21. Daniel H. Craft, "Resource Management In A Decentralized System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, October 1983, pp. 11-19.

22. Mike Jones, Richard F. Rashid, and Mary ThompsonDepartment of Computer Science, Carnegie-Mellon University, *Sesame: The Spice File System*, 1982, Spice Document S140