Local-area distributed systems

by R. C. Summers

Advances in computing and networking have led to the use of local-area distributed systems. The following are example configurations: workstations and file servers, multiple computers that present the image of a single computer, and heterogeneous workstations and mainframes that cooperate loosely. The paper focuses on the system software. It first discusses the forces leading to distributed systems and the obstacles to realizing the full value of the systems. Discussed also are common current uses of local-area distributed systems. Concepts and models are introduced. Requirements for user and program interfaces and for administration are presented, as well as major design attributes and design issues. Systems that represent the main approaches are described.

Advances in computing and networking have led to the use of local-area distributed systems. This overview treats the value, function, attributes, and design of these systems. It aims at providing a basic understanding as a first step for those who use, deploy, or build distributed systems. It assumes no specialized knowledge of networks or operating systems.

A *local-area distributed system* is a collection of computers in a limited area (such as a building or campus), connected by a high-speed local area network (LAN). This paper uses the term *node* for a computer in such a system. Thus a node can be anything from a PC to a large mainframe. Some examples of local-area distributed system configurations are the following: workstations and file servers on a LAN, collections of computers that present the image of a single computer, and heterogeneous workstations and mainframes that cooperate loosely.[1,2] LANs differ from wide-area networks (WANs) in their greater speed and capacity and their lower cost, and in their being controlled by the organizations that use them. These differences all encourage network use. WANs and multiprocessor systems are not discussed in this paper. Rather, this paper focuses on local-area system software.

Discussed first are the forces leading to distributed systems, the obstacles to realizing their full value, and the common current uses. The next section introduces concepts and models for describing distributed systems. This is followed by a discussion of the features that are needed. The last two sections present design attributes and issues, and examples of systems that represent the main approaches.

Forces, obstacles, and uses

The growing use of distributed systems reflects both organizational and technical forces. Distributed systems come about through the acts of distributing centralized systems or connecting formerly separate systems, and through choosing distributed structures for new systems. One impetus for distribution is the need for computing power, as applications multiply

and become more demanding. Consider, for example, a centralized system that is used for orders, personnel, and computer program development. As demand grows, a separate computing system is devoted to each of these functions, yet the functions continue to share data and other resources. Each system can be managed by the department responsible for the application. Security is enhanced, because each system is used only by people who need its functions. The hardware of each node can be selected for its appropriateness to the application.

A force leading to interconnection is the benefit of a larger user community.

Alternatively, the nodes of a distributed system can all perform the same functions, in which case the availability of the total complex is increased. If one node should fail, the others can take over its work. Such a distributed system can grow incrementally through the addition of new nodes.

A force leading to interconnection is the benefit of a larger user community, giving greater value to the sharing of data and applications and to communication through mail and messages. Also, distribution of software to the nodes becomes much simpler.

Local-area distributed systems allow specialized hardware or software of one node to be used from another node. For example, one node may have a high-speed graphics printer and another node a database management system. That hardware and software may be essential to other nodes, but used by them less frequently. Thus it is possible for the hardware and software to be shared. Such sharing is especially important for workstations, because their local function is more limited. Sharing allows more reliable and more cost-effective devices to be used for file storage. Price and performance considerations lead to systems in which microcomputers assume much of the computing load.

The potential of local-area distributed systems has not yet been fully realized. Centralized systems represent many years of experience, and the art of designing and managing them is highly developed. Distributed systems lack a comparably mature culture for their use and management. The software technology is still in its early stages. The potential security advantages are accompanied by new security exposures. Auditing becomes more complex as systems proliferate and disperse. Applications established in a centralized environment are not always easily transformed into distributed applications, and new distributed applications have been difficult to develop.

Distributed systems are most often used in universities and other campus-like clusterings of users. They usually develop from the bottom up, with departments or groups creating their own LAN systems that later join a global system connected by a backbone LAN. Small businesses use workstations on LANs. Engineering and research groups use LANs of high-performance workstations. The most common functions are file sharing, printing, messages, mail, document distribution, and remote task execution, though far more is possible.

Concepts and models

A system is said to present a *single-system image* if it behaves like a single computer, with the user essentially unaware what node is doing what function. One system structure that presents a single-system image is the *distributed operating system* (DOS).[3] A DOS distributes the basic operating system objects—files, processes or tasks, queues, and, perhaps, segments of memory. (A *process* is sometimes defined as an execution of a program, sometimes as a stream of activity.) A *distributed file system* (DFS)[4] provides some of the features of a DOS and can present a single file system image. A distributed system can be built on a DFS model, with distributed files supporting all the other functions of the system.

In other systems, users may be quite aware of the network and of where functions are carried out. This is true for a *network operating system* (NOS).[5,6] An NOS often involves heterogeneous hardware and operating systems, whereas a DOS is limited to a single operating system and often a single machine architecture. The NOS preserves the varying system images of its nodes. The DOS and NOS typically differ in their design as well as their image. The DOS usually changes or replaces the kernel of an existing operat-

ing system, whereas the NOS adds a network component at a fairly high level.

The *client-server model* applies to both DOS and NOS. A *client* (software executing at one node) makes a request for a service that is provided somewhere in the system. The request is handled by a *server*, which is software executing possibly at another node. (In commercial LAN systems, *server* often refers to a node specialized to provide a service.) In a DOS, each node is capable of playing both the client and the server roles. In other systems, some nodes lack the server capability. Even where all nodes have the same base capability, some nodes can be dedicated to specific functions. The client-server model is the basis for *resource-sharing systems*,[7–9] which are systems that emphasize the sharing of computing resources or services (as opposed to files or devices). A task is performed at a node that is specialized to handle the task or that has spare capacity.

In recent years, the *object-oriented model* has become increasingly important for system design. In such a model, every system object has a *type* that defines a set of operations to create and manipulate objects of that type. The implementation of an object is hidden from its users, who see only the type definition. This approach is fitting for distributed systems, where it is especially appropriate to hide how an object is implemented—perhaps on another computer of a different architecture. The client-server model supports the object model, in the sense that an object can be encapsulated by a set of servers. The object model can unify the views of users, programs, and system designers. This unification is rare, however, because most systems must support pre-existing user and program views.

Models of communication architectures are also valuable for understanding distributed systems. The OSI Reference Model[10] was not originally intended to apply within systems, where a system was defined as one or more autonomous computers. As LANs became more prevalent, however, OSI standards work was applied to the nodes of local systems. OSI, meaning Open Systems Interconnection, emphasizes the fact that the interfaces are in the public domain. The OSI model specifies layers, where each layer provides services to the layer above, and each layer adds value to the layers below. The lower layers are closer to the physical media, and the higher layers are closer to the application. The seven OSI layers are described elsewhere.[11] The functions discussed in this paper are primarily at layers 5 and 6, the *session* and *presentation* layers. These layers are supported by layer 3, the *network* layer, and layer 4, the *transport* layer. SNA similarly provides a layered architecture.[12,13]

For a homogeneous DOS, layering is not needed and may be inefficient, and the DOS can be viewed as a single system of an OSI network. For a heterogeneous NOS, however, and for the individual computers that are potential nodes of various distributed systems, there are benefits from following a standard model.[11,14] The design and business advantages of layering may outweigh its cost, which is becoming relatively smaller. Even personal computers can now support SNA and OSI protocols.

Ideally, the user should see no difference between local and remote function.

Requirements

User interface. As the scope of a user's work extends beyond the local computer to a distributed system, new and somewhat conflicting needs arise. This is especially true for heterogeneous systems. Ideally, the user should see no difference between local and remote function and should be able to use a familiar or preferred mode of interaction. From a user's point of view, *local* means the user's node, and *remote* means any other node. From an observer's point of view, all nodes are local. Systems of *heterogeneous* nodes require the interposition of a user interface management system that masks inessential differences, a difficult problem that is not solved by current systems. One approach is to use a workstation as a front end to all services, with workstation software masking some of the heterogeneity.

A major benefit of a distributed system is the user's greater ability to carry out concurrent activities, some of which are supported by remote nodes. A display interface with multiple windows allows the

user to monitor the progress of all these activities. With the X Window System,[15] which originated to meet the needs of network environments, an application can use windows on any display in the network and in a network-transparent way. X is built on the client-server model, the client being the application and the server controlling the display. Since multiple clients can have connections to a server at once, X supports the requirement for monitoring concurrent network-wide activities.

Program interface. Distribution imposes two new kinds of requirements. In the first instance, conventional and pre-existing programs must run properly and must be able to take advantage of distributed resources. For example, these programs must be able to share files on a file server without any changes to the programs. In the second instance, programs written for a distributed environment must be able to request services of other nodes, offer services to other nodes, or simply communicate with programs at other nodes.

In one approach to this, an operating system call (such as file I/O) is intercepted and transformed by programs called *stub routines* into a request on another node. The transformation may use information supplied in a prior command, such as the information necessary to map a disk drive to a remote directory. The generated request may itself be considered a program in a *network control language.*[16] Another approach is a system interface that can be invoked from various programming languages. This approach is used in the IBM Advanced Program-to-Program Communication (APPC), which provides verbs such as SEND_DATA and RECEIVE_AND_WAIT. Some systems conveniently generate such calls from multiple languages by means of *stub generators.*[17–19] The third approach is a programming language or language extensions designed specifically for distributed programming. Building distributed applications is difficult, and an appropriate language can help. A language also serves as a conceptual framework for the distributed system.[20–24] One of the most fully developed languages is Argus, an object-oriented language for distributed programming. Argus resources are encapsulated in *guardians*, which are abstract data types that expose the operations available on those resources. One guardian may use another's operations by means of remote calls, without regard for location.

Management. Controlling a distributed system is not in principle different from controlling a centralized one that supports varied work and many users. In both structures it is useful to distribute control to the people who own data, applications, and computing resources. A distributed system, even if composed entirely of personal computers, must be managed—a function called *network management.* Some of this management[13,25] has to do with the physical network—keeping track of the hardware configuration, detecting and correcting faults, monitoring performance, and providing data that can be used in planning for expansion. LAN hardware typically provides features to collect the needed data. At the higher *systems management* level, different features are needed. System software must control access to services and data. The software must identify users and authenticate service requests. It must perform accounting and ensure that software at different nodes is compatible. Also, the systems management software must maintain performance through functions such as load balancing and the migration of services and data between machines.

Most applications assume a model of local sequential execution, and distributed systems preserve that model to varying degrees.

Attributes of distributed systems

Transparency. A highly desirable attribute is *location transparency*, which is a system view that eliminates program and user concern with where resources are and even whether they are in one or many locations. Transparency is important for three reasons. It reduces the complexity of applications, ideally making them as simple to build as applications for centralized systems. Also, transparency allows applications to move easily between centralized and distributed environments. Further, transparency allows resources and applications to move from node to node without changing the behavior of the applications. Most applications assume a model of local sequential execution, and distributed systems preserve that

model to varying degrees. For transparency to be maintained, locations should not appear in programs. Assume for example, that the global name of a file starts with its node name. If the node name appears in a program, moving the file invalidates the program. If the meaning of a name depends on the node that uses the name, a program cannot execute at different nodes. Although complete transparency is a goal, systems that provide less than complete transparency can nevertheless be quite powerful.

The need is for a more structured form of communication and for addressing by function, rather than by process identifier.

Mechanisms for distribution. The methods used for distributing the activity of a system are extremely important, because they determine not only the efficiency and reliability of the system but also its convenience for users and application developers.

A method that is well-developed in centralized systems is *interprocess communication* (IPC) in which one process communicates with another by sending it a message that specifies the identity of the intended recipient. It seems natural to extend IPC so that the communicating processes can be at different nodes. This method has the advantage of compatibility with the centralized environment, but it also has weaknesses. It is difficult to write and debug distributed programs that use messages, without any higher-level structure. Also, it is inappropriate for one process to know the identity of another when the processes are distributed over a possibly heterogeneous network. The need is for a more structured form of communication and for addressing by function, rather than by process identifier.

A widely-used method for more structured communication extends the familiar concept of the procedure call. A *remote procedure call* (RPC)[4,26] behaves very much like a procedure call in a programming language. Parameters are passed by value, and return parameters are available when the call completes. In terms of the client-server model, the procedure being called is a *service* that may be either local or remote.[7] The caller is the client and the handler of the call is the server. (Note that remote calls can be supported locally also.) RPC has the advantage of simplicity and familiarity, but may be inefficient when the caller needs no results or acknowledgement. Another disadvantage is that RPC does not fully exploit the parallelism of a distributed system. The client is blocked (i.e., cannot execute) until the call completes. For this reason, some systems provide an *asynchronous RPC*. After making the call the client continues to execute and checks for completion at a later time. With this form, a process can be a client for multiple requests at once. In some systems, a service request sets up a *virtual circuit* that connects the client and server, and further messages can be exchanged using that circuit. A still more structured form of service request is a *distributed transaction*, which is discussed later in this section. Security features may be associated with a service request.

Once a service request mechanism is in place, it can be used for traditional operating system services, as well as for those service requests that are unique to a distributed environment. Printing, for example, becomes a service. A *name service* looks up objects including services and finds their locations and possibly other attributes. Other services provide file transfer, remote file access, and remote execution. Although motivated by distribution, this structuring of the operating system into services is valuable in its own right, contributing to a clean modular structure.

Systems vary a great deal in the way they implement distribution. LOCUS,[27] for example, intercepts file I/O calls and other operating system calls and invokes tailored high-performance communication protocols. In NOS, the communication component typically resides at a higher level and has a layered form. Low-level tailored mechanisms perform better, but limit portability. Many systems use standard protocols, such as TCP/IP,[28] which provide error checking, flow control, and other services. TCP/IP was developed for networks of hosts, but it is now well within the capabilities of workstations. The trend for workstations is to place the communication function in network interface units with their own processors and memory. The issue of special-purpose versus general-purpose transport protocols is discussed elsewhere.[29]

A distributed system requires special operating system support, including *lightweight* processes for system work. These are processes that can be created, dispatched, and destroyed quickly, whose use of memory is efficient, and whose IPC is fast. Such processes usually share an address space and are often queue-driven. Many systems assign each service request to a lightweight process that is either created on demand or assigned from a pre-existing pool corresponding to a service. The system must have efficient interrupt handling (because the completion of requests occurs asynchronously) and must have efficient storage management for buffering of messages. A time-out facility is needed to detect the failure of a service request.

Naming. Users and programs need to refer to the objects of a distributed system (users, workstations, services, . . .) so as to preserve location transparency. This means using a name that will not change when the object moves. Some component must then map names into locations. This component is called a *name service* or *directory service.*[30,31] The name service may be provided by one node or by each node for the objects that it owns. For greater availability, the directory used by the name service may be replicated at some or all nodes. The name service may also determine what objects satisfy the user's needs. For example, the user may need a printing service that accepts PostScript® format for desktop publishing. The name service must then associate objects with their names and their attributes. One essential attribute is location. Examples of other attributes for a print service are the formats it accepts and its speed. Required for a compute service are its architecture, memory size, and special coprocessors. By analogy with telephone directories, the finding of the attributes given the name is a *white-pages* service. The finding of the object given some attributes is a *yellow-pages* service. As systems grow and interconnect, users need the yellow-pages function to take full advantage of the resources of the distributed system.

A name may be *absolute*, i.e., having the same interpretation wherever it is used, or a name may be *relative* to the node that is using it, or even to some context within that node. Absolute names have the advantage that they can be passed around and have the same meaning wherever they are used. Absolute names must be unique, and one way to assure uniqueness is for a centralized name service to generate them when objects are created. An alternative is a hierarchically-structured name space, with part of a name identifying a node (which defeats transparency). Relative names must be interpreted within the appropriate context. An example occurs in LAN file systems. To refer to a remote directory the client uses a name that is mapped at the client node—using the local context—into the name offered by the server. For example, a server offers a directory under the name MEMOS, and a PC DOS user specifies that drive L refers to MEMOS.

When the naming systems of the nodes are different, a global scheme is used. When local-area systems are enlarged or interconnected, uniqueness must be ensured in the larger context. A promising solution is to retain the name services of the nodes or LANs and to bridge them by a global naming service. Naming requirements are complex, and many problems remain unsolved.

Homogeneity and heterogeneity. The original plans for a distributed system known as Andrew at the Carnegie–Mellon University "... assumed there would be nothing else on campus except Andrew. This model ideal was comparable to building an expressway across Wyoming, working with virgin territory, when in fact we were talking about running one through something like Chicago."[32] Heterogeneity is as common in organizations as in big cities, because local decisions are made autonomously. Clearly, the more alike the nodes, the simpler it is to build a workable system; but techniques for dealing with heterogeneity are being developed. Heterogeneity occurs at various levels: LAN hardware, communication protocols, machine architecture, operating system, and application interface. Both OSI and SNA address heterogeneity at the communication level, and OSI standards organizations are addressing heterogeneity at higher levels. IBM's Systems Application Architecture (SAA)[33] aims at consistent user and programming interfaces and communications, across heterogeneous architectures and operating systems.

File sharing illustrates many of the problems introduced by heterogeneity. Different character sets are used. Files are organized either as sequences of bytes or as sequences of records. Directories are flat or hierarchical. Integers have different lengths, and floating point representations differ. A system that provides remote task execution must also cope with differences in instruction set and configuration. Finally, the NOS itself must be portable across the different architectures or must exist in different versions. Problems of heterogeneity are discussed more fully elsewhere.[5,34]

Concurrency control and reliability. Even a centralized system must control concurrent access to shared data, to ensure that each operation is carried out properly and that integrity of data is maintained. Distributed systems are not different in principle, but concurrency control is far more difficult. Concurrency control is discussed along with reliability,[35] because the techniques used must consider both objectives.

The most common approach to providing reliability is through redundancy. Centralized systems are increasingly redundant, but a distributed structure is a natural framework for redundancy. File service can be provided by multiple nodes, and a file can be replicated at each of the nodes. This replication both protects the file against loss and increases its availability for access. At a lower level, replication of storage can be used to build an abstraction called *stable storage.*[36] Other services, or generalized processing power, can also be provided at multiple nodes. A system can go still further and replicate processes. Every process can have a backup process at a different node that receives the same messages and performs the same processing. If one process fails, the other can continue. In an object-oriented system, it is objects that are replicated. Systems that continue operating correctly even though some components fail are called *fault tolerant* or *highly available.*[37] Replicating data and processes introduces new overhead, because the nodes involved must coordinate their activity. Some systems (Argus, for example) allow an application developer to specify which objects are resilient to failures. Replication—especially of heavily-shared system data, such as directories—can improve performance. File replication is ideally transparent to users and to components outside the file system. The system must ensure that a request receives the latest version and must keep the copies consistent, even if failures occur.

File replication is ideally transparent to users and to components outside the file system.

The concept of a *transaction* is useful to understanding concurrency and reliability requirements.[38–41] The essence of a transaction is that it is *atomic*: either all its operations complete satisfactorily, or the effect is as though the transaction never started. Once the transaction *commits*, its updates can be assumed to be valid and actions complete. The transaction concept originated for database systems but is increasingly applied to file systems. Some applications for which distributed systems are used (in banking, for example) depend on transactions. A large system supports many transactions at once, and different transactions use and modify the same data. Operations from different transactions are interleaved in time, and operations of one transaction are carried out at different nodes. Despite the complexity, it must appear as though the transactions were executed in some order, without interleaving. This demands specialized support.[42,43]

An important concurrency control technique, used with or without transactions, is *locking*. When applications are moved from separate workstations to a LAN, locking becomes necessary and must be provided automatically. Locking must be coordinated in a distributed system. In LOCUS, for example, the node where the file is stored gives out locking tokens to other nodes. Another technique is *optimistic concurrency control*, where transactions are allowed to run without locking until they commit. At that time, a test is made to ensure that the committed transactions have executed in a serializable way. The optimism here is the belief that most transactions do not update the same data, and thus locking would generally be wasted. Transaction recovery is supported by *logging*: logs must be kept at all the nodes involved in a distributed transaction. Transaction-commit often uses a *two-phase commit protocol*, in which one node, acting as coordinator, directs all the nodes to prepare to commit and then, after confirmation from all nodes, directs them to commit.

In a distributed system, *deadlock* can occur not only because files or other resources are locked, but also because of communication. Node 1 may be waiting for a message from node 2, which is waiting for a message from node 3, which is waiting for a message from node 1. Deadlock detection is more complex in a distributed system, and many systems use the simpler technique of *time-outs*. In a time-out, if some event (typically a message receipt) does not occur within a specified time limit, the related service request or transaction is discarded. Time-outs have the advantage of responding to any kind of failure at another node.

Security. Distributed systems introduce new security problems.[44–47] Much attention has been given to network security, including LAN security, but relatively little to the overall system security. Any system must protect the confidentiality of its users' information and provide *access control*, that is, the system must ensure that resources are used only by those who are authorized to use them and that the resources are used in authorized ways. In order to do this a system must provide secure communication, ways of *authenticating* users and service requests, and control of access to services and data. It must back up these facilities with a protected audit trail.

The eavesdropping problem can be solved by encrypting all LAN transmissions.

A LAN presents several communication security problems. Data on the LAN are available to every network interface unit (NIU). A well-behaved NIU looks only at the data addressed to its node, but a serious intruder can tap into the LAN or capture a node, especially if the node is a personal computer. The intruder can then eavesdrop on any data on the LAN. The intruder can go further and insert phony messages onto the LAN, such as messages that purport to originate at another NIU. The intruder can also delete, modify, or replay legitimate messages. The combination of these passive and active attacks can gain the intruder unauthorized access to resources. Deletion of messages can result in denial of service to authorized users. The eavesdropping problem can be solved by encrypting all LAN transmissions, and digital signatures can be used to *authenticate* messages, that is, to determine if a received message is in fact identical to the one sent by the sender.[48] Message identifiers such as time stamps or sequence numbers can be used to prevent or detect replay.

Access control can be modeled as an *access matrix* where the columns represent resources or objects and the rows represent users. The entries in element *i, j* of the matrix then represent the access authority user *i* has to object *j*. Examples of access authorities are *read, update,* and *control*. Some systems do not follow this model, but instead rely on resource passwords.

Two ways to implement the access matrix model are access lists and capabilities. An *access list* is associated with an object and lists all the users who have access authority for that object, along with their specific authorization. It is often kept at the node where the object resides. A *capability* is a token or ticket that is passed by a user when requesting access to an object. A service request must contain either the user's identity or the capability and must be secure from modification. Access control can be enforced at the node of the object, either by general mechanisms of the node or by the server for that object. Alternatively, a trusted service at a secure node can be involved, as with the Kerberos system,[49] developed for the MIT Project Athena. Using encryption-based protocols, Kerberos authenticates users, and it authenticates clients and servers to one another.

Example systems

This section presents systems that have been chosen as examples of different approaches to local-area distributed systems. The systems are discussed in relation to the issues presented in the two preceding sections.

Distributed file systems. A DFS is crucial for networks of workstations, and it can also play a major role in configurations of larger computers. Various configurations are possible. There are those in which any node can offer files for use by other nodes. In other cases, all shared files reside at *file servers*, which may use specialized hardware or operating systems, or which may be specialized in function only. A mainframe can act as a file server. In a workstation environment, file servers provide availability and trustworthiness. They can use faster and more cost-effective storage, provide automatic backup and recovery, and allow users to move from workstation to workstation. In short, they provide many of the advantages of a centralized system. A DFS typically aims at some level of transparency—at a minimum, supporting existing programs that were not written for a distributed system. Remote file access can perform as well as or better than local access, if the file server has faster storage media than the local node.

Most DFSs are extensions of either the UNIX® file system or a personal computer file system. This section describes some UNIX-based systems.

We begin with the widely used NFS™[50,51] of Sun Microsystems, Inc. NFS is designed for portability to machines other than Sun's workstations and to operating systems other than UNIX. Its protocols, which are public, have been implemented by many vendors. NFS provides transparency and largely preserves the semantics of the UNIX file system. UNIX uses tree-structured directories, and a MOUNT command attaches a directory at any point in the currently active tree. With NFS, a MOUNT command can also attach remote files. Any subdirectory of a server node can be mounted.

Two related issues in the implementation of a DFS are whether a virtual circuit is used and whether state information[52] is kept at the server. An approach without states is simpler, but potentially less efficient. NFS uses an RPC protocol without virtual circuits that can be supported by different transport mechanisms. The server does not keep track of past requests. This greatly simplifies crash recovery—none is required at the server, and the client simply retries. In order to preserve UNIX semantics, file changes must be made on the disk for each write operation, which is a possible source of inefficiency.

The RFS[53] system of AT&T, Inc. has similar functions, but a quite different design. Any node may be client, server, or both. A server explicitly offers a subtree, under a symbolic name, and the client mounts the directory using that name. A distributed name server keeps track of all currently offered subtrees and their nodes. The client and server communicate through a virtual circuit between two nodes. The circuit is held as long as there are any mounts and is used for all requests between those nodes. The server maintains directory state information in cases where it can expect another access from the client to the same file (as in OPEN). Since directory state information is maintained, a recovery mechanism must be provided.

Other issues concern the amount or unit of data that is transferred per request, whether there is caching of data by the client, and the unit used for locking. In RFS each file system call is passed to the server, and no caching is done by the client; thus a record is the typical unit of transfer. File and record locking work in the remote case as well as the local case. A potential problem for a DFS is *time skew*. The server and client may not agree perfectly about the time of day, causing problems for users and programs that rely on the time a file was created or last modified.

Communication security is provided by encryption-based authentication and by the encryption of communications.

RFS solves this problem by calculating the difference between client and server views of time when a virtual circuit is established. Any time-based information is adjusted to compensate for this difference.

The Andrew[4,54,55] file system is part of a more general facility for distributed personal computing that is designed to support thousands of workstations. Andrew utilizes file sharing as the supporting base for most shared facilities. Other services are requested, for example, by depositing a request file in an appropriate directory. Andrew assumes that client workstations have significant amounts of disk storage. This allows Andrew to rely on client caching. Another requirement is that the file servers and communication be considered secure, whereas the workstations and network are not. The file servers are dedicated to that function and physical access to them is controlled. Thus they are not threatened by actions of either user programs or users. Communication security is provided by encryption-based authentication and by the encryption of communications. These security features are part of Andrew's remote procedure call.

In contrast to NFS and RFS, where the client MOUNT refers to a subtree offered by a specific node, Andrew presents one global shared name space. From the workstation viewpoint, the name space is divided into local and shared portions. The unit of sharing is a whole file, and whole files are cached on disk at the workstation. This works because the working set of files for a user is fairly small and because high-

performance, high-capacity workstations are used. Andrew and other DFSs also gain performance through caching in memory, which can be done at both client and server nodes. A workstation process, running on behalf of the file system, determines whether a shared file request can be satisfied locally—that is, if there is a valid copy in the local cache memory. If not, the file is fetched from the server that is the custodian of that file. Each server stores a subtree of the shared name space, and each keeps a copy of a database that identifies the custodian for each file. The request for a file can specify *callback*, which causes the workstation to be notified when the cached copy becomes invalid. When a cached file is closed, the updated copy is transmitted to the appropriate custodian. Authorization is designated by user or group for each directory and is implemented by access lists in a protected database replicated at each server. A server uses a pool of lightweight processes, with each such process handling one RPC connection.

Andrew supports PCs running PC DOS by means of PC servers that run on UNIX workstations. The PC servers receive PC DOS file requests from client PCs and transform them into UNIX file requests. Inasmuch as these servers have access to the global file tree, their clients also have that access.

Other DFSs are RT PC™ Distributed Services (which also provides distributed message queues),[56] Cedar,[57] and Sprite.[58] IBM's Distributed Data Management (DDM)[59] is an architecture for data sharing among systems of heterogenous hardware and software.

Distributed operating systems. We have seen that transparency is a powerful concept. Research on transparency led to the LOCUS[27] system. LOCUS is primarily a DFS, but it also provides remote processes and remote IPC. The nodes all run UNIX, but can use different hardware. For LOCUS, as for Andrew, the heart of the system is its file system. LOCUS supports distributed, possibly replicated files. It provides concurrency controls and atomic file update. The Transparent Computing Facility of IBM's AIX™ system incorporates technology similar to that of LOCUS.

Users and programs at all *using sites*—the LOCUS term for nodes—see a single global tree of files. The term *filegroup* is used for a self-contained subtree mounted at some point in the global tree. Copies of a file may exist at one or more *storage sites* (SSs). One of these sites stores the *primary copy* of the file—the copy guaranteed to be the latest version. Another site, possibly different from the using and storage sites, is the *current synchronization site* (CSS), which is responsible for synchronization and also for selecting the SS to be used when a file is opened for use.

Corresponding to each filegroup are physical containers. A filegroup may be partially replicated in containers at various sites. The primary copy of the container is always complete. The system ensures that the copies stay consistent and that a file request is satisfied by the most recent version. The file system implementation replicates its own data structures with the same mechanisms.

The OPEN for a remote file is passed to the CSS, which then selects the SS. The CSS identifies the latest version and asks each potential SS whether it has that latest version. The CSS then passes the OPEN to the proper SS. In contrast to Andrew, LOCUS reads the file on demand, one page at a time. All updates are atomic. Closing a file commits the changes, and explicit calls are also provided to commit changes and to discard changes. The changes are first committed in the primary copy. The SS notifies the CSS and all the other SSs, which then obtain the latest changes by remote reading of the primary copy. Multiple UNIX processes can use files concurrently. Even though these processes can be at different sites, LOCUS behaves exactly like UNIX in this respect.

LOCUS supports the UNIX pipe facility for interprocess communication. Because the behavior of pipes resembles that of files, the mechanisms also are similar.

LOCUS *remote tasking* allows a user to run a program anywhere in the network, if the user has the required authorizations. A running process can move between sites that have the same hardware architecture. Messages and signals can be sent to remote processes. LOCUS supports the UNIX FORK to create a child process with the same program as its parent, and EXEC to replace the executing program. FORK can be local or remote; EXEC can be used with dissimilar sites and can cause the process to migrate to other sites. One process can also signal another to migrate. When a process is created or moves, a site must be chosen for it. This choice is partly under application control. A process can specify where its subprocesses will execute, but this specification is not always respected, because, for example, of hardware limitations. The original site is responsible for keeping track of where a process resides, so messages and signals can be properly forwarded. A message for a remote process goes to the original site, which either

forwards the message or notifies the sender of the proper site to use. If the original site is not in the network, another site assumes its role.

Unlike many other systems, LOCUS relies on protocols that are tailored to the specific problem. For example, there is a simple protocol for OPEN that requires only four messages, after path name resolution is complete. Overhead is reduced by doing the simplest processing through interrupts and the rest through lightweight server processes.

MOS[60] is a distributed system with a structure different from that of LOCUS. The MOS kernel consists of an *upper kernel* that is considered a logical extension of the user's program, a *lower kernel* that implements the local objects of a machine, and a *linker* that allows the upper kernel to use the lower kernel of any machine. It is the linker that distinguishes local from remote operations and that intervenes for operations on network objects, whether local or remote. The state of a process is completely independent of the lower kernel state, which greatly simplifies process migration. VAXclusters[61] uses a very high-speed connection for processors that have their own memories and share disk storage. The system is controlled by a VAX/VMS distributed operating system. Research efforts in distributed operating systems include Accent,[62] the V system,[63] Emerald,[64] Amoeba,[65] and QuickSilver.[42]

A popular form of distributed system is a collection of personal computers in a LAN.

LAN operating systems for personal computers. A popular form of distributed system is a collection of personal computers in a LAN. A LAN operating system typically is a limited NOS, based on an existing OS, such as PC DOS. Most systems run on various LANs. Low cost is important for both the NIU and the software. Memory usage must be kept to a minimum, which leads to different software configurations for clients and servers. PC LAN systems originally provided simple disk servers, but evolved rapidly toward greater sophistication. Today these systems all provide file, print, mail, and message service. Others provide gateway service to other LANs, to a minicomputer or mainframe, or to the telephone network. Still others provide remote task execution on another PC in the LAN. The approach and implementation vary considerably.

Our first example is the IBM PC LAN Program.[66] Using that program any PC can act as a server, offering file or print service. A PC can be used locally, while acting as a server. If the usage is moderate, the local user may not experience any change in responsiveness. There is no centralized control, and users do not log on to the network. A file server offers a directory at any level in the hierarchy, assigning it a name and optionally a password and specifying the type of access allowed. To the client PC, the directory appears as a remote disk drive. No name service is provided. A user assigns a name to the local PC, and the system checks for duplicate names. Security is provided by passwords on offered directories. The same directory can be offered under more than one symbolic name, so that different users can be given different access.

Novell NetWare®[67] is designed quite differently. NetWare uses dedicated file servers that run special software aimed at good network performance. Although the DOS file interface is preserved for client programs, a NetWare server uses its own data format and access techniques, including caching of disk data in the server memory. NetWare users must log on, and resource authorizations are given to individual users and to user groups.

RM[7] is an experimental system for PC LAN resource sharing. It provides a general client-server framework within which services are built as applications. Remote execution is supported within that framework. RM also provides a user interface for the concurrent use of network services, and a programming interface for distributed applications.

An introduction to the use and management of PC LANs is found elsewhere,[68] as is a discussion of Microsoft's® LAN Manager.[8]

Heterogeneous systems. The Distributed Academic Computing (DAC)[5] system assumes heterogeneity of hardware and operating system, while encompassing mainframes, mini-computers, and workstations. Ex-

isting applications and operating systems are supported. The goals of supporting both heterogeneity and transparency lead to a complex but coherent design. The system supports the use of distributed objects in a location-transparent way. Each local operating system is augmented with both a local multitasking kernel and NOS kernel. The local kernel has a different implementation for each local operating system, whereas the NOS kernel is highly portable. The NOS kernel provides a *remote service call* (RSC) that extends RPC to provide asynchrony, access control, and accounting. Other functions of the NOS are built as system services that use kernel facilities. A global transport system[69] provides communication in a way that is independent of node architecture, network, and network protocol. There is a remote file access facility (RFA)[70] for heterogeneous file systems. This paper touched earlier on the problems arising from heterogeneity of file systems. The approach taken in RFA is the definition of a homogeneous global file system and the building of bridges between it and each local file system. RFA has client and server components, each providing a bridge to a local file system. The client RFA intercepts local requests, transforms them into requests on the global system, and sends them to a file server. The server transforms the global request into a request on its local file system. Thus the global file system does not exist in storage, but only as an intermediate form between bridges.

The Network Computer Architecture (NCA)[19] developed by Apollo is aimed at making it easier to do remote computation in a heterogeneous network. NCA takes an object-oriented approach. A client program uses RPC to avail itself of operations provided by some object of the network. The architecture includes a language for defining interfaces to objects. A compiler for that language produces stub procedures for both the client and server. These stubs convert parameters and results to and from a common network data representation. NCA uses the *broker* concept. Brokers do the work that allows clients and servers to cooperate. For example, brokers find objects, establish secure communications, and ensure that software license requirements are met.

Concluding remarks

The motivations for local-area distributed systems are strong and likely to become stronger. The hardware technology is advancing at an astounding rate, and the software technology is making good progress in performance, network transparency, and reliability. Greater LAN speed and capacity will lead to more applications that use image and voice. The great progress occurring in WAN technology means that WANs can soon support today's LAN functions, such as distributed file systems. The big challenges for the near future are in managing such complex systems, coping with heterogeneity, making applications easier to develop, and providing interfaces that allow the user to enjoy the full value of the distributed system.

Acknowledgments

The author is grateful to Hal Lorin, Patrick Smith, Charles Sauer, Herrmann Schmutz, Kurt Geihs, Richard Mosteller, Paula Newman, and Gene Tsudik for their insightful comments, and to John Marberg for his work in planning the paper. The support of James Jordan and Peter Woon made it possible to complete the effort.

Cited references

1. A. L. Scherr, "Structures for networks of systems," *IBM Systems Journal* **26**, No. 1, 4–12 (1987).
2. H. Lorin, "Systems architecture in transition—An overview," *IBM Systems Journal* **25**, Nos. 3/4, 256–273 (1986).
3. A. S. Tanenbaum and R. van Renesse, "Distributed operating systems," *Computing Surveys* **17**, No. 4, 419–470 (December 1985).
4. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A distributed personal computing environment," *Communications of the ACM* **29**, No. 3, 184–201 (March 1986).
5. K. Geihs, B. Schöner, U. Hollberg, H. Schmutz, and H. Eberle, "An architecture for the cooperation of heterogeneous operating systems," Report 43.8703 (1987); may be obtained from the IBM European Networking Center, Tiergartenstrasse 15, D-6900 Heidelberg, West Germany.
6. M. Seifert and H. Eberle, "Remote Service Call: A NOS kernel and its protocols," *Proceedings of the Eighth ICCC*, P. J. Keuehn, Editor, North Holland, Amsterdam (1986).
7. R. C. Summers, "A resource sharing system for personal computers in a LAN: Concepts, design, and experience," *IEEE Transactions on Software Engineering* **SE-13**, No. 2, 895–904 (August 1987). Reprinted in *Tutorial: Local Network Technology*, W. Stallings, Editor, IEEE Computer Society Press (1988), pp. 467–476.

8. A. Kessler, "OS/2 LAN Manager provides a platform for server-based network applications," *Microsoft Systems Journal* **3**, No. 2, 29–38 (1988).
9. S. A. Mamrak, D. W. Leinbaugh, and T. S. Berk, "Software support for distributed resource sharing," *Computing Networks and ISDN Systems* **9**, 91–107 (1985).
10. J. D. Day and H. Zimmermann, "The OSI Reference Model," *Proceedings of the IEEE* **71**, No. 12, 1334–1340 (December 1983).
11. J. R. Aschenbrenner, "Open Systems Interconnection," *IBM Systems Journal* **25**, Nos. 3/4, 369–379 (1986).
12. *IBM Systems Journal* (special issue: Systems Network Architecture) **22**, No. 4, whole issue (1983).
13. R. J. Sundstrom, J. B. Staton III, G. D. Schultz, M. L. Hess, G. A. Deaton, Jr., L. J. Cole, and R. M. Amy, "SNA: Current requirements and direction," *IBM Systems Journal* **26**, No. 1, 13–36 (1987).
14. F. M. Burg, C. T. Chen, and H. C. Folts, "Of local networks, protocols, and the OSI reference model," *Data Communications* **13**, No. 11, 129–150 (November 1984).
15. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics* **5**, No. 4, 79–109 (April 1986).
16. J. R. Falcone, "A programmable interface language for heterogeneous distributed systems," *ACM Transactions on Computer Systems* **5**, 330–351 (November 1987).
17. M. B. Jones, R. F. Rashid, and M. R. Thompson, "Matchmaker: An interface specification language for distributed processing," *Conference Record of the 12th ACM Symposium on the Principles of Programming Languages*, 1985, pp. 225–235; may be obtained from the Association for Computing Machinery, 11 West 42 Street, New York, NY 10036.
18. P. B. Gibbons, "A stub generator for multilanguage RPC in heterogeneous environments," *IEEE Transactions on Software Engineering* **SE-13**, No. 1, 77–87 (January 1987).
19. T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, and G. L. Wyant, "The network computing architecture and system: An environment for developing distributed applications," *Proceedings, IEEE COMPCON Spring 88*, 385–398 (March 1988); may be obtained from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855.
20. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and abstract types in Emerald," *IEEE Transactions on Software Engineering* **SE-13**, No. 1, 65–76 (January 1987).
21. B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl, *Argus Reference Manual*, MIT Laboratory for Computer Science, PMG Memo 54 (March 1987).
22. B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, "Implementation of Argus," *Proceedings, 11th ACM Symposium on Operating System Principles*, 115–126 (November 1987); may be obtained from the Association for Computing Machinery, 11 West 42 Street, New York, NY 10036.
23. B. Liskov, "Distributed programming in Argus," *Communications of the ACM* **31**, No. 3, 300–312 (March 1988).
24. R. Strom and S. Yemini, *The NIL Distributed Systems Programming Language: A Status Report*, Research Report RC-10864, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 (December 1984).
25. D. Coffield and D. Hutchinson, "Making a case for local network management," *New Communication Services: A Challenge to Computer Technology*, P. Kuhn, Editor, 451–456 (1986).
26. A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems* **2**, No. 1, 39–59 (February 1984).
27. G. J. Popek and B. J. Walker, Editors, *The LOCUS Distributed System Architecture*, The MIT Press, Cambridge, MA (1985).
28. A. Davidson, *An Introduction to TCP/IP*, Springer-Verlag, New York (1988).
29. R. W. Watson and S. A. Mamrak, "Gaining efficiency in transport services by appropriate design and implementation choices," *ACM Transactions on Computer Systems* **5**, No. 5, 97–120 (May 1987).
30. D. C. Oppen and Y. K. Dalal, "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment," *ACM Transactions on Office Information Systems* **1**, No. 3, 230–253 (July 1983).
31. L. L. Peterson, "The Profile naming service," *ACM Transactions on Computer Systems* **6**, No. 4, 341–364 (November 1988).
32. W. Y. Arms, quoted in *Carnegie Mellon University: Reaching for World Leadership in Educational Computing and Communications*, IBM Corporation Application Brief, GK21-0036 (October 1986); may be obtained through IBM branch offices.
33. E. F. Wheeler and A. G. Ganek, "Introduction to Systems Application Architecture," *IBM Systems Journal* **27**, No. 3, 250–263 (1988).
34. G. J. Popek, "Heterogeneity," *The LOCUS Distributed System Architecture*, The MIT Press, Cambridge, MA (1985), pp. 98–105.
35. A. S. Tanenbaum and R. van Renesse, "Reliability issues in distributed operating systems," *Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems, March 17–19, 1987 Williamsburg, VA*, 3–11 (March 1987); may be obtained from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855.
36. S. J. Mullender, "A distributed file service based on optimistic concurrency control," *Proceedings, 10th ACM Symposium on Operating System Principles*, 51–62 (December 1985); may be obtained from the Association for Computing Machinery, 11 West 42 Street, New York, NY 10036.
37. F. Cristian, "Issues in the design of highly available computing systems," IBM Research Report RJ5856 (October 1987); may be obtained from the IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099.
38. M. Herlihy, "Concurrency versus availability: Atomicity mechanisms for replicated data," *ACM Transactions on Computer Systems* **5**, 249–274 (August 1987).
39. B. J. Walker and S. H. Kiser, "The LOCUS distributed file system," *The LOCUS Distributed System Architecture*, The MIT Press, Cambridge, MA (1985), pp. 29–72.
40. M. J. Weinstein, T. W. Page, Jr., B. K. Livezey, and G. J. Popek, "Transactions and synchronization in a distributed operating system," *Proceedings, 10th ACM Symposium on Operating System Principles*, 115–126 (December 1985); may be obtained from the Association for Computing Machinery, 11 West 42 Street, New York, NY 10036.
41. B. M. Oki, B. H. Liskov, and R. W. Scheifler, "Reliable object storage to support atomic actions," *Proceedings, 10th ACM Symposium on Operating System Principles*, 147–159 (December 1985); may be obtained from the Association for Computing Machinery, 11 West 42 Street, New York, NY 10036.
42. R. Haskin, Y. Malachi, W. Sawdon, and G. Chan, "Recovery management in QuickSilver," *ACM Transactions on Computer Systems* **6**, No. 1, 82–108 (February 1988).
43. A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, and R. Pausch, "Distributed transactions for reliable systems," *Proceedings, 10th ACM Symposium on Operating System*

Principles, 127–146 (December 1985); may be obtained from the Association for Computing Machinery, 11 West 42 Street, New York, NY 10036.
44. V. L. Voydock and S. T. Kent, "Security in high-level network protocols," *IEEE Communications Magazine* **23**, No. 7, 12–24 (July 1985).
45. M. D. Abrams, "Observations on local area network security," *Proceedings, Aerospace Computer Security Conference; Protecting Intellectual Property in Space, March 2, 1985*, 77–82 (1985); may be obtained from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855.
46. M. D. Abrams and H. J. Podell, Editors, *Tutorial: Computer and Network Security*, IEEE Computer Society Press, Washington, DC (1987).
47. D. M. Nessett, "Factors affecting distributed system security," *IEEE Transactions on Software Engineering* **SE-13**, No. 2, 233–248 (February 1987).
48. R. R. Jueneman, S. M. Matyas, and C. H. Meyer, "Message authentication," *IEEE Communications Magazine* **23**, No. 9, 29–40 (September 1985).
49. J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," *Proceedings*, USENIX Association Winter Conference, Dallas (February 1988), pp. 191–202.
50. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, *Design and Implementation of the Sun Network File System*, Sun Microsystems Report, Mountain View, CA 94042 (1986).
51. J. DeVries, "NFS—An approach to distributed file systems in heterogeneous networks," *Digest of Papers—Eighth IEEE Symposium on Mass Storage Systems*, Tucson (May 1987), pp. 77–80.
52. C. H. Sauer, D. W. Johnson, L. K. Loucks, A. A. Shaheen-Gouda, and T. A. Smith, "Statelessness and statefulness in distributed services," *1988 Conference Proceedings, Uniforum*, Dallas (February 1988), pp. 145–155.
53. A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, "RFS architectural overview," *Proceedings*, USENIX Conference, Atlanta, GA (1986), pp. 248–259.
54. J. H. Howard, "An overview of the Andrew file system," *Proceedings*, USENIX Association Winter Conference, Dallas (February 1988), pp. 25–30.
55. J. H. Howard et al., "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems* **6**, No. 1, 51–81 (February 1988).
56. C. H. Sauer, D. W. Johnson, L. K. Loucks, A. A. Shaheen-Gouda, and T. A. Smith, "RT PC Distributed Services overview," *Operating Systems Review* **21**, 18–29 (July 1987).
57. D. K. Gifford, R. M. Needham, and M. D. Schroeder, "The Cedar file system," *Communications of the ACM* **31**, No. 3, 288–298 (March 1988).
58. M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *Communications of the ACM* **31**, No. 3, 134–154 (March 1988).
59. R. A. Demers, "Distributed files for SAA," *IBM Systems Journal* **27**, No. 3, 348–361 (1988).
60. A. Barak and A. Litman, "MOS: A multicomputer distributed operating system," *Software Practice and Experience* **15**, No. 8, 725–737 (August 1985).
61. N. P. Kronenberg, H. Levy, and W. D. Strecker "VAXclusters: A closely-coupled distributed system," *ACM Transactions on Computer Systems* **4**, No. 2, 130–146 (May 1986).
62. R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network operating system kernel," *Proceedings, 8th ACM Symposium on Operating Systems Principles* (December 1981), pp. 64–75.
63. D. R. Cheriton, "The V distributed system," *Communications of the ACM* **31**, No. 3, 315–333 (March 1988).
64. E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald System," *ACM Transactions on Computer Systems* **6**, No. 1, 109–133 (February 1988).
65. A. S. Tanenbaum, S. J. Mullender, and R. van Renesse, "Using sparse capabilities in a distributed operating system," *Proceedings, 6th International Conference on Distributed Computing Systems* (1986), pp. 558–563.
66. *IBM PC Local Area Network Program User's Guide*, IBM Corporation, Boca Raton, FL 88429-1328; the User's Guide 84X0495 and program diskette 84X0519 may be obtained through IBM branch offices.
67. S. S. King, "Novell advances," *PC Tech Journal* **6**, No. 6, 58–72 (June 1988).
68. J. Barkley, *Personal Computer Networks*, Special Publication 500-140, National Bureau of Standards, Gaithersburg, MD (July 1986).
69. M. Salmony, *Experiences in the Design of a Transport System for Heterogeneous Environments*, IBM European Networking Center Report 8601 (April 1986); may be obtained from the IBM European Networking Center, Tiergartenstrasse 15, D-6900 Heidelberg, West Germany.
70. U. Hollberg, H. Schmutz, and P. Silberbusch, "Remote File Access: A distributed file system for heterogeneous networks," IBM European Networking Center Report 43.8611 (November 1986); may be obtained from the IBM European Networking Center, Tiergartenstrasse 15, D-6900 Heidelberg, West Germany.

Rita C. Summers *IBM Los Angeles Scientific Center, 11601 Wilshire Blvd., Los Angeles, California 90025.* Ms. Summers is an IBM senior technical staff member and project manager at the IBM Los Angeles Scientific Center where her current work is in knowledge systems, distributed systems, and computer security. She has designed and implemented time-sharing and resource-sharing systems. She received two IBM Outstanding Contribution Awards for her work on virtual memory. She has published articles on resource sharing, computer security, and database security, and is co-author of a book on database security.

Reprint Order No. G321-5356.