Inside IBM's Distributed Data Management architecture

by R. A. Demers
J. D. Fisher
S. S. Gaitonde
R. R. Sanders

IBM's Distributed Data Management (DDM) architecture is an element of Systems Application Architecture™ that defines an open environment for sharing data in files and relational databases. DDM is a key element of IBM's Distributed Relational Database Architecture. DDM architecture enables programs to access and manage data stored on remote systems. It is a framework for a wide range of additional application services. Influenced by the concepts of object-oriented technology, DDM architecture is designed to be object-oriented. This paper examines DDM architecture from a number of viewpoints, considering why and how it was created, what it is, and how it has evolved.

Systems Application Architecture* (SAA*) defines a consistent set of application services that span IBM system platforms from personal computers to large systems, thereby making application services a unifying force for the future. Distributed application services are provided by SAA Common Communications Support (CCS). These SAA elements deal with how systems work together to provide services to applications distributed throughout a network. For the most part, these CCS services are not seen by the user. Software products that companies install provide the interfaces employed by programmers to develop applications that use CCS services. Knowing what is within these products enhances an understanding of how they work and how to develop effective distributed applications.

Two key SAA application services are Distributed File Management[1] and Distributed Relational Database Management.[2] Both of these services are built to the specifications provided by IBM's Distributed Data Management (DDM) architecture. DDM architecture enables programs to access and manage data stored on remote systems in a client/server relationship.

This paper is not intended to be a tutorial on the technical details of DDM architecture. That level of information is available in documents published on the architecture (particularly in Reference 3). Rather it examines DDM architecture from a number of points of view, considering why and how it was created, what it is now, and how it has evolved. The paper begins with a discussion of the role of software architecture in general, and continues with a brief history of DDM architecture. Next, the role of object-orientation in making DDM architecture uniform in style and highly consistent in structure is presented. It is followed by a presentation of the conceptual framework of the architecture in terms of formally defined layers of objects, along with discussions of its key classes of objects. Concluding the paper are discussions of

product-unique extensions to DDM architecture and the relationship of DDM architecture to international standards.

The role of software architecture

We begin by distinguishing a software *architecture* from *implementations* of the architecture. A software architecture is really just a set of specifications, a set of blueprints, for constructing products. As such, a software architecture is not installed on computer systems. Rather, products based on a software architecture are installed. What, then, is the role of software architecture in developing products?

Consider, for example, a suspension bridge over a large river. The bridge is designed by an architectural firm to meet the requirements of its owners and of its users, the driving public. It is then built by a separate construction company. Only after being certified for use by independent building inspectors are people then allowed to drive over it. There are clear similarities between this example and designing, constructing, assuring, and delivering computer software products.

An architectural firm is responsible for designing the bridge so that it meets the needs of the people who will own it and the people who will use it. For the owners of the bridge, the architects must consider the cost and schedule of its construction, the cost of its operation, and its long-term maintenance requirements. For users of the bridge, they must consider the structural integrity of the bridge, its carrying capacity, its relationship to its environment, and its aesthetic appeal. All of these considerations must be reflected in the blueprints produced by the architects. Software architects have similar concerns for product owners and users. A software design must meet the needs of the companies that implement products based on the design, and it must meet the needs of the users of those products. For large software development projects, these concerns are the ones that determine the success or failure of the project. They must be given as much consideration as they would be given in designing a major highway bridge.

A construction company is bound by its contract to build a bridge according to the blueprints provided by the architects, but real-world conditions can require changes. In these cases, the architects are informed of the problems encountered and make changes to the blueprints. Often, the architects visit the construction site to see for themselves how well their specifications are being realized and sometimes spot troubles before the construction crew is aware of them. The relationship between the architects and construction company is an important factor in the success of the project. So too, with software architecture. The architects must be independent of the programming team but have a close working relationship with them. The design produced by the software architects must be seen as the essence of the implementation contract, but it must also be open to change as real-world problems are detected.

Building inspectors also study the blueprints, not to build a bridge themselves, but to ensure that the bridge is actually built according to the specifications. They test the quality of materials and construction as the bridge is being built so that they can certify it meets the requirements of the blueprints. In software development, this role is often played by a systems assurance or quality assurance team. But for the role to be played effectively, the assurance team must measure an implementation against up-to-date specifications that clearly define what was supposed to be built.

Finally, the bridge is opened to traffic, and the driving public decides whether it meets their needs, namely, getting across the river in a safe, timely, cost-effective manner. Similarly, a software product must fulfill the needs of its users.

After thousands of years of experience in constructing bridges, buildings, and other large, complex structures, this process has been formalized in both building codes and legal practices. Clear divisions of responsibility among independent experts are carefully observed. We are only now beginning to understand that this process applies equally well to software projects, especially for large projects. In particular, we are slowly learning that software architecture requires a different skill than programming and should be done by people other than programmers.

The history of DDM architecture

What we call DDM architecture is actually an evolving set of specifications for distributed application services. The story of DDM architecture

begins in the mid-1970s when IBM's Systems Network Architecture (SNA) logical unit (LU) 6.2[4] was being designed. An important feature of LU 6.2 is that it is possible for a program on one system to create a conversation with a program on another system and pass it parameters (effectively, a remote procedure call), then interchange messages with it. In addition to application programs using LU 6.2, its architects saw the possibility of requesting system services and using the resources of remote systems with this mechanism. Indeed, they saw LU 6.2 as the foundation for the development of a distributed operating system, with access to remote files and databases a clear priority.

DDM architecture is actually an evolving set of specifications for distributed application services.

Work began on an architecture that would use LU 6.2 capabilities to provide distributed data management services, and it was thus called DDM architecture. Two things quickly became obvious. First, the interfaces and capabilities of the data management facilities of the participating IBM systems were quite varied and would make the design of a common distributed data management facility difficult, essentially being an exercise in negotiation and standardization. Second, there was no strong demand for LU 6.2 services on IBM large systems at that time because other communications services were in wide use. As a result, work on the DDM architecture languished.

However, circumstances were somewhat different for the IBM System/34 family of midrange computers. No large installed base of users was already employing other communications services, and some users were beginning to install System/34s in multiples for both horizontal growth and to decentralize their processing. These factors made a peer-to-peer communications facility like LU 6.2 desirable to System/34 users and made access to data in remote System/34s crucial. From the initial work that had been done on DDM architecture, a product called System/34 Distributed Data File Facility (DDFF) was created and was announced in 1980. Because it was intended to communicate only with other System/34s, DDFF was able to bypass the problems of nonstandard file interfaces and prove the value of the fundamental idea of DDM architecture; that is, it is useful to be able to request services from remote systems as if they are local.

The DDFF product was carried forward to the successor of the System/34, the System/36*, in 1982. But in the attempt to do this for the System/38* family of computers, the problem of data management standardization again arose because of the different data management interfaces and facilities of the System/38. This time, however, the people interested in solving it were all part of the same IBM programming laboratory, in IBM's Rochester, Minnesota facility, responsible for the System/34, System/36, and System/38 families. A new architecture group was formed in Rochester in 1983 to work with representatives of these systems and to define the syntax and semantics of DDM messages. The result was Level 1 of DDM architecture, which was announced and published in 1986. At the same time, IBM announced DDM products developed by the Rochester laboratory for use with the IBM Personal Computer Disk Operating System (PC-DOS), System/36, System/38, and Customer Information Control System/Multiple Virtual Storage (CICS/MVS*).

The next step in the evolution of DDM architecture was the support of stream-oriented files. Client/server products for the System/36 and System/38 required the ability to store, access, and manage PC-DOS stream-oriented files and hierarchical directories on a System/36 or System/38. This work resulted in Level 2 of DDM architecture, which was published in 1988.

At about the same time, the SAA subset of the record file support of DDM architecture was defined and announced as an SAA Common Communications Support Architecture. As such, IBM also announced its intention to provide Distributed File Management (DFM) products on all four IBM SAA systems: Operating System/2* (OS/2*), Operating System/400* (OS/400*), Virtual Machine/Enterprise Systems Architecture (VM/ESA*), and Multiple Virtual Storage/Enterprise Systems

Architecture (MVS/ESA*). The DFM product effort, led by the IBM laboratory in San Jose, California, has also involved IBM laboratories in Boca Raton, Florida; Boulder, Colorado; Cary, North Carolina; Endicott, New York; Rochester, Minnesota; and Sindelfingen, Germany—truly a worldwide IBM effort.

Level 3 of the DDM architecture, published in 1990, supports IBM's Distributed Relational Database Architecture (DRDA).[5] This, too, was and continues to be an IBM-wide effort. DRDA is based on the System/R research originally done by the IBM Almaden Research Laboratory in San Jose, California. Under the leadership of the IBM Santa Teresa laboratory, also in San Jose, the DRDA design group includes individuals representing IBM Research, the relational database management products of all four SAA systems, DDM architecture, Formatted Data: Object Content Architecture (FD:OCA), Character Data Representation Architecture, and SNA. DDM Level 3 architecture defines messages for requesting relational database (RDB) services, for example, binding Structured Query Language (SQL) statements into RDB packages and subsequently executing those packaged statements. Included in these messages are DDM structures for carrying SQL statements and responses and for carrying FD:OCA data descriptors. DDM also defines how SNA LU 6.2 communications protocols are used by DRDA.

Level 4 of DDM architecture is expected to be published in 1992. For the DFM products, storage management attributes are supported for files, as defined by the IBM Enterprise Storage Management* architecture,[6] as well as support for user-defined attributes for files. For DRDA, two-phase commitment control protocols are defined for application-directed distributed units of work. And for the OS/400 DDM and OS/400 PC Support products, data queuing and system command processing classes are defined.

Clearly, there is a lag in time between the completion of architectural work and the shipment of products. Each of the IBM products implementing some part of DDM architecture is doing so within the context of a complex system development environment, competing priorities, and limited resources. In general, a new level of DDM architecture is announced when at least one implementing product is ready to be announced. In this way, feedback from at least one product on any ambiguities or problems found can go to the architects. The result is that published levels of DDM architecture have been of high quality, with little need for subsequent corrections. Although non-IBM vendors have not been invited to participate in the design of DDM architecture, they have had the benefit of quality specifications without significant delay.

Blueprints by themselves are of little value to end users. What counts to them are products they can use. IBM products using DDM architecture are given in the appendix.

Numerous other companies have also expressed an interest in implementing products that provide connectivity with IBM's Distributed File Management and Distributed Relational Database products. An example of such a product is one under development by Object Technology International for OS/2 and Windows** that supports the development of cooperative applications with OS/400s. Written in Smalltalk/V**, this product supports the development of OS/2 and Windows applications written in Smalltalk/V that use OS/400 services. Among other services, it implements DDM client architecture for accessing and managing record files.

The relationship of DDM architecture to various products is illustrated by Figure 1.

Object-orientation and DDM architecture

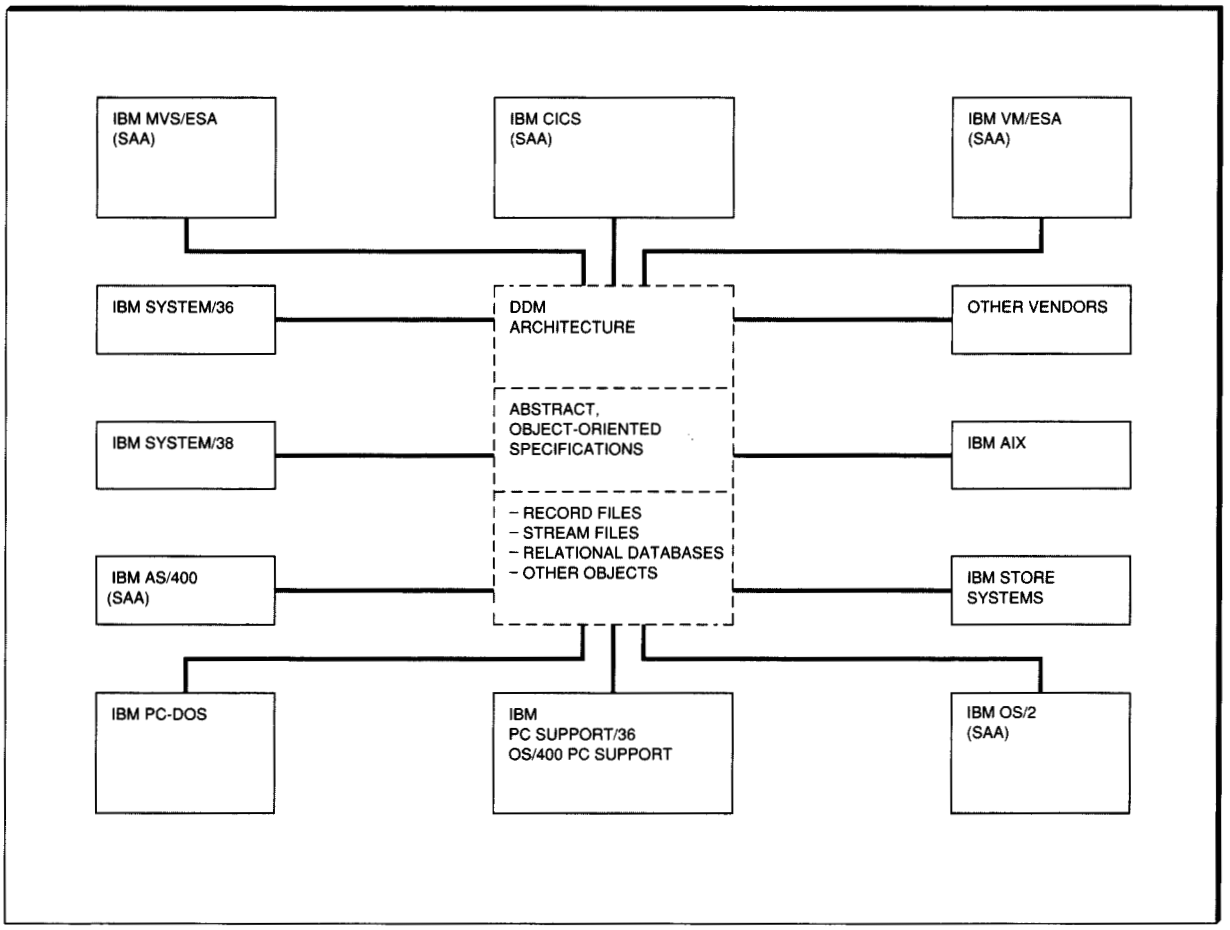
Being *object-oriented* is a frequent claim in the computer industry today, but it is not always clear what that means. Calling all the entities that comprise an architecture *objects* is not sufficient (but is certainly a start). More important is a careful adherence to the fundamental concepts of object-orientation and the design discipline they imply.

Before we look at how DDM architecture uses object-oriented concepts, a few definitions are necessary. From the object-oriented point of view, an object is a self-contained entity that has its own private data and a set of operations to manipulate those data.

Objects are created by special objects called *classes* and are known as *instances* of a class.

As defined by Wegner[7] there are three primary characteristics of object-oriented systems:

Figure 1 DDM architecture and its product environments



1. *Encapsulation*: A technique for minimizing interdependences among separately written modules by defining strict external interfaces. In this definition, the word *object* could be exchanged for *module*.
2. *Inheritance*: A technique that allows new, more specialized classes to be built from the existing classes while retaining all of the characteristics and capabilities of the existing class.
3. *Polymorphism*: A technique that allows the same command (or *message* in object-oriented terms) to be understood by different objects, which respond differently. Often, this goes hand in hand with *dynamic binding*. Polymorphism eliminates much of the control structure traditionally needed to differentiate between objects, such as *case* statements and *if then else* statements.

All entities in DDM architecture are objects, and the architecture consists of a large number of classes to which these objects belong. The reference manual[8] for DDM architecture actually consists of formatted printouts of a large set of classes. These classes each have a name and are arranged alphabetically for ease of reference. Because these terms refer to each other extensively, the reference manual is actually a hypertext document. Extensive cross-referencing information and indexes are also provided.

The variables of a DDM architecture class specify its inheritance, describe the variables of the class

and the variables of its instances, and specify the commands to which the class responds and the commands to which its instances respond. The variables of the class object are encapsulated by the commands of the class, and the variables of the instances of the class are encapsulated by the instance commands of the class.

DDM architecture uses the concept of inheritance to simplify the architectural specifications. For example, the class of MANAGER defines the structure and private data that are common to all DDM managers. The class of FILE, which is a subclass of MANAGER, inherits variables and commands from the class of MANAGER. Since many managers respond to some of the same commands, polymorphism is an inherent attribute of DDM architecture.

The fundamental concepts of object-orientation used in DDM architecture were derived from the book *Smalltalk-80: The Language and Its Implementation*.[9] Although the DDM architects attempt to remain faithful to these concepts, there are a number of important differences between DDM architecture and Smalltalk:

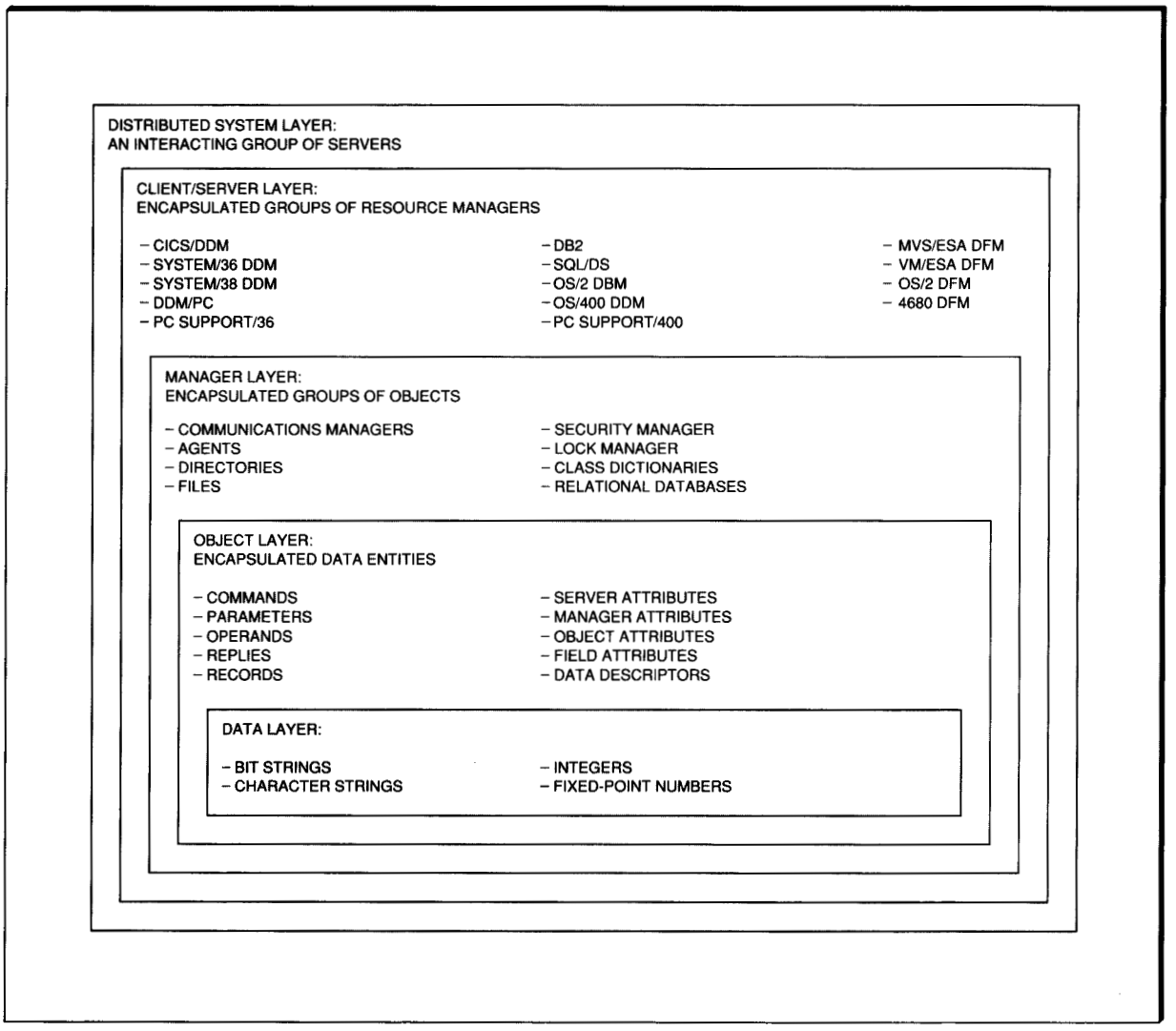
- The DDM architecture class library is not designed for the same purposes as a Smalltalk class library. A Smalltalk class library provides programmers with a computer-aided software engineering (CASE) environment and includes a wide variety of classes that can be reused when creating Smalltalk applications. In contrast, the class library of DDM architecture contains only classes related to the services covered by the architecture. Although DDM architecture can be implemented in Smalltalk, the DDM architecture class specifications are not intended to be a working prototype of the architecture.
- The Smalltalk concept of meta-classes as first-class objects was considered both confusing and of marginal value to DDM architecture. Instead, DDM architecture collapses each meta-class into its corresponding class. That is, each class defines its own variables and its own behavior (with inheritance).
- Because DDM architecture is concerned with the form of the messages that flow between clients and servers, each *command* message is formally defined by its own class object. For example, the *Clear File* message is defined by the CLEAR class. Each class whose instances can be a receiver for this message specifies CLEAR as an instance command.
- Smalltalk defines only one error message: *Does not understand*. It is returned if the class of a receiving object does not recognize a request. This action was not considered adequate given the heterogeneous nature of DDM architecture clients and servers. Therefore, for each command message, DDM architecture defines the set of *reply messages* that can be returned by any methods that implement the command. Each reply message is also defined by its own class object, and all reply messages are subclasses of class RPYMSG, which defines a set of instance variables that they all inherit. DDM architecture defines a separate reply message class for each exception that can occur. These reply messages are then specified for whatever commands can raise each exception.
- Also, because DDM architecture is particular about message formats, each instance variable of a command or reply class is fully typed. That is, DDM architecture specifies precisely what classes of objects can be specified for each variable. Clients are required to create messages within the range of this typing. Servers type check each variable of a command before it is passed on to its receiver.
- Perhaps the most original aspect of the object-orientation of DDM architecture is the division of its objects and their classes into hierarchical layers. Whereas Smalltalk considers all objects to exist in a single, uniform space within a single virtual image, DDM architecture considers them to exist in a multilayer space where the objects in one layer are composed of objects from the next lower layer. This difference is discussed in the next section.

With the exception of the Object Technology International product that is being implemented in Smalltalk/V, DDM products have been implemented using procedural programming languages, such as C, because object-oriented programming languages were not available or because the products had to be integrated with other products. They have, nevertheless, benefited from the clarity, conciseness, and completeness of the object-oriented specifications of DDM architecture.

The DDM architecture framework

The initial services defined by DDM architecture pertain to data management, but these services

Figure 2 The structural layers of DDM architecture



are specified within the context of a framework designed to accommodate the full range of application services typically provided by operating systems. This framework, illustrated in Figure 2, consists of nested structural layers. It is similar to what we see in natural world systems, where molecules are composed of atoms, atoms of particles, and particles of quarks. For DDM architecture, the following five layers are defined:

- The Distributed System Layer consists of one or more distributed systems in a network of heterogeneous computer systems. Each distributed system provides a single system image of operating system services for its client programs. Each is composed of DDM architecture clients and servers.
- The Client/Server Layer consists of the various clients and servers of a distributed system. The clients provide requesting programs with local or remote transparency, and the servers manage and provide access to the resources of a single system. A client or server is composed of DDM architecture resource and service managers.
- The Manager Layer consists of the major com-

ponents of a server. For clients, managers provide local or remote transparency, routing services, communications services, and support services. For servers, managers provide resource access and management services, routing services, communications services, and support services. A manager is composed of DDM architecture objects.

- The Object Layer consists of self-identifying data used by managers, stored by managers, or communicated between managers. An object consists of DDM architecture data elements.
- The Data Layer consists of the data elements that describe objects or represent their value.

The Distributed System Layer. A distributed system is one in which users and programs see a single system image of the services available to them through a network of heterogeneous, networked systems. In DDM architecture, a distributed system consists of clients and servers that interact to process application requests. The clients and servers use whatever communication links and whatever communication protocols are available between them in a peer-to-peer fashion. In this sense, a DDM architecture distributed system is independent of the topology of the network used by its servers. Multiple DDM architecture distributed systems can reside in the same networks, each with clients and servers in some of the same systems. A DDM architecture distributed system is illustrated in Figure 3.

There are two possibilities to be considered when designing a single system image: canonical services and mapped services. With canonical services, a single programming interface is defined for each service, all client programs use that interface to request it, and all servers support precisely that interface for that service. A client program need not be concerned with what server will actually provide the service, and a server need not be concerned with who is requesting it. The design of the messages that flow between the client and server systems is a simple translation of each service interface into a transmittable stream of bytes. Client programs are highly portable between systems but have to be specially written to the canonical interfaces. This approach is generally being taken by the Open Software Foundation Distributed Computing Environment (OSF/DCE**). The presumption is that new client and server products will be developed over time to provide the canonical services and interfaces.

The second possibility for a single system image is the one adopted for DDM architecture: mapped services. Client programs use the interfaces of their local system to request services, regardless of the system that actually provides them. If a request is for a local service, it is directed to the local facility that provides that service. But if it is for a remote service, the client system translates the request into a message designed for that class of service. The remote server translates the message into a request specified through a programming interface of the server. Thus, three programming interfaces are involved in a mapped request for a remote service:

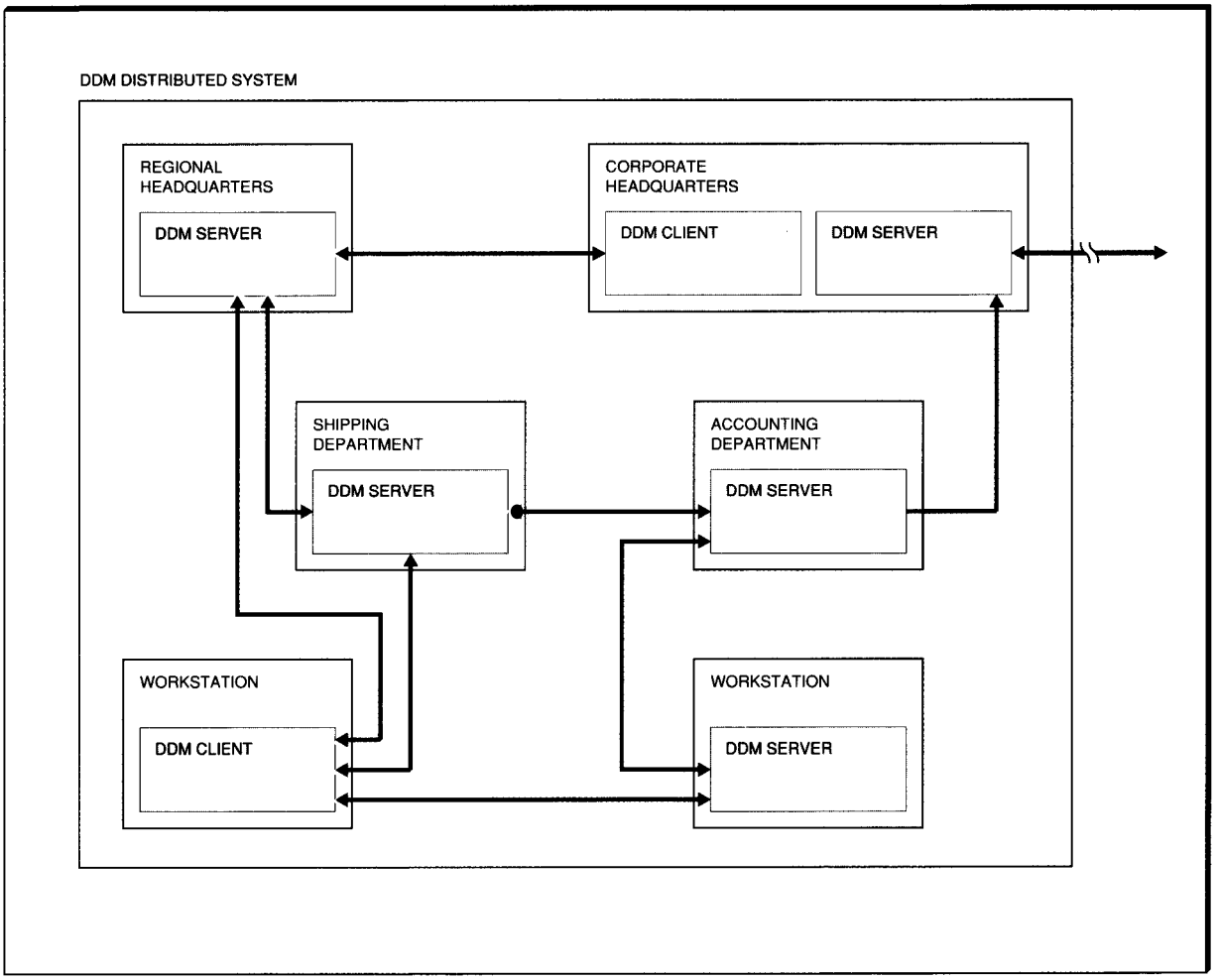
1. The programming interfaces of the client system
2. The abstract programming interfaces defined by messages
3. The programming interfaces of the remote server

Multiple DDM architecture distributed systems can reside in the same networks.

In this way, client programs can request services without concern for what programming interfaces actually need to be used. And conversely, a system can provide services without concern for the programming interfaces actually used by the requesting application program. Any existing or new program that uses a service can be a client of either the local system or of a remote system, but these programs are not portable to another system without conversion to its service interfaces.

Whether canonical services or mapped services are used in the design of a single system image is largely a matter of objectives. Local interfaces can often be mapped to canonical services, and local interfaces can be designed to match the messages of mapped services. In both cases, the syn-

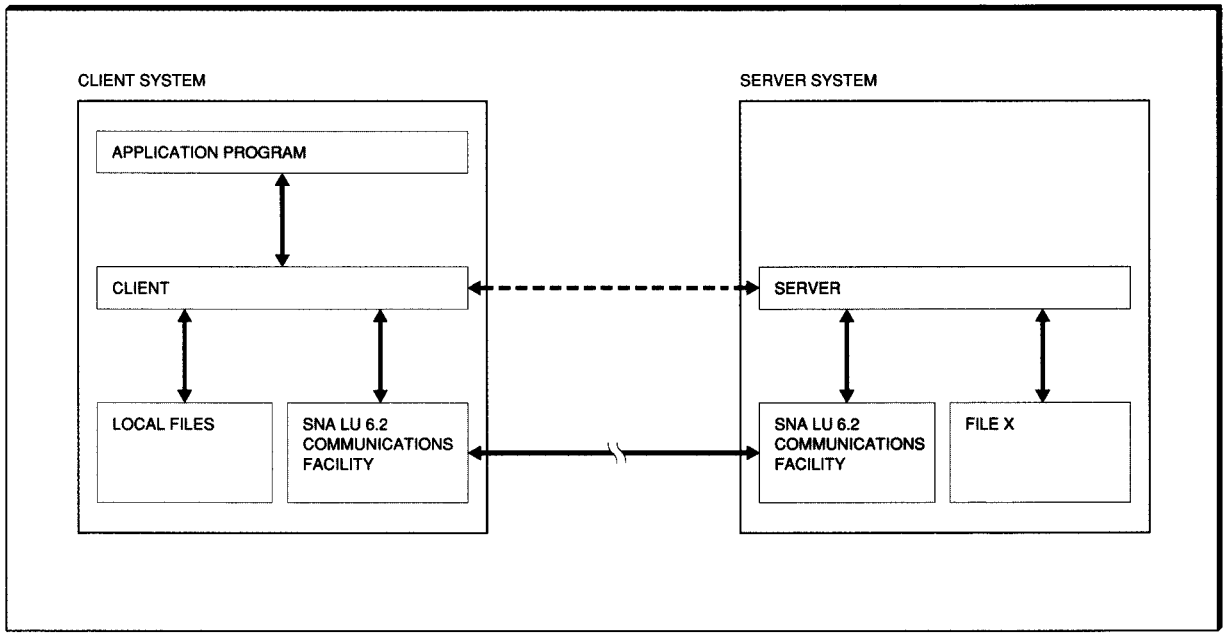
Figure 3 DDM distributed systems and servers



tax and semantics of the services must be carefully and formally defined for both clients and servers. The real issue is whether existing client programs and existing services should be changed to meet the requirements of canonical services. For DDM architecture, the choice of mapped services was dictated by the very large number of existing programs, files, and databases on IBM systems. Ease of migration from single systems to distributed systems mandated the mapped services approach.

A final comment on this subject is that the approach taken affects what services are defined. To define canonical services is largely an academic exercise, based on knowledge of similar services (in the UNIX** environment for OSF/DCE), with attention to consistency and completeness, and certainly with review and approval by peers in the sponsoring body. In contrast, the design of mapped services is largely a standards exercise, requiring participation by representatives of multiple client and server systems. As a mapped service model is designed, each representative must critique it in terms of the local services and interfaces available on his or her system, either finding good mappings or arguing for changes that would allow good mappings. This process is arduous and time-consuming, but it leads to distributed service models that better match the needs of existing client programs and the capabilities of existing system services.

Figure 4 Client/server architecture



The Client/Server Layer. In general terms, a client is a layer of software in one system that routes service requests to the server that owns a needed resource, such as a file, database, printer, or processor. The server can be either local or remote. The client/server model is illustrated by Figure 4. DDM architecture is in the client/server category.

DDM architecture calls clients *source servers* because they are the source of requests for services, and it calls servers *target servers* because they are the target of those requests. This difference in terminology is simply the result of DDM architecture predating what is now called client/server processing. This paper uses client/server terminology to avoid further confusion.

The servers in DDM architecture can be specialized to handle specific resources. For example, one server (CICS/DDM) handles only files, whereas another server (MVS DATABASE 2*) handles only relational databases, and a third (Operating System/400*) handles both files and relational databases. Multiple servers can reside in the same system.

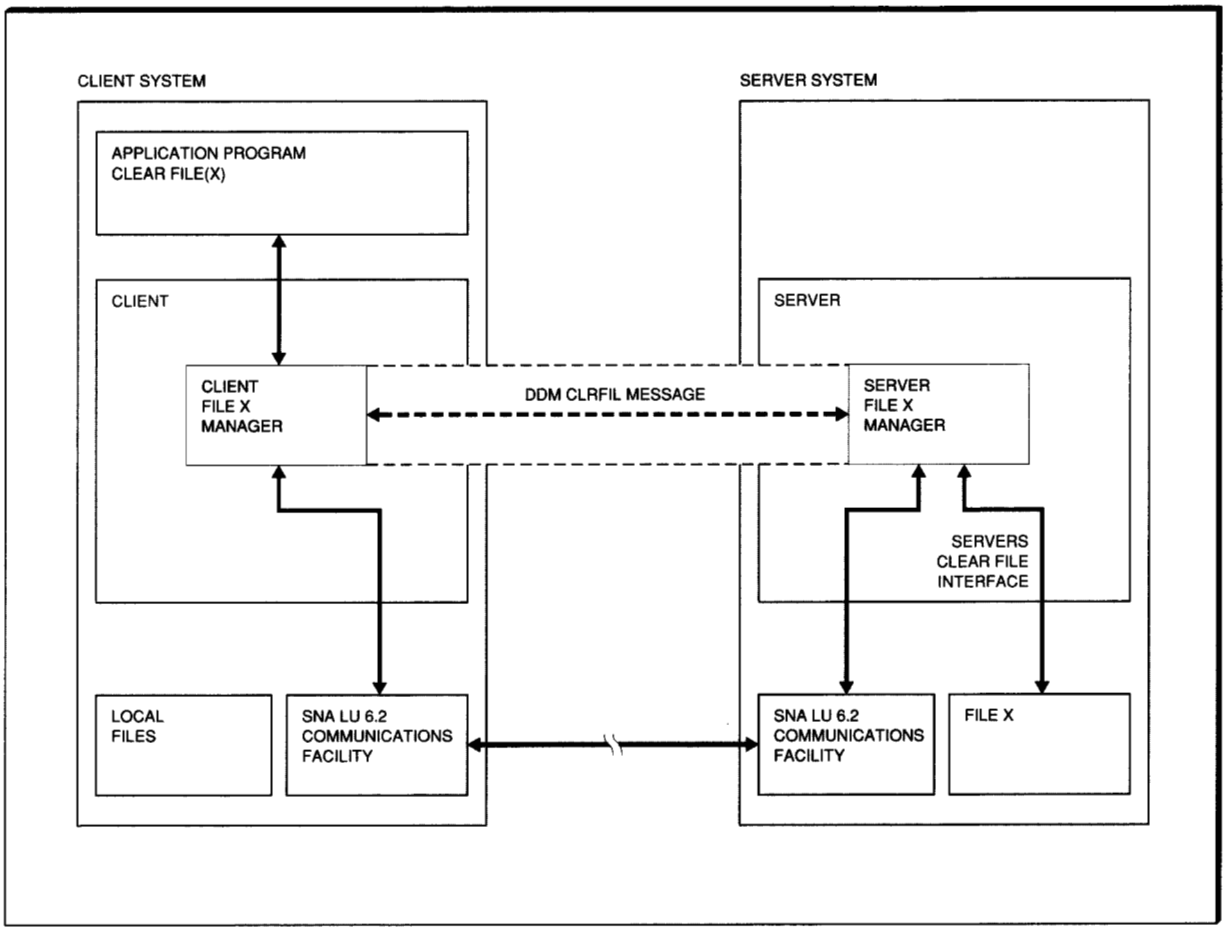
A product implementing DDM architecture can provide both client and server support, or it can be specialized as just a client or just a server. As examples, AS/400 DDM is both a client and a server, the DDM/PC product is only a client, and the CICS/DDM product is only a server.

The Manager Layer. A client or server in DDM architecture consists of a set of entities each of which manages some aspect of the service, such as a file, a database, or a conversation linking the server to other servers. These entities are all called *managers* in DDM architecture. A client or server can have zero or more *instances* of each kind of manager defined by the architecture. For example, each file in a server is an instance of one of the file manager classes in DDM architecture.

If a server supports only a subset of DDM architecture, it has the managers of that subset, plus any managers on which they are dependent. For example, if a server supports only files, it only has instances of the manager classes required by files, but it does not have instances of the managers concerned with relational databases.

The manager classes currently defined by DDM architecture can be categorized as files, databases, queues, access managers, support managers, agents, and communications managers.

Figure 5 File manager request processing



Files. DDM architecture defines file classes for both record-oriented files typically found on large systems and stream files typically found in workstations.

The DDM architecture model of a file consists of information about the file (its attributes), the data content of the file (either records or a stream), and for keyed files, an index over the records of the file. Messages can be sent to a server to create, delete, rename, lock, unlock, or clear files. Further, data can be loaded into or unloaded from a file. Figure 5 illustrates how a request to clear a file of its data is passed from the client file surrogate to the server. The client file surrogate knows where the real file is located and how to access it.

Some of the attributes of a file must be specified by the requester when the file is created to determine its size, capabilities, and status in the system. Other attributes can also be specified to provide application or user information about the file. And still other attributes, such as the date the file was last changed, are managed by the file system. For each file class, DDM architecture allows certain attributes to be retrieved from the file and certain attributes to be changed.

Record-oriented files. A record-oriented file consists of a set of slots in which application-defined units of data, called records, can be stored. In some files, all records are the same length and have the same data structure, whereas in other files, records can vary in length and have varying,

sometimes quite complex, structures. Most file systems contain no information about the data content of the records in files. A record-oriented file simply stores, and provides access to, the records of a file as whole units. DDM architecture record-file classes also treat records in this way, but metadata about records would allow many additional services to be provided, as discussed in References 10 and 11.

DDM architecture defines several subclasses of record-oriented files in order to better match the semantics of existing file systems:

- *Sequential files*: These files simply consist of a set of numbered slots with no relationships defined among them. The records in these files can be accessed relative to the position of an access method cursor (next or previous slot) or randomly by slot number.
- *Direct files*: These files consist of a set of numbered slots, but there is an application-defined relationship between the data contents of the records in the slots and the number of the slot. The records in these files can be accessed relative to the position of an access method cursor (next or previous slot) or randomly by slot number.
- *Keyed files*: These files consist of a set of numbered slots and an index that associates the values of the key fields of each record with slot numbers. In addition to relative and random accessing by slot number, these files can be accessed by using the index. The record with a key value that is next or previous, relative to the key value of the current record addressed by the access method cursor, can be accessed. Any record can be randomly accessed by specifying its key value.
- *Alternate index files*: These files consist of only an index that associates alternate fields of the records of a base file with their slot numbers in the base file. They provide an alternate access path to the records in the base file. The base file can be a direct file, a sequential file, or a keyed file. The access path of the index maintains a specified ordering relationship for these fields.

Stream files. These files consist of a single stream of bytes. Applications can access or update any part of the stream by specifying the starting position and length of a portion of the stream. As with the record file classes, the DDM architecture stream file class has no knowledge of the contents of the stream.

File access managers. DDM architecture file access manager classes each define a model of how a file can be used by a single local or remote user. Here too, the key problem is that each host system includes access managers with unique interfaces. The DDM architecture file access managers each define a single, consistent set of interfaces for working with a file that can be mapped to and from the interfaces of host systems. An example of a file access manager is the *Relative by Record Number Access Method* of DDM architecture.

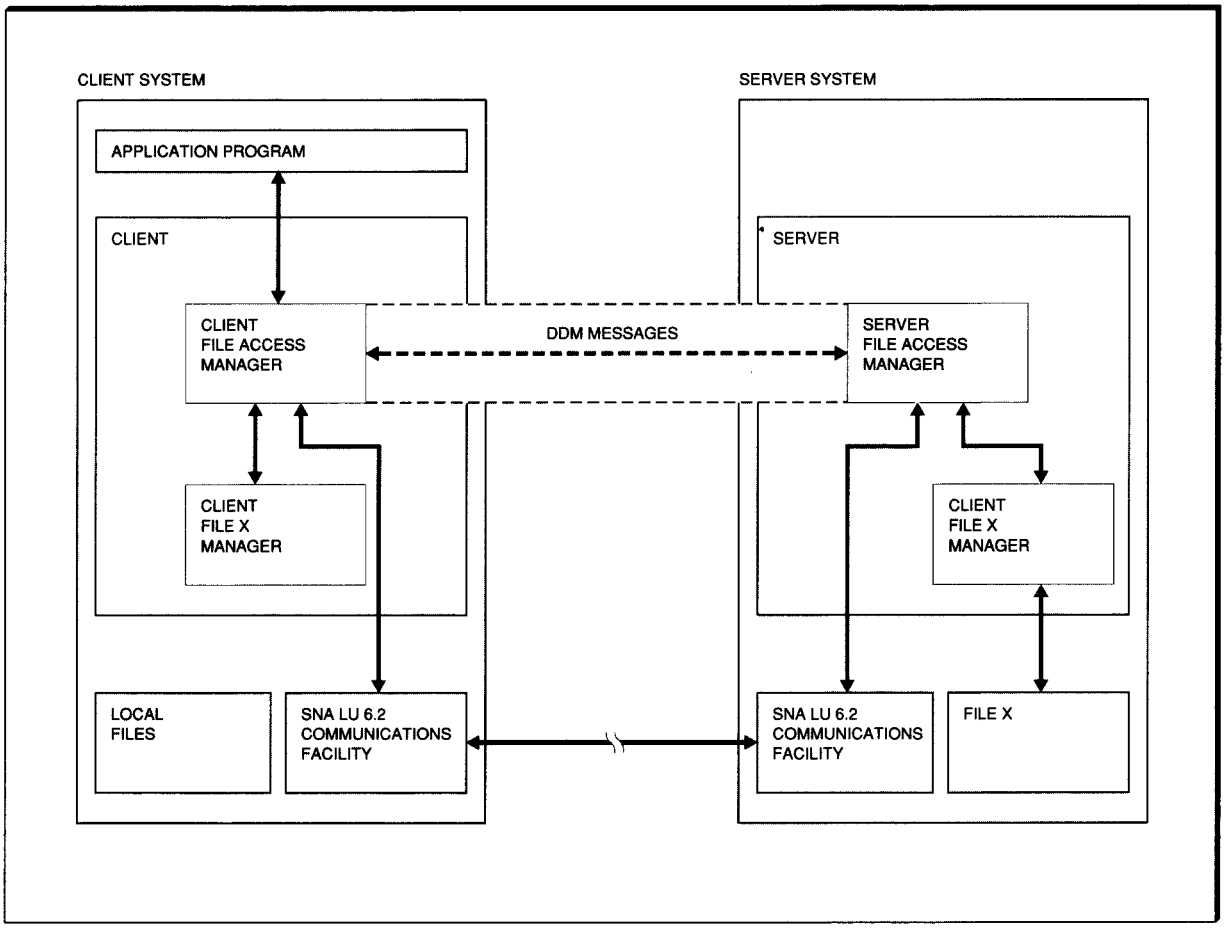
Figure 6 shows how an application accesses a remote file. When the application program opens the file, an instance of a client file access manager is created. The client file access manager obtains the location of the file from the client file surrogate and sends a request to create an access manager to the server. This server opens a path to the real file. The bound path remains in existence until the application closes the client access method.

Figure 7 shows multiple application programs in the same server accessing the same remote file. Note that they share the use of the same communications facility.

Relational databases. Relational databases (RDBs) have been implemented on many systems, each with its own interfaces and internal structures. But they also have much in common; in particular, support for Structured Query Language (SQL). Distributed Relational Database Architecture (DRDA) is the SAA CCS architecture concerned with distributed relational databases. DRDA is a composite architecture, as shown in Figure 8.

- DDM architecture defines the DRDA model of relational databases, the DRDA model of an SQL access manager, the messages and replies transmitted between a DRDA client and server, the means by which SQL statements and Formatted Data:Object Content Architecture (FD:OCA) descriptors are transmitted, and the ways in which SNA communications protocols are used.
- SNA LU 6.2 is the communications facility used by DRDA.

Figure 6 Access manager request processing



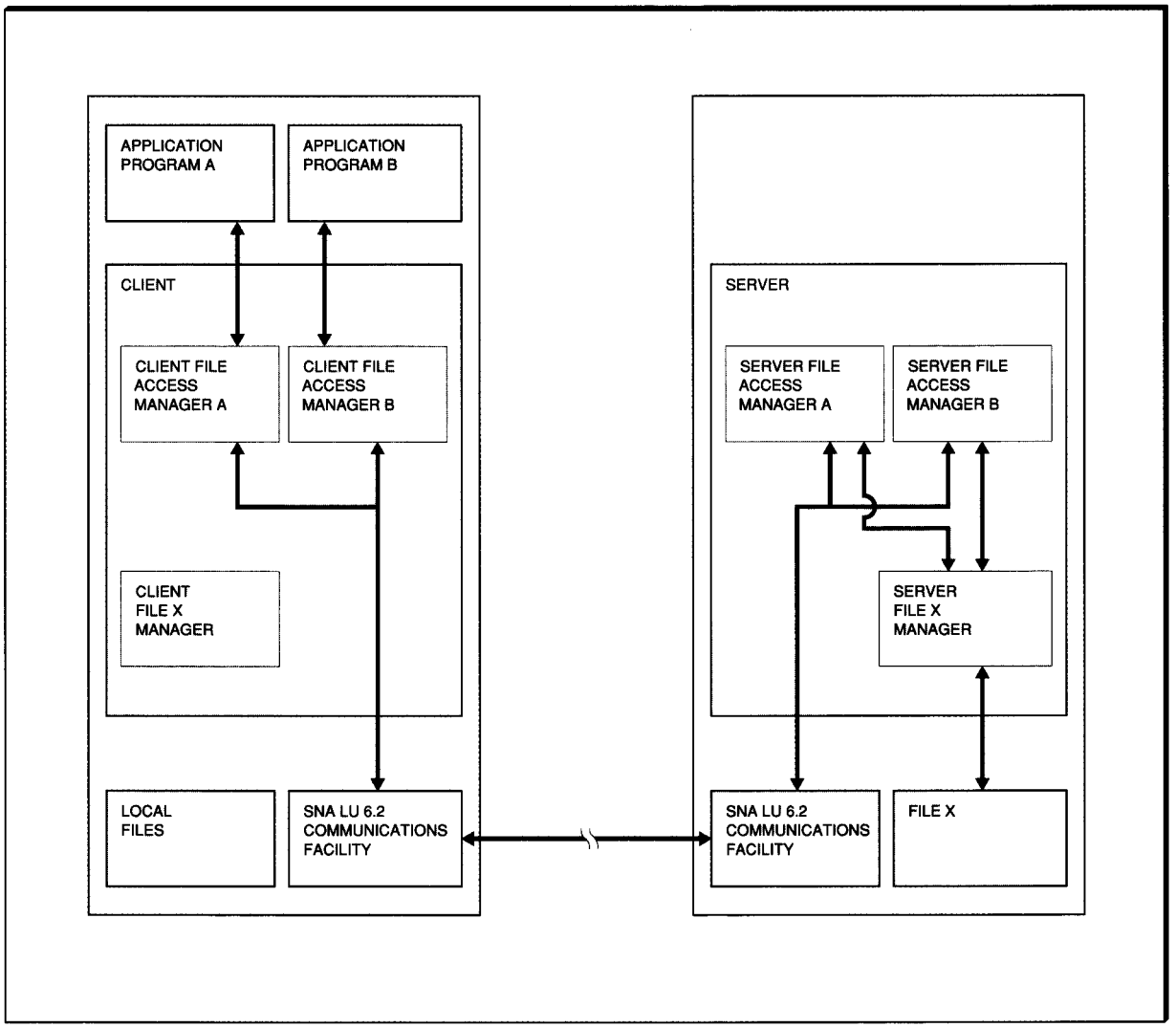
- FD:OCA is used by DRDA to describe data.[12] The data are either input data to an SQL statement to be executed or a result of executing an SQL statement.
- Character Data Representation Architecture (CDRA)[13] is used by FD:OCA to tag character data with its encoding scheme (such as ASCII and EBCDIC).

All database operations performed by the RDB occur within a logical unit of work. A logical unit of work identifies the processing performed to take a set of recoverable resources from one state (A) to another (B) in an atomic fashion. If the processing succeeds, the new state of the recoverable resources is B, and if the processing fails, the recoverable resources return to state A. The changes made by the application during the desired transition from state A to state B become permanent (committed) when the processing succeeds, and the changes are undone (rolled back) when the processing fails.

Using normal SQL interfaces, applications and users can interact with local or remote relational databases. Consider the following types of support for distributed relational databases:

- User-assisted distribution, in which users are involved in extracting data from an RDB, moving the data to another system, and then loading the data into their RDB.
- Remote request, in which each request for serv-

Figure 7 Multiple file access processing



ices is independently sent to a remote RDB and executed, and the results are then returned. Each request is an independent unit of work.

- Remote unit of work, in which related requests for services from a remote RDB are executed within a single unit of work whose effects on the RDB can then be committed or rolled back.
- Distributed unit of work, in which related requests for services from multiple RDBs, local or remote, are executed within a single unit of work, whose effects on all of the RDBs can be committed or rolled back.
- Distributed requests, in which each request within a distributed unit of work can result in interactions with multiple remote RDBs.

Initially, DRDA defines support for a remote unit of work. The DDM architecture model for performing these operations is independent of the hardware architecture, the operating system, and the local RDB interfaces and facilities of either the client system where the application executes or the server system where the RDB is located. The

model consists of the SQL Application Manager (SQLAM) and the relational database manager.

DDM architecture defines a single-system model of the relationships between an application and its local RDB. This model is illustrated by Figure 9. In this model, the application uses the SQLAM to communicate with the local RDB. The SQLAM uses other system services, such as directories and the security manager, in establishing a connection with the RDB.

This model is extended to distributed processing as shown in Figure 10 by allowing the processing of the SQLAM to be split between the client and server systems. Communications between the client and server SQLAMs are provided by agent and communications managers of the DDM architecture that are themselves functionally split between the client and server systems. The client system and the server system can be of different types. The individual DDM architecture implementations can be designed in whatever manner best utilizes their system environment.

The process for obtaining access to a remote RDB is as follows. A client application obtains RDB services by interfacing with a client SQLAM. The client SQLAM establishes a connection to the remote RDB. Client directory services are used to determine the network location of the RDB being contacted. The client SQLAM then creates a client AGENT and sends the Access Relational Database (ACCRDB) command to that agent for transmission to the server system. When the server system processes the ACCRDB command, a connection is established between the server SQLAM and the server RDB. In general, the database requests sent by the application to the client SQLAM are routed to the server SQLAM for processing. However, the client and server SQLAMs cooperate in the processing of many commands. For example, during query processing, data are buffered by both the client and server SQLAMs to optimize the use of the communications facilities. The server SQLAM manages all accesses to the RDB by a single remote requester. The server SQLAM is bound between the server agent and a single RDB. DDM architecture commands are then received by the SQLAM for processing by the bound RDB.

The client and server SQLAMs exchange the FD:OCA data representation specifications when the connection is established with the relational

Figure 8 The architectures used by DRDA

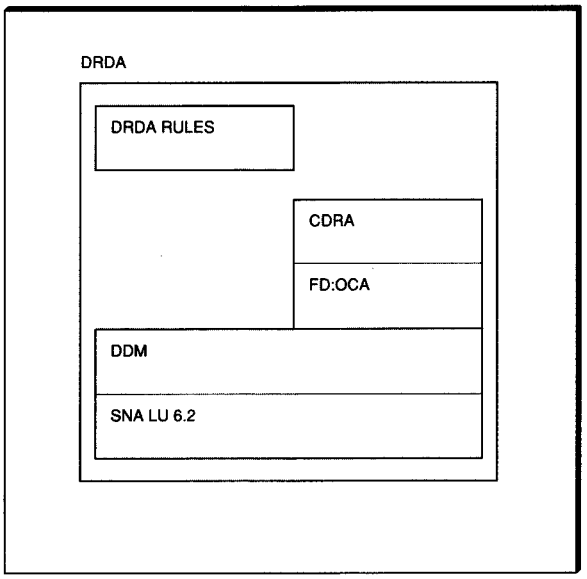


Figure 9 The single-system model of RDB access

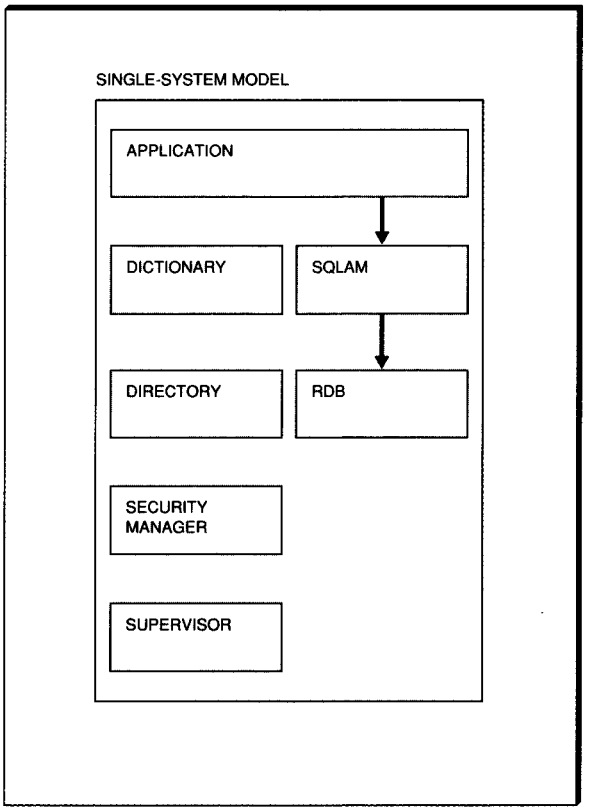
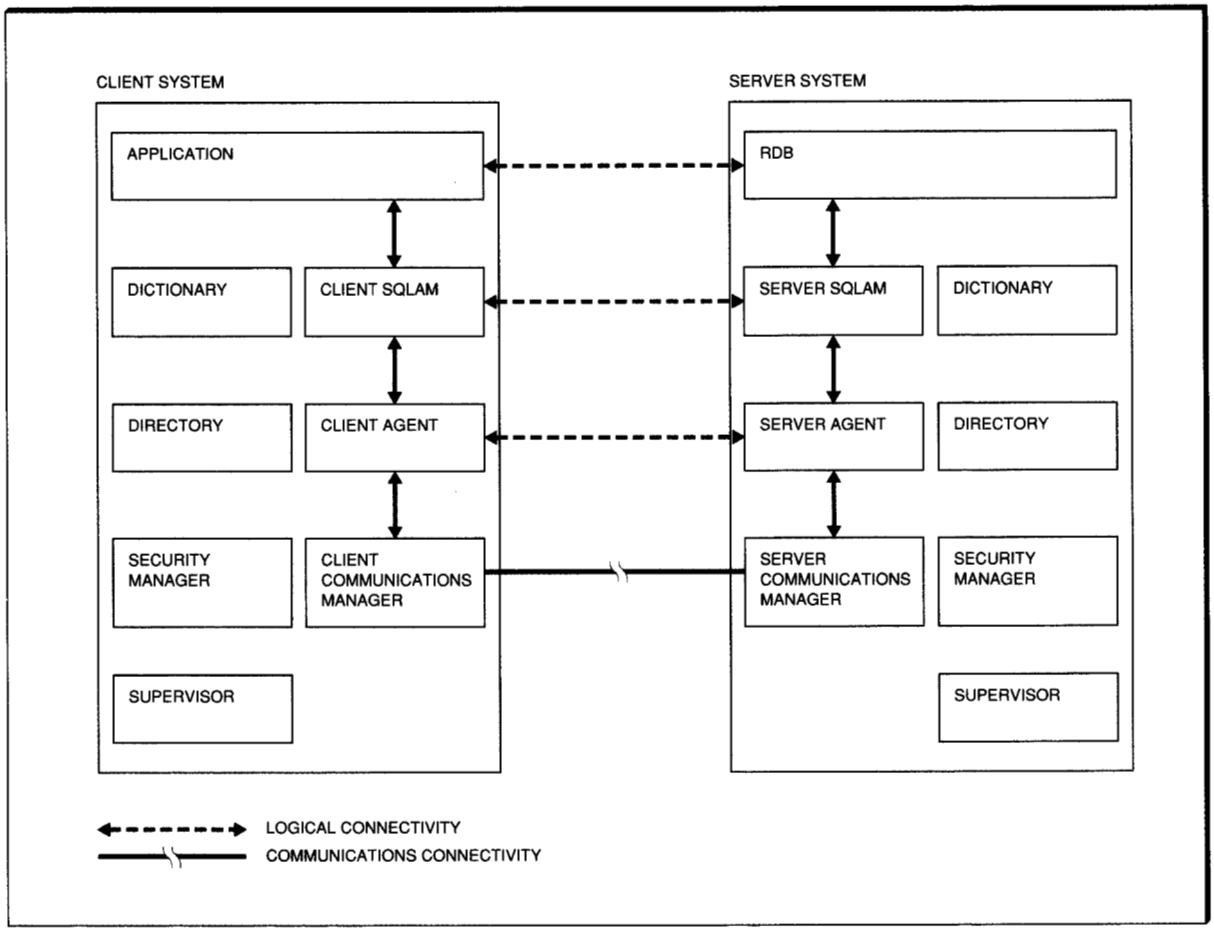


Figure 10 The remote system model of RDB access



database. These specifications provide the information necessary for the SQLAMs to determine how their counterparts represent data, for example, character data or floating-point data. The SQLAMs can then decide whether any data conversions are necessary. When necessary, the data conversions are performed by the SQLAM (client or server) that receives the data.

When an application runs, a call is made to the client SQLAM for each executable SQL statement in the application. The type of call to the SQLAM depends on the RDB function to be performed as determined by the precompiler from the SQL statement. The client SQLAM builds a DDM architecture command to send to the server SQLAM based on the type of call received from the application.

Queues. Queues are important to many applications because they allow either data or requests for services to be passed asynchronously from one program to another. DDM Level 4 architecture defines three subclasses of queues:

- First-in-first-out queues that act as an asynchronous pipe between the enqueuing and dequeuing programs
- Last-in-first-out queues that act as a push-down stack
- Keyed queues that act as a fan-out mechanism,

allowing different programs to dequeue only selected entries

As with files, the contents of the entries on a queue, called records, are application-defined and can be of any format or complexity. The instance commands defined by DDM architecture allow a queue to be created, deleted, or cleared in a remote server. The attributes of a queue can be retrieved, and selected attributes can be changed. Records can be added to a queue or received from a queue. When receiving a record from a queue, the client specifies the amount of time to wait before timing-out, if there is no entry on the queue.

Support managers. These managers support the use of the resource and access managers by multiple local and remote users.

Supervisor. Every client and server has a single instance of a manager called the *supervisor*. In the current levels of DDM architecture, the supervisor is only responsible for exchanging information between clients and servers about their class name, product release level, and the architectural levels of the manager classes supported.

Class dictionaries. When two people speak to each other in the same language, they can understand each other because they share common words, common syntax, common protocols, and common concepts. So too with distributed systems. In DDM architecture, the common words exchanged between clients and servers are instances of the command, reply, and data classes. The common syntax consists of the ways in which DDM architecture allows these objects to be related to each other. The common protocol consists of the rules in DDM architecture governing when each server can send messages. And the common concepts consist of the semantics of the DDM architecture manager models in response to predefined commands.

With the exception of protocols, which are the responsibility of communications managers, the syntax and semantics of messages in DDM architecture are defined by special objects, called *classes*, that reside in managers called *class dictionaries*. To be able to communicate with each other, the client and the server must each have a copy of the same class dictionaries.

Security manager. When a local user logs onto a system, the user's rights to use the system are validated by the security facilities of the system. When that user then attempts to use a system resource, such as a file or database, his or her authorization to use the resource in the way in which an attempt is being made to use it is also validated. Each type of system has its own security facilities, each with its own model of security and its own programming interfaces to that model. Among these interfaces are those that allow a user to be authorized to the system as a whole, and those that allow authorizations to specific resources to be granted, revoked, and shared with other users. Examples of security facilities are the Resource Access Control Facility (RACF*) on MVS/ESA and VM/ESA, the security manager of OS/400, and the GRANT/REVOKE functions of relational database managers.

Validation of authorizations must also be performed when a remote user attempts to use a DDM architecture server or one of its resources. When an SNA LU 6.2 conversation is used for communications, the user identity and password of the requester are passed to the communications facility of the server system for validation. This process occurs before any part of a DDM architecture server is invoked, thereby guaranteeing that all remote requesters are authorized to the server. Although no interfaces are defined by DDM architecture between SNA LU 6.2 and the security manager, their existence and use is assumed. Any security violations are reported back to the client by SNA LU 6.2 messages.

But not all communications facilities provide this level of security. For example, if Transmission Control Protocol/Internet Protocol (TCP/IP) is used, a communications manager for TCP/IP would have to pass the user identity and password in special messages and then call the security manager to validate the requested authorization. In general, communications managers must make up for any deficiencies in the communications facilities they use.

Similarly, when a DDM architecture command is received by a server, the remote requester's rights to issue the command and to use the resources it requests are validated by the security manager. Here too, no interfaces are defined by DDM architecture for performing these validations. The use of local security facilities are assumed. However, DDM architecture does define a

variety of messages for reporting any authorization violations back to the client.

In the current levels of the DDM architecture, the security manager is essentially just a stub that represents whatever security facilities are available on the local system. No DDM architecture messages have yet been defined for working with

Files that are created, deleted, renamed, or moved are addressed by techniques used in ECF and AFS.

or modifying the authorizations of users to server resources. All such changes to user authorizations must be performed by logging onto the system that owns the resource. Clearly this is an inconvenience to users, and clearly, supporting these services would be a desirable enhancement to DDM architecture.

Directories. When an application accesses a local resource, it does so by specifying its name, according to the naming scheme of the local system. This name is used to search the directories of resources maintained by the local system to obtain addressability to the resource. Each type of system has its own model of directory services and its own programming interfaces for using them. For example, MVS/ESA has a catalog that supports multipart names, Virtual Machine/Conversational Monitor System (VM/CMS) has minidisks, OS/400 has libraries, and OS/2 has hierarchical directories.

But how can these directories be extended to include resources that are actually located in other systems and have names that are foreign in syntax to the local directory services? Several answers to this question are available among the IBM products that have implemented DDM architecture.

In the System/36 DDM product, a side directory called the *Network Resource Directory* (NRD) was developed in which the information required to locate a remote file can be placed. Names in the System/36 are simple eight-byte tokens, and the system maintains only a single directory of all such names. The NRD is really just an extension of the system directory such that there is still only one name space in the system. If a name cannot be found in the system directory, the NRD is searched, and if the name is found, the following information is available about the file:

- The real name of the file on the server system
- The network address of the DDM architecture server
- Other communications-related parameters

Thus, it is possible for the client to specify the name expected by the server in the DDM architecture commands to open and access the file, while a System/36 name is used as an alias in the client. An NRD entry is actually a surrogate for the remote file, or, in DDM architecture terminology, it is a client manager of class FILE.

A similar approach was taken with the System/38 and the AS/400, except that these systems support multiple user *libraries* and allow a remote file to be addressed from any of them. In these systems, a special type of system object, called a *DDM File*, is created in a library to contain the real name of the file and communications information. These objects also act as file surrogates.

In contrast, the DDM/PC and PC Support products took the view that they were primarily acting as clients attached to known hosts. Therefore, it was only necessary to redirect requests to the correct server. The name specified by an application program is then just a *drive letter*, indicating the remote host system, followed by the server system name. Although the concept of a client file surrogate is a bit vague in this approach, it is nonetheless implicit.

These approaches all serve to provide addressability to a remote file. It is a programmer's or administrator's responsibility to set up conditions correctly so that needed remote files can be located. But such approaches clearly have limits.

First, these approaches ignore the problems that arise because new files are created and old files are deleted, renamed, and moved to different servers for a variety of application reasons. Except for relatively static applications, maintaining file surrogates in the servers of a distributed sys-

tem can quickly become a major usability problem. A variety of solutions have been devised for dealing with this problem:

- The IBM Enhanced Connectivity Facilities (ECF) product, which implemented a client/server architecture, provided name-mapping services. Given that a client workstation could only be attached to a limited number of host servers of known types, it is possible to map client names to host names for a broad range of names. The OS/2 name *a:FLOWERS.SCR*, for example, can be mapped to the VM/CMS name *FLOWERS SCRIPT A*. But this approach clearly has limited scope.
- The Andrew File System (AFS)[14] has taken a more comprehensive approach. AFS provides hierarchical directories that allow the name space to be of indefinite size and depth. This name space is mapped over all AFS file servers giving all clients the ability to locate any Andrew file. Communications between AFS clients and servers, along with caching and other optimizations, are required to provide directory services with good performance. In broad terms, the same approach is also being taken by Open Systems Interconnection (OSI) directory services and the X.500 network interface standard. By making the assumption that the lowest levels of a qualified name actually specify a local name, this approach allows the local directory services of each system to become a part of the encompassing distributed, hierarchical directory.

A second problem with the approaches taken by early DDM architecture products is that it is not possible to work with the directories on remote servers; that is, to list their entries, create and delete subdirectories, or establish a current directory to shorten the length of transmitted names. Given the range of directory systems found on IBM systems, it is clear that many of these services cannot be supported. However, it is possible to adopt a general hierarchical model of directories and allow each server to support whatever common services it can. In Level 2 of DDM architecture, such a hierarchical directory model was added. Because transparent support of PC-DOS and OS/2 interfaces was considered crucial to the PC Support products, the DDM architecture directory model and its capabilities were based on those of PC-DOS and OS/2. This still does not provide the level of distributed directory support that has been pioneered by AFS and should be added to DDM architecture. Clearly it is needed.

Lock manager. When an application program opens a file, there is always the possibility that some other application program has already opened it or will open it while the first program is still using it. The potential exists, therefore, for programs to interfere with one another's use of the file and to corrupt one another's data. To avoid these problems, each system provides some level of concurrency control that serializes the use of a file or parts of a file by multiple applications. In general, control is in the form of *locks* that are requested on a file prior to its use and that are released afterward.

If one client has a lock on a file and another client requests a lock, the requested lock may or may not conflict with the locks already held. If there is no conflict, the lock is granted; otherwise, the requester must wait for the lock. But waiting for locks leaves open the possibility of deadlocks occurring. To prevent deadlocks, DDM architecture requires lock requests to time-out, that is, to fail after a period of time determined by each server.

The lock manager of a server is the DDM architecture abstraction of local system concurrency control facilities. Whereas relational databases provide their own concurrency control for their internal resources, files and other resources typically depend on a system concurrency control facility.

Recovery manager. It is one thing to access and manage remote files and databases, but quite another to do so in support of complex application transactions that can update multiple files and databases in a variety of systems. Each transaction must be completed successfully or all of its effects on all files and databases must be backed out. No data can be allowed to remain in a file or database in a partially updated, inconsistent form.

The recovery manager of DDM architecture is based on whatever local facility each system provides for transaction commitment or rollback. No special flows are defined in DDM architecture for these operations. Instead, these operations are managed by the sync point manager of the communications facility. The purpose of the recovery manager is to allow the semantics of recovery

operations on files and databases to be correctly defined in the architecture.

System command processor. Each system on which a DDM architecture server product is implemented provides many more services than are available through DDM architecture messages. These services are requested through command language statements unique to that system. For example, through AS/400 commands, operations on jobs, spooling queues, and user profiles can be requested, none of which is yet covered by DDM architecture. Other systems have many similar services, but there is no common command language syntax for requesting them. DDM Level 4 architecture allows a client to submit commands, in the syntax of the command language of a specific system, to a remote server. A client system command processor sends the command to the system command processor of the remote server for execution.

Here DDM architecture provides a common model of system command processing but not of the execution of any particular commands. It is certainly contrary to the concept of a single system image but is of great use when a client knows what type of server it is using, which is often the case. As products identify a need for a single system image in a given area, DDM architecture can be formally enhanced to meet that need.

Agents. As shown in Figures 11 and 12, agents have several functions, but they can be viewed primarily as message routers. In a client, an agent routes a request for services to a communications manager and routes responses back to the requesting resource or access manager. In the server, an agent routes requests to a resource or access manager and routes responses back to a communications manager.

Agents also represent an application program within a DDM architecture client or server. A client agent keeps track of the access paths that have been opened by its application to various resources and knows which communications managers have outstanding requests pending responses. A server agent also keeps track of paths opened by its application and represents its application in regard to security and concurrency considerations. Together, they perform various cleanup and recovery functions in cases of communications failures or failures in the application.

Communications managers. When one server communicates with another server, the programming interfaces provided by local communications facilities must be used to initiate communications, send messages, receive messages, and eventually to terminate communications. The programming interfaces defined for SNA LU 6.2 communications facilities, for example, are rich in options, supporting a wide variety of communications requirements, and can be used in many different ways. The designers of a distributed application are free to use whatever communications capabilities they want. But to achieve connectivity among DDM architecture clients and servers implemented in a variety of products, communications must be tightly and specifically defined. That is, only certain communications capabilities can be used, in only predefined ways, in support of predefined protocols. Otherwise, the client and server simply do not understand each other.

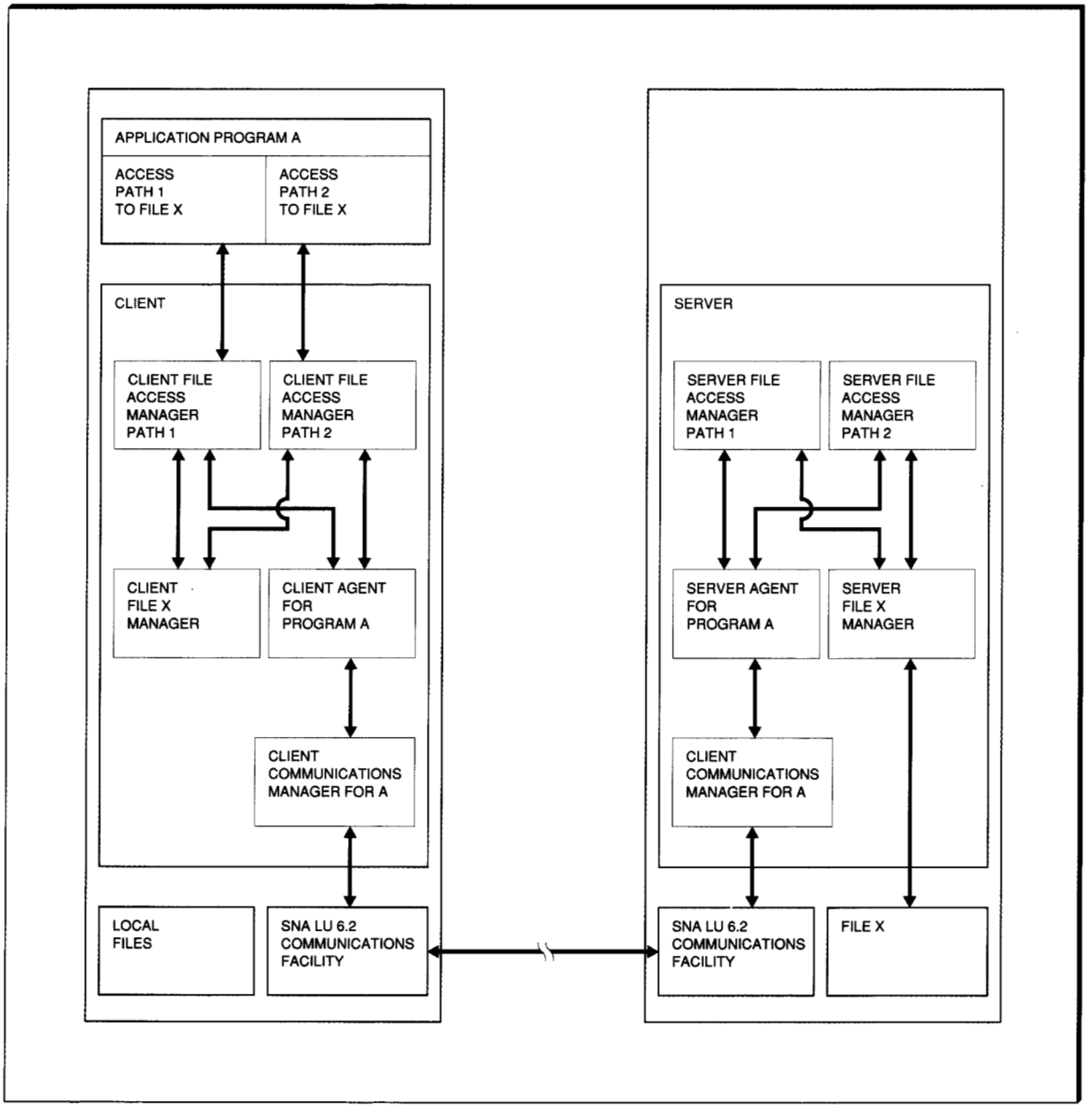
Two principles that have guided DDM architecture communications usage are the following:

1. Only a small number of protocols are necessary to support the communications requirements of a wide range of distributed services.
2. If each protocol is kept relatively simple, it can be mapped to a wide variety of different communications facilities.

The DDM architecture communications managers that have been defined are the following, but this should not be viewed as an exhaustive list.

- SNA LU 6.2 conversational: This communications manager defines how the basic DDM architecture conversational protocol should be implemented when using SNA LU 6.2 conversations. It is a straightforward *I speak and you listen, and then you speak and I listen* protocol. The client sends commands and data to the server; the server executes the commands and then returns reply messages and data to the client.
- SNA LU 6.2 multitasking: This communications manager also defines how the basic conversational protocol is implemented when using SNA LU 6.2 conversations but adds the ability to multiplex concurrent conversations for several tasks onto a single SNA LU 6.2 conversation.

Figure 11 Agent routing to resource and access managers



Products are free to define whatever additional communications managers are required for the network protocols they have available. For example, DDM architecture communications could be implemented using the Remote Procedure Call (RPC) facility of the Open Software Foundation's DCE instead of SNA LU 6.2.

The Object Layer. Although all entities of DDM architecture are objects (that is, instances of classes), the entities in the Object Layer are called objects because they closely resemble the objects found in object-oriented programming environments with dynamic binding, such as Smalltalk. In this layer, an object is a self-identifying

Figure 12 Agent routing to communications managers

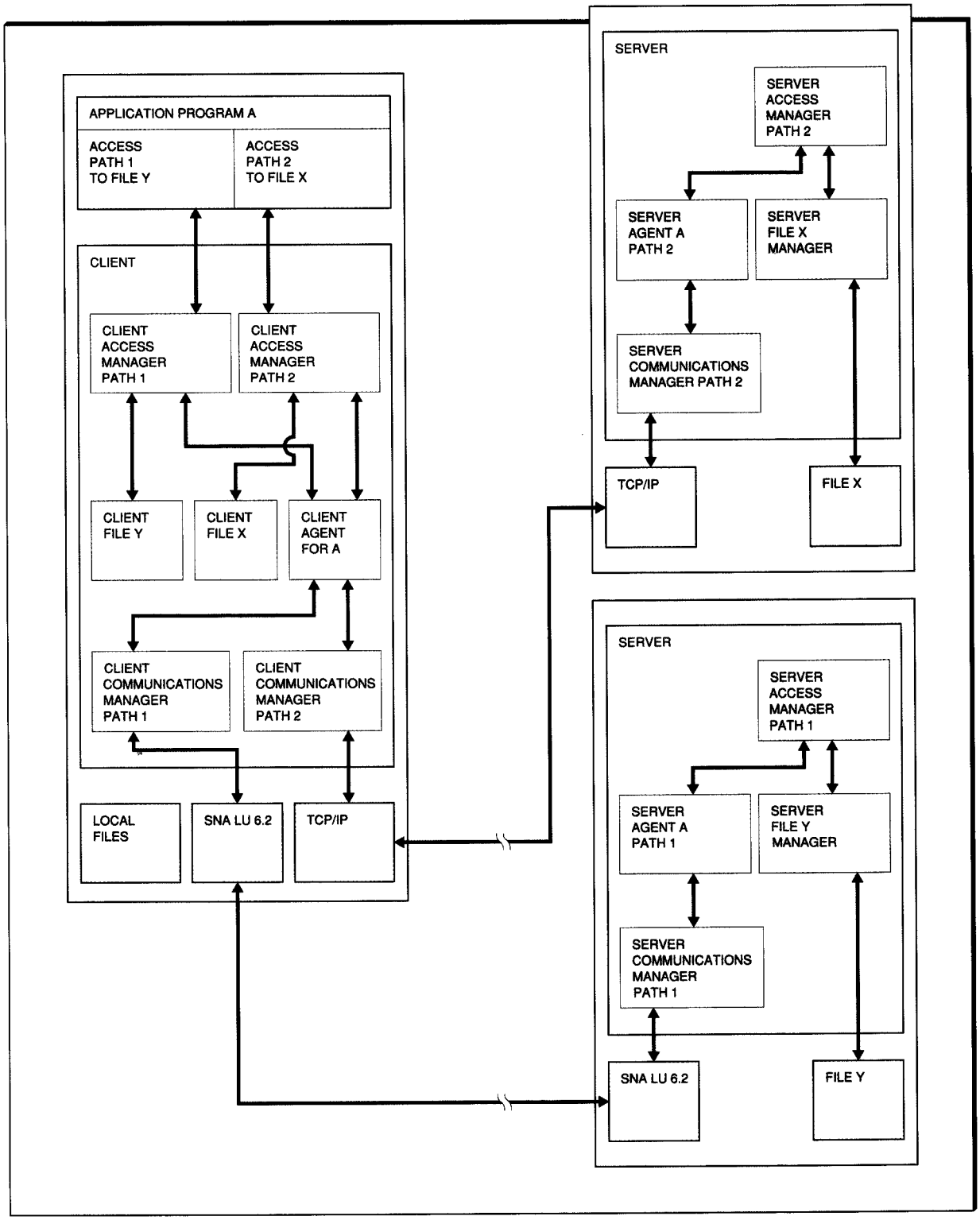
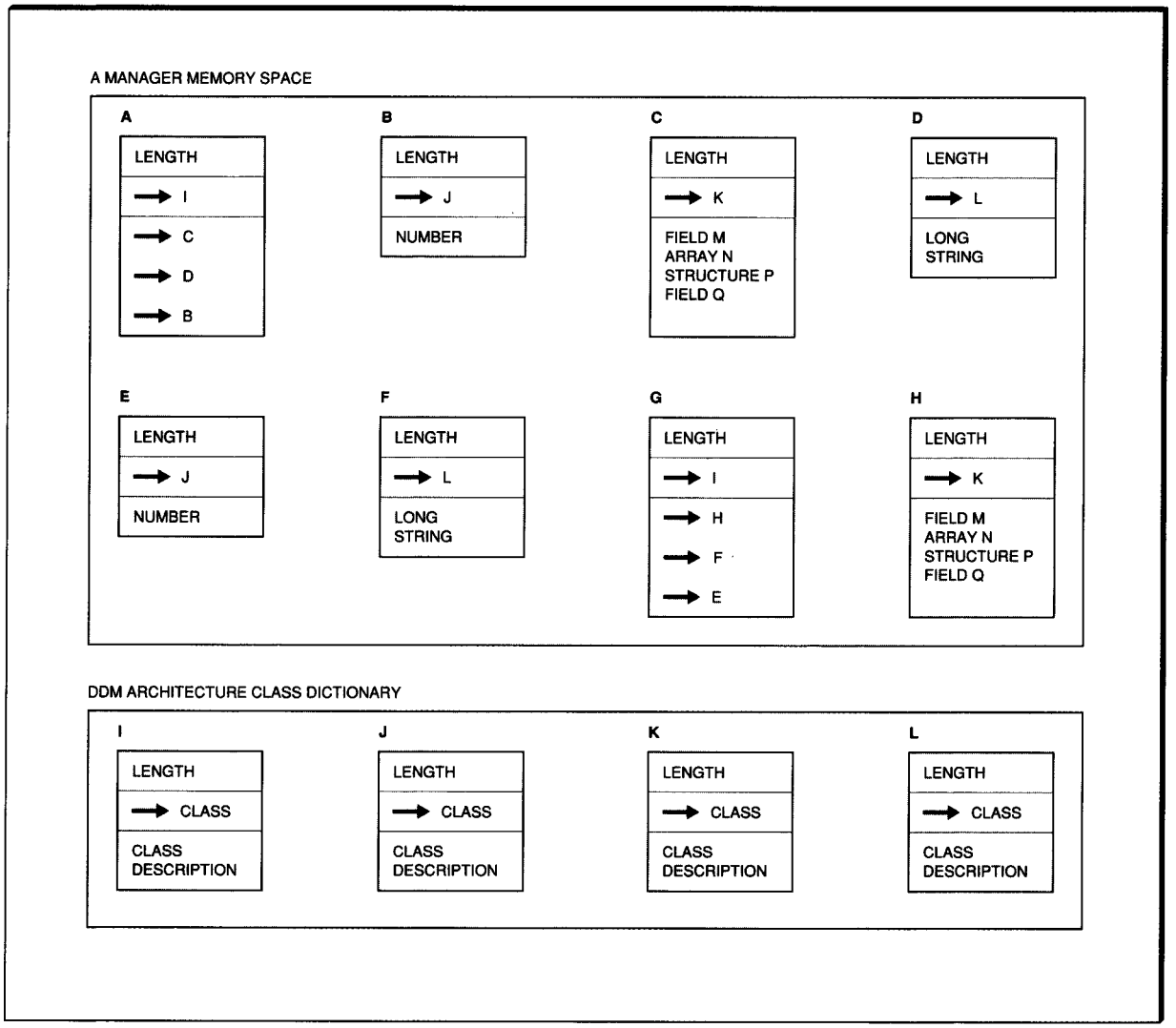


Figure 13 DDM architecture objects



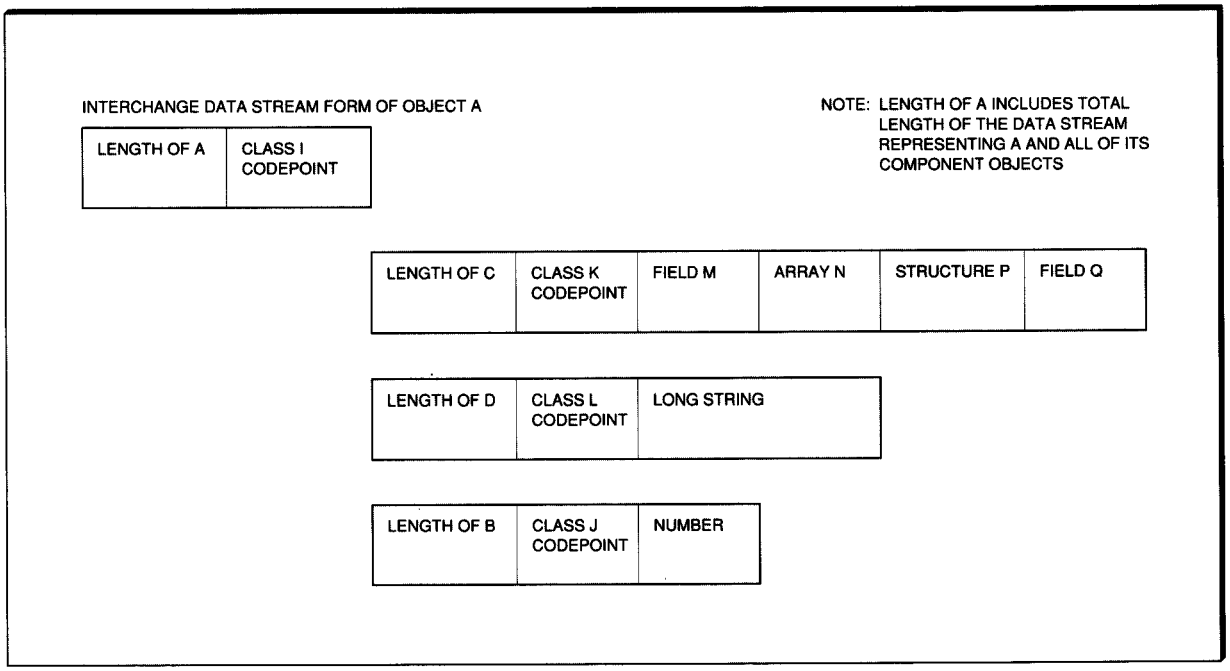
data structure. Each object specifies the following about itself:

- Its total length in bytes
- The identifier of its class
- Its data variables

Only three kinds of DDM architecture objects are in this layer, as shown in Figures 13 and 14:

- Simple scalars, which contain only a single instance of a DDM architecture data class, such as a single number or a single character string. DDM architecture attributes, such as the LENGTH attribute, are simple scalars, consisting of their length, class identifier, and the data value of the attribute. The memory space format and data stream format of simple scalars are the same.
- Mapped scalars, which contain a sequence of instances of the data classes. Records are an example of mapped scalars, consisting of their length, class identifier, and a sequence of data values. The memory space format and data stream format of mapped scalars are the same.
- Collections, which contain a sequence of ob-

Figure 14 DDM architecture object interchange format



jects or collections of objects. DDM architecture commands and reply messages are examples of collection objects. There are two formats for collections, a memory space format and a data stream format. In the memory space format, each variable of a collection is actually a pointer to an object of the collection. For example, collection *A* contains pointers to the mapped scalar *C*, the simple scalar *D*, and the simple scalar *B*. In the data stream format, the tree structure of collection *A* has been linearized. The length of collection *A* now includes the sum of the lengths of the scalars *C*, *D*, and *B*, and the pointers of *A* have been replaced by full copies of the scalars *C*, *D*, and *B*.

DDM architecture defines many subclasses of these kinds of objects. Although there have been many temptations to define classes of objects that are a hybrid of mapped scalars and collections (structures containing both scalar values and pointers to other objects), the DDM architects have, so far, resisted doing so. The marginal gains in efficiency that can be obtained have not been considered worth the tradeoff of increased complexity.

In the abstract model of DDM architecture, managers store information as objects and communicate by exchanging objects. Within a client or server, the memory space format of objects is used, and between clients and servers, the data stream format is used. Consider, for example, the creation of DDM architecture commands in response to a service request through a local interface. First, objects are individually created in a memory space for each command, parameter, and data item to be transmitted. The values of the variables of collection objects, such as commands, point to other objects. The result is an easily processed structure of objects linked by pointers. This structure is passed through a client agent to a client communications manager, which linearizes it in a data stream. The communications manager of the receiving server reverses this process to recreate linked objects in a server memory space that can be easily processed by the server's agent, resource managers, or access managers.

The key advantages of this approach are that only the communications managers need be concerned with building or parsing linearized data streams

and that many different forms of linearized data streams can be used. One such form is defined by DDM architecture, but as an alternative, the Basic Encoding Rules of OSI could also be used.

The Data Layer. Each type of system supports a number of ways of representing data of different types. For example, Personal System/2* computers support ASCII character data encodings, byte-reversed two's complement binary numbers, and IEEE (Institute of Electrical and Electronics Engineers) floating-point numbers. In contrast, a System/390* supports EBCDIC character data encodings, two's complement binary numbers, and hexadecimal floating-point numbers. Fundamental to communications between systems is establishing how each type of data is to be represented. Several approaches to this problem have been taken.

The first is to convert all data to or from a canonical form that is used primarily for communications. This approach was taken by the Basic Encoding Rules of OSI as the concrete representation of its Abstract Syntax Notation 1 (ASN.1). In general, this approach requires all data items to always be converted twice, once from the data representations of the client to the canonical form, and then from the canonical form to the representations of the server system. These conversions are performed whether or not the client and server systems are of the same type or have any representations in common.

The OSI approach does achieve universal connectivity, but only if both the sending and receiving applications agree on what data are being transmitted. OSI assumes that ASN.1 descriptions of the transmitted data are separately communicated between the programmers of the sending and receiving applications. It is then up to the programmers to include the necessary conversions in their communicating applications.

A second approach is based on the fact that for each type of data only a small number of different representation schemes are commonly used. Therefore, a flag can be sent from the client system to the server system, identifying which schemes the client system will use for each type. Where the client and the server use the same scheme, no conversions are required. Otherwise, conversions need be performed only once by the server system. This general approach has been adopted by the Open Software Foundation's DCE for remote procedure calls. As with the OSI approach, a separate flow of information is required between the programmers of the sending and receiving applications. For DCE, a file containing Network Interface Description Language, essentially enhanced C-language data declarations, must also be transmitted.

A third approach simply acknowledges that all data have to be represented according to some scheme, and if all senders and receivers use the same scheme, only some conversions will be required some of the time. The key is to pick the representation schemes that are most frequently used in order to minimize the number of conversions performed.

The primary products involved in the definition of DDM architecture (in 1982) were the System/36, System/38, and System/370*. Most of the data representation schemes used by these systems are identical, and DDM architecture was not concerned with those data types that were not, such as the floating-point formats. But along came the DDM/PC and PC Support products which use different schemes for binary numbers and character data. In order to communicate with the System/36, System/38, and System/370 DDM architecture products, the PC products were forced to construct and parse objects according to the representations previously chosen by DDM architecture.

Given the DDM architecture two-step model of translating programming interfaces to data stream structures (see the previous subsection), the PC products are able to create objects whose values are represented according to PC representations. The client communications manager can then convert them, as needed, to the representations required by DDM architecture. But it would also be possible to enhance the architecture along the lines of the DCE approach. If this were done, a flag sent by the client to the server could identify the representation schemes to be used, thereby allowing DDM architecture objects to be transmitted in whatever way the communicating servers choose. In particular, it would allow like systems, such as an OS/2 client and an OS/2 server, to communicate using their native representation schemes.

The approach used by DDM architecture works well for objects defined by the architecture, but it ignores mapped scalars that contain application data, such as file records, which DDM architecture treats as streams of undefined bytes. There is no way for DDM architecture products to know what these data look like or to know how the data are wanted. Any necessary conversion of these data is a responsibility of the requesting application program.

Distributed relational database support in Level 3 of DDM architecture addresses data representations and any necessary conversions.

With the introduction of distributed relational database support in Level 3 of DDM architecture, this situation changed. Clearly, the client and server database managers each know their representations of the SQL data types, and conversions can be performed by either server if descriptions of these data are also transmitted. The IBM Formatted Data:Object Content Architecture (FD:OCA) was selected by the Distributed Relational Database Architecture to convey this descriptive information. Previously defined as a means of describing tabular data included in documents, FD:OCA was well-suited to describing tabular relational database data. New DDM architecture objects were defined to carry FD:OCA data streams between the client and server SQL application (access) managers, which performed any necessary conversions.

The approach being considered for describing and converting file records is discussed in References 10 and 11.

Product-unique extensions

File and relational database managers of DDM architecture define standardized models of data management. DDM architecture also defines abstract services to complement these models and defines common data stream structures for the canonical representation of data objects, commands, and replies.

This framework has also been designed to support extensions to DDM architecture for homogeneous product connectivity. Any extensions that pertain to multiple products are candidates for the development of *standardized* DDM architecture. But other requirements are unique to single products, especially requirements for horizontal product growth or function distribution. Although product-unique extensions are not candidates for standardization, architectural definition, by the product, is still required.

In both cases, the framework of existing DDM architecture classes can be used as the basis for extensions to the architecture. DDM architecture allows the following types of *product-unique* enhancements:

- Whole new classes of managers (such as libraries or mailboxes) can be defined, either with new commands and replies unique to the class or reusing DDM architecture commands and replies as appropriate.
- The function of DDM architecture managers can be enhanced by defining new commands for a class.
- New parameters can be added to existing DDM architecture commands.
- New values can be defined for existing DDM architecture parameters.

A good example of product-unique extensions is provided by the AS/400. When connected to a non-AS/400 server, an AS/400 server complies strictly with DDM architecture. But when connected to an AS/400 server, a wide variety of extensions are used to make the full function of the data management system of the AS/400 available to remote AS/400 users, including many capabilities not covered by DDM architecture.

DDM architecture and international standards

The International Organization for Standardization (ISO) has defined an evolving set of standards

for OSI. The OSI standards are formulated around a powerful, seven-layer framework whose purpose is to allow venders to mix and match implementations of each layer and thereby achieve interconnectivity among their various implementations.

The DDM architecture fits into the ISO standard in the Open Systems Interconnection layer 7, which is the application layer.

How does IBM's DDM architecture fit into this framework? The simple answer is that all of the architecture fits into the OSI layer 7, the application layer. As mentioned previously, DDM architecture was designed to be independent of the communications facilities that are used to actually transmit messages between systems. We have even noted that OSI's Basic Encoding Rules could be used to encode messages in DDM architecture.

But what about the OSI File Transfer, Access, and Management (FTAM) standard that appears to be in competition with the file support of DDM architecture in layer 7? A comparison of the file manager classes of DDM architecture with FTAM shows that they are actually complementary and not really in competition, since they were designed to meet different requirements. For DDM architecture, the prime requirement was an ability to provide local or remote transparency to existing application programs and file systems. For FTAM, the prime requirement was to define a powerful new file model for use by new applications in accessing new file systems. As an international standard, FTAM certainly sets a goal for long-term file system revolution, but not evolution, as does DDM architecture.

Since IBM has committed to supporting OSI on its SAA systems, a challenge to DDM architecture is to integrate the FTAM file models into the DDM architecture framework, and for DDM architecture to make use of OSI level 6 communications facilities.

Summary

As discussed in this paper, DDM architecture is a framework for distributed application services. Multilayered and object-oriented, DDM architecture has defined a variety of distributed file and relational database services that can be provided over many different communications facilities. These services have evolved as a series of published levels of the architecture was developed. The openness of the DDM architecture framework invites the addition of a wide range of additional distributed application services.

Acknowledgments

When as many people have worked on a project as have worked on DDM architecture and its implementations, it is impossible to acknowledge them all. However, special acknowledgment must be given to John L. Bondy, the IBM manager who encouraged the DDM architects to explore innovative approaches to software architecture.

Appendix: Products using DDM architecture

The specifications of DDM architecture have been used to build and deliver a variety of IBM products allowing users to access and manage distributed files, including:

- System/36 System Support Program Distributed Data Management
- System/38 Control Program Facility Distributed Data Management
- Operating System/400 Distributed Data Management
- The DDM/PC and NetView/PC* client products for PC-DOS systems
- The CICS/DDM server product for use under the IBM Customer Information Control System (for both Multiple Virtual Storage and Virtual Storage Extended)[15]
- PC Support/36 and AS/400 PC Support client/server products
- 4680 Store Systems Distributed File Management

New products are under development to extend distributed file services to:

- Operating System/2 Distributed File Management (OS/2 DFM)
- MVS/ESA Distributed File Management (MVS DFM)
- VM/ESA Distributed File Management (VM DFM)

Additional IBM products will provide distributed relational database services to users of:

- DATABASE 2 (MVS DB2*)
- Structured Query Language/Data System (SQL/DS*)
- OS/2 Extended Services Database Manager (OS/2 DBM)
- RISC System/6000* Advanced Interactive Executive* (AIX*)

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, Digitalk, Inc., Xerox, Inc., the Open Software Foundation, or UNIX Systems Laboratories, Inc.

Cited references

1. R. A. Demers, "Distributed Files for SAA," *IBM Systems Journal* **27**, No. 3, 348–361 (1988).
2. R. Reinsch, "Distributed Database for SAA," *IBM Systems Journal* **27**, No. 3, 362–369 (1988).
3. *IBM Distributed Data Management Architecture: Implementation Programmer's Guide*, SC21-9529-03, IBM Corporation; available through IBM branch offices.
4. *SNA Transaction Programmer's Reference Manual for LU Type 6.2,* GC30-3084, IBM Corporation; available through IBM branch offices.
5. *IBM Distributed Relational Database Architecture Reference,* SC26-4651, IBM Corporation; available through IBM branch offices.
6. J. P. Gelb, "System-Managed Storage," *IBM Systems Journal* **28**, No. 1, 77–103 (1989).
7. P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," *OOPS Messenger* **1**, No. 1, 7–87 (August 1990).
8. *IBM Distributed Data Management Architecture: Reference Manual*, SC21-9526-03, IBM Corporation; available through IBM branch offices.
9. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Co., Reading, MA (1983).
10. R. A. Demers and K. Yamaguchi, "Data Description and Conversion Architecture," *IBM Systems Journal* **31**, No. 3, 488–515 (1992, this issue).
11. *IBM Distributed Data Management Architecture: Specifications for A Data Language*, SC21-8286, IBM Corporation; available through IBM branch offices.
12. *Formatted Data Object Content Architecture Reference*, SC31-6806, IBM Corporation; available through IBM branch offices.
13. *Character Data Representation Architecture Level 1, Reference*, SC09-1390, IBM Corporation; available through IBM branch offices.
14. 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. 2, 184–201 (March 1986).
15. K. Deinhart, "SAA Distributed File Access to the CICS Environment," *IBM Systems Journal* **31**, No. 3, 516–534 (1992, this issue).

Accepted for publication March 6, 1992.

Richard A. Demers *110 Salem Point SW, Rochester, Minnesota 55902.* Mr. Demers is a consultant on software architecture. In 1968 he joined IBM as an applications programmer in White Plains, New York. He received a B.A. degree in philosophy from Canisius College in 1969. In 1972, he moved to Endicott, New York, where he worked on systems software for the IBM 3895 Optical Check Reader. Mr. Demers moved to Rochester in 1975 to design the message handling and service components of the System/38 operating system. From 1982 to 1991, he was the lead architect in the design of IBM's Distributed Data Management (DDM) architecture, participated in the design of IBM's Distributed Relational Database Architecture (DRDA), and was the lead architect for IBM's data description and conversion architecture. He has received IBM Outstanding Innovation Awards for his work on DDM architecture (1987) and DRDA (1991). A member of the Association for Computing Machinery, his professional interests include operating systems, distributed processing, data management, programming languages, and object-oriented programming.

Jan David Fisher *IBM Application Business Systems, 3605 Highway 52 North, Rochester, Minnesota 55901.* Mr. Fisher is a senior programmer in the Rochester programming laboratory. He has been with IBM for 27 years. He joined the Wichita, Kansas, IBM branch office as a systems engineer, then moved to Rochester to join the Plastics Competency Center (IBM System/7s and plastic injection molding machines). He also worked in the Rochester Marketing Support Center and System/34, System/36 planning organization as the communications software product planner. He joined the Distributed Data Architecture Department in 1985 as a senior planner and later became the team leader of the architects working on the Distributed Data Management architecture. Mr. Fisher graduated from Purdue University with a B.S.E.E. in 1965, and has been taking graduate courses. He is a member of Toastmasters International, and his interests include both history and science fiction. He is a judge for Odessey of the Mind.

Sunil S. Gaitonde *IBM Application Business Systems, 3605 Highway 52 North, Rochester, Minnesota 55901.* Dr. Gaitonde is an advisory programmer working on the operating system of the AS/400. He received his bachelor's degree from the Indian Institute of Technology, Kharagpur, India in electrical engineering and holds a master's degree and Ph.D. in computer engineering from Iowa State University. He joined IBM at Rochester in 1988. Dr. Gaitonde worked on the Distributed Data Management architecture to extend it to support IBM's Distributed Relational Database Architecture. His current focus is on the Open Software Foundation's Distributed

Computing Environment. His interests include operating systems, computer network protocols, and object-oriented programming languages.

Richard R. Sanders *IBM Application Business Systems, 3605 Highway 52 North, Rochester, Minnesota 55901.* Mr. Sanders is a senior programmer in the AS/400 Data Management Design Control group. He was an architect of the Distributed Data Management (DDM) architecture for over five years and spent over two years as the AS/400 representative to the Distributed Relational Database Architecture (DRDA). He has received two IBM Outstanding Technical Achievement Awards for his contributions to DDM and DRDA. Mr. Sanders received a B.S. in computer science and mathematics from Mankato State College in 1972. He joined IBM in Rochester in 1977.

Reprint Order No. G321-5482.