Various types of middleware are **classified**, their **properties** described, and their **evolution** explained, providing a **conceptual model** for understanding today's and tomorrow's distributed system software.

Middle

The computing facilities of large-scale enterprises are evolving into a utility, much like power and telecommunications. In the vision of an information utility, each knowledge worker has a desktop appliance that connects to the utility. The desktop appliance is a computer or computer-like device, such as a terminal, personal computer, workstation, word processor, or stock trader's station. The utility itself is an enterprise-wide network of information services, including applications and databases, on the local-area and wide-area networks.

Servers on the local-area network (LAN) typically support files and file-based applications, such as electronic mail, bulletin boards, document preparation, and printing. Local-area servers also support a directory service, to help a desktop user find other users and find and connect to services of interest. Servers on the wide-area network (WAN) typically support access to databases, such as corporate directories and electronic libraries, or transaction processing applications, such as purchasing, billing, and inventory control. Some servers are gateways to services offered outside the enterprise, such as travel or information retrieval services, news feeds (e.g., weather, stock prices), and electronic document interchange with business partners. In response to such connectivity, some businesses are redefining their business processes to use the utility to bridge formerly isolated component activities. In the long term, the utility should provide the information that people need when, where, and how they need it.

Today's enterprise computing facilities are only an approximation of the vision of an information utility. Most organizations have a wide variety of heterogeneous hardware systems, including personal computers, workstations, minicomputers, and mainframes. These systems run different operating systems (OSs) and rely on different network architectures. As a result, integration is difficult and its achievement uneven. For example, local-area servers are often isolated from the WAN. An appliance can access files and printers on its local server, but often not those on the servers of other LANs. Sometimes an application available on one local area server is not available on other servers, because other departments use servers

Philip A. Bernstein

eware: A Model for Distributed System Services

on which the application cannot run. So some appliances cannot access the application. Wide-area servers often can support only dumb terminals, which a desktop appliance must emulate to access a server. Sometimes a desktop appliance can gain access to a wide-area server only if a local-area server was explicitly programmed for access to the wide-area server. A user may have to log into each server separately with independently maintained passwords and through different user interfaces, each with a different look and feel. Even if the desktop appliance can access a remote application, its spreadsheet or word processor often cannot access data provided by that application without special programming. These are only some of the limitations of such systems.

In response to their frustrations in implementing enterprise-wide information systems, large enterprises are pressuring their vendors to help them solve heterogeneity and distribution problems. One way is by supporting standard programming interfaces. Standard programming interfaces make it easier to port applications to a variety of server types, giving the customer some independence from the vendors. In the past, supporting a standard programming language, such as Cobol or C, was enough, but not anymore. Today's typical application may use database, communication, presentation, and other services, whose interfaces are not part of the language definition. To port an application, these interfaces must be supported on the target platform. So users want interfaces that are widely supported, that is, standard.

Increasingly, standard interfaces are important to server vendors themselves. Customers buy applications, not servers. Customers will choose any server that can run the applications they want. By supporting many standard interfaces, a vendor increases the number of applications that run on their servers, making their servers more attractive to customers.

Another way vendors solve heterogeneity problems is by supporting standard protocols. Standard protocols enable programs to interoperate. By *interoperate*, we mean that a program on one system can access programs and data on another system. Interoperation is possible only if the two systems use the same protocol, that is, the same message formats and sequences. Also, the applications running on the systems must have similar semantics, so the messages map to operations that the applications understand. The systems supporting the protocol may use different machine architectures and OSs, yet they can still interoperate.

For example, the Open Software Foundation's Distributed Computing Environment (OSF DCE) fully specifies its remote procedure call (RPC) protocol. An implementation of DCE RPC includes a compiler that translates an interface definition into a client stub, which marshals a procedure call and its parameters into a packet, and a server stub, which unmarshals the packet into a local server call (see Figure 1). The client stub can marshal parameters from a language and machine representation different from the server stub's, thereby enabling interoperation. An RPC implementation also includes a run-time library, which implements the protocol for message exchanges on a variety of network transports, enabling interoperation at that level. At least one vendor (Microsoft) has implemented the protocol independently (without using the OSF implementation) yet can interoperate with other implementations.

As another example, many vendors have implemented IBM's 3270 protocol to interoperate with IBM mainframe applications. These vendors often support a programming interface different from what IBM offers on its systems. But since they support the same protocol, the systems can interoperate.

To help solve customers' heterogeneity and distribution problems, and thereby enable the implementation of an information utility, vendors are offering distributed system services that have standard programming interfaces and protocols. These services are called *middleware services*, because they sit "in the middle,'' in a layer above the OS and networking software and below industry-specific applications.

Like large enterprises, application developers also have heterogeneity and distribution problems they want vendors to solve. Developers want their applications to depend only on standard programming interfaces, so the applications will run on most popular systems. This increases their potential market and helps their large-enterprise customers who must run applications on different types of systems. Developers need higher-level interfaces, which mask the complexity of networks and protocols and thereby allow developers to focus on application-specific issues, where they are most qualified to add value. Since customers focus on buying applications, not the underlying computer systems, vendors are anxious to meet these requirements so they can attract popular applications to their systems. As with large enterprises, vendors respond to application developers by offering middleware.

For many new applications, middleware components are becoming more important than the underlying OS and networking services on which the applications formerly depended. For example, new applications often depend on a relational database system rather than on the OS's record-oriented file system and on an RPC mechanism rather than on transport-level messaging (e.g., send-message, receive-message). In general, middleware is replacing the nondistributed functions of OSs with distributed functions that use the network (e.g., distributed database, remote file access, RPC). For many applications, the programming interface provided by middleware defines the application's computing environment. For example, many applications regard fourth-generation languages (4GLs), transaction processing (TP) monitors (e.g., IBM's CICS, Digital's ACMSxp), and office frameworks (e.g., Lotus Notes, Digital's LinkWorks) in this way.

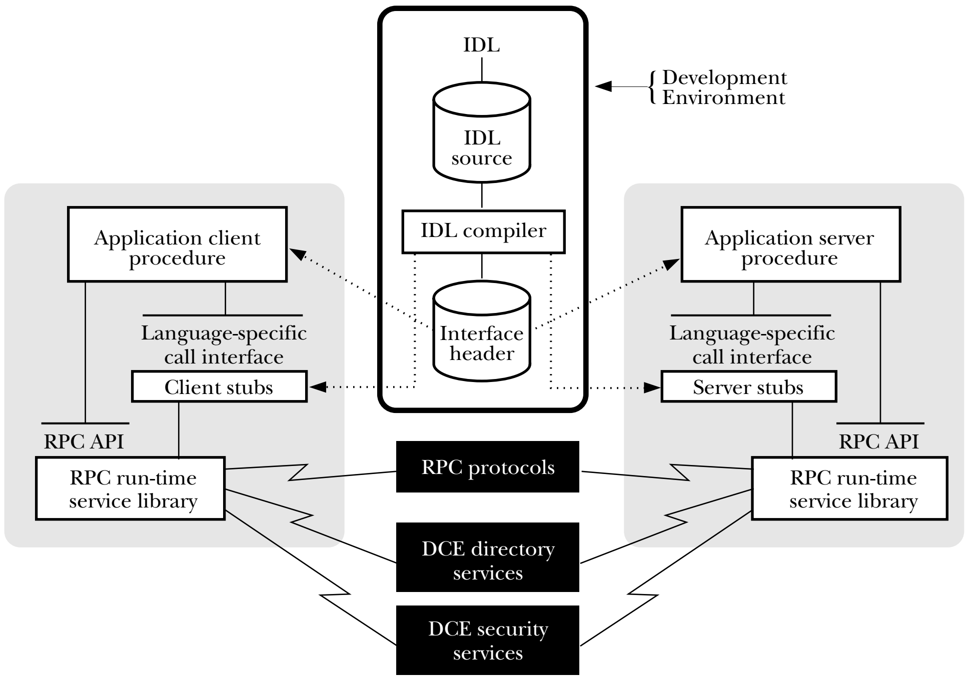

IDL
IDL source
IDL compiler
Interface header
Development Environment
Application client procedure
Language-specific call interface
Client stubs
RPC API
RPC run-time service library
Application server procedure
Language-specific call interface
Server stubs
RPC API
RPC run-time service library
RPC protocols
DCE directory services
DCE security services

Figure 1. Remote Procedure Call architecture

Legacy applications, built before portable middleware became popular, can also benefit from middleware services. One can encapsulate the legacy application as a set of functions and use communications middleware services to provide remote access to those functions. For example, one can use an implementation of the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) for this purpose. This includes an object-oriented aplication programming interface (API) for binding to remote applications (objects) and for invoking them, through either a static interface (like RPC stubs) or a dynamic interface (of the form Call(application-name, parameter1, parameter2, ...)). One can also provide an advanced user interface to a legacy application through high-level presentation middleware services, for example, by intercepting character terminal I/O and translating it into operations on a graphical user interface (GUI). Or one can replace some internal components of the legacy application with middleware services. For example, one can replace application-specific database functions with middleware database services. This saves maintenance by relying more on the middleware vendor, which often improves functionality, since the middleware vendor has greater resources to expend on these distributed system functions than the application developer does.

This article classifies different kinds of middleware, describes their properties, and explains their evolution. The goal is to provide a conceptual model for understanding today's and tomorrow's distributed system software. Regrettably, there is no standard terminology for many of the concepts discussed here. We have attempted to follow common industry usage of terms and to add some precision to the definitions. However, vendors and consortia use these terms in conflicting ways. We therefore warn the reader to be careful in interpreting these terms outside the context of this article.

Middleware Services

We describe properties of middleware and the problems they do and don't solve. See also [4].

A *middleware service* is a general-purpose service that sits between platforms and applications (see Figure 2). By platform, we mean a set of low-level services and processing elements defined by a processor architecture and an OS's API, such as Intel *x*86 and Win-32, Sun SPARCstation and Sun OS, IBM RS/6000 and AIX, Alpha AXP and OpenVMS, and Alpha AXP and Windows NT.

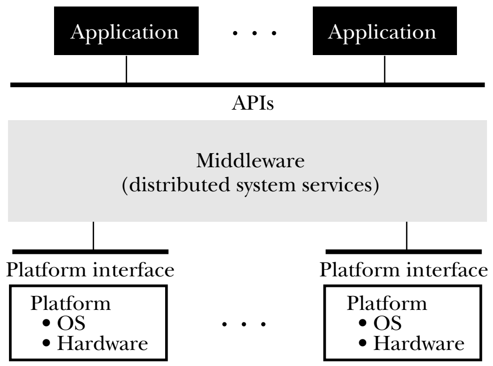


Figure 2. Middleware

A middleware service is defined by the APIs and pro-

A middleware service meets the needs of a wide variety of applications
across many industries.

tocols it supports. It may have multiple implementations that conform to its interface and protocol specifications, such as the different implementations of DCE RPC and IBM 3270 protocol mentioned earlier.

Like many high-level system concepts, middleware is hard to define in a technically precise way. However, middleware components have several properties that, taken together, usually make clear that the component is not an application or platform-specific service: They are generic across applications and industries, they run on multiple platforms, they are distributed, and they support standard interfaces and protocols. We describe each of these properties in turn.

A middleware service meets the needs of a wide variety of applications across many industries. For example, a message switch, which translates messages between different formats, is considered middleware if it makes it easy to add new formats and is usable by many applications. If it deals with formats only for a single industry (e.g., trading securities) and is embedded in a single application (e.g., a back-office brokerage system), then it is not middleware.

A middleware service must have implementations that run on multiple platforms. Otherwise, it is a platform service. For example, relational database management systems (DBMSs) are middleware. Many relational database products run on multiple platforms. By contrast, byte-stream file systems are platform services. Each OS has its own implementation, usually with an OS-specific interface. By running on multiple platforms, a middleware service enhances the platform coverage of applications that depend on it. If the service is distributed, this also enhances interoperability, since applications on different platforms can use the service to communicate and/or exchange data. To have good platform coverage, middleware services are usually programmed to be *portable*, meaning that they can be ported to another platform with modest and predictable effort.

A middleware service is distributed. That is, it either can be accessed remotely (e.g., a database or presentation service) or enables other services and applications to be accessed remotely (e.g., a communications service). A remotely accessible middleware service usually includes a *client* part, which supports the service's API running in the application's address space, and a *server* part, which supports the service's main functions and may run in a different address space (i.e., on a different system). There may be multiple implementations of each part.

Ideally, a middleware service supports a standard protocol (e.g., TCP/IP or the ISO OSI protocol suite), or at least a published one (e.g., IBM's SNA LU6.2). That way, multiple implementations of the service can be developed and those implementations will interoperate. However, if a middleware service really does run on all popular platforms, it may be regarded as standard even though its protocols are not published. For example, most database system products have this property. If platform coverage is good enough, customers may not push vendors to conform to a standard protocol when it is developed. For example, the SQL Access Group's client-server protocol has not gained much acceptance, although its API (ODBC) is quite successful, since it is supported by Microsoft on its Windows operating systems.

A middleware service should support a standard API. A middleware service is *transparent* with respect to an API if it can be accessed via that API without modifying that API. Nontransparent middleware requires a new API. Middleware that is transparent with respect to a standard API is more easily accepted by the market, because applications that use the existing API can use the new service without modification. For example, several different distributed file-sharing protocols have been implemented under the standard file access API, as in Digital's PATHWORKS product, which includes file services for personal computers.

If a vendor has broad platform coverage and substantial market share, then its API and protocol may be regarded as de facto standards, even if they are not supported by other vendors. For example, the relational database systems ORACLE and SYBASE support their own dialects of the SQL language yet are regarded as standard enough by most customers. Similarly, IBM's CICS TP monitor uses a proprietary API and protocol (LU6.2) yet is a de facto standard.

Whether a given service is classified as middleware may change over time. A facility that is currently regarded as part of a platform may, in the future, become middleware, to simplify the OS implementation and make the service generally available for all platforms. For example, we used to regard a record-oriented file system as a standard part of OSs, as indeed they were in all commercial OSs developed before 1980. However, today we often think of this as middleware, such as implementations that conform to the X/Open C-ISAM API. Conversely, middleware can migrate into the platform, to improve the middleware's performance and to increase the commercial value of the platform. For example, interfaces to transport-level protocols were often regarded as "communications access method'' products, separate from the OS. Now they are usually bundled with the OS.

Due to the importance of standard interfaces for application portability and standard protocols for interoperability, middleware has been the subject of many standardization efforts, some through formal standards bodies such as ISO and ANSI, some through industry consortia such as X/Open, OSF, and OMG, and some through the sponsorship of a company with a major market share, such as Microsoft's Windows Open Services Architecture (WOSA). (The existence of these industry-wide organizations and their growing impact on products is itself a testimonial to the importance of middleware.) Sometimes an individual service gathers a major market share and thereby becomes a de facto standard, such as Adobe's PostScript, IBM's CICS TP monitor, and Sun's Network File Service (NFS).

Standardization efforts are, in turn, leading to corporate procurement standards. That is, companies and governments are selecting some standards as vendor requirements, to ensure that similar products obtained from different vendors will support the same applications and will interoperate. Some examples of customer-oriented procurement standards are the U.S. government's Government Open System Interconnect Profile (GOSIP) and Nippon Telegraph and Telephone's Multivendor Integration Architecture (MIA), which has led to the broader SPIRIT Consortium, covering telecommunications and perhaps other industries.

Vendors often respond to standards by adding advanced nonstandard features, in an attempt to produce more desirable products and lock in customers to features that are not available from competing standard products. For example, relational database vendors have been adding features to SQL for many years, with the standards processes struggling to keep up by defining vendor-independent standard definitions for those features.

The following components are or could be middleware services:

- *Presentation management:* Forms manager, graphics manager, hypermedia linker, and printing manager.
- *Computation:* Sorting, math services, internationalization services (for character and string manipulation), data converters, and time services.
- *Information management:* Directory server, log manager, file manager, record manager, relational database system, object-oriented database system, repository manager.
- *Communications:* Peer-to-peer messaging, remote procedure call, message queuing, electronic mail, electronic data interchange.
- *Control:* Thread manager, transaction manager, resource broker, fine-grained request scheduler, coarse-grained job scheduler.
- *System management:* Event notification service, accounting service, configuration manager, software installation manager, fault detector, recovery coordinator, authentication service, auditing service, encryption service, access controller.

Not all of these services are currently distributed, portable, and standard. But a sufficiently large number of them are or will be to make the middleware abstraction worthwhile.

The categories listed here are arbitrary. They are just a convenient way of grouping the services, making them easier to remember and discuss.

The main purpose of middleware services is to help solve many of the problems discussed earlier in this article. They provide platform-independent APIs, so applications will run on multiple platforms. And they include high-level services that mask much of the complexity of networks and distributed systems. They also factor out commonly used functions into independent components, so they can be shared across platforms and software environments.

However, middleware services are not a panacea. First, there is a gap between principles and practice. Many popular middleware services use proprietary APIs (usually making applications dependent on a single vendor's product) and proprietary and unpublished protocols (making it difficult for different vendors to build interoperable implementations). For example, as mentioned earlier, many relational DBMSs support proprietary SQL dialects and proprietary protocols. Some are not available on many popular platforms (limiting the customer's ability to connect or port to heterogeneous systems), such as Oracle's (formerly Digital's) Rdb and IBM's DB2 relational DBMSs. Even when a middleware service is state-of-the-art, an application developer that depends on it has a new risk to manage, the risk that the service will not keep pace with technology. For example, many applications that used a network (e.g., CODASYL) DBMS had to be rewritten to benefit from relational DBMSs that have replaced them as the de facto standard.

Second, the sheer number of middleware services is a barrier to using them. Even a small number of middleware services can lead to much programming complexity, when one considers each service's *full API*, including not only service calls but also language bindings, system management interfaces, and data definition facilities. To keep their computing environment manageably simple, developers have to select a small number of services that meet their needs for functionality and platform coverage.

Third, while middleware services raise the level of abstraction of programming distributed applications, they still leave the application developer with hard design choices. For example, the developer must still decide what functionality to put on the client and server sides of a distributed application. It is usually beneficial to put presentation services on the client side, close to the display, and data services on the server side, close to the database. But this isn't always ideal, and in any case leaves open the question of where to put other application functions.

Frameworks

A *framework* is a software environment that is designed to simplify application development and system management for a specialized application domain (see Figure 3). A framework is defined by an API, a user interface, and a set of tools. It may also have framework-private middleware services, in addition to ones that it imports from other products. Some popular types of frameworks include office system environments (e.g., Lotus Notes, Microsoft Office, Digital's LinkWorks), TP monitors (e.g., IBM's CICS, Digital's ACMSxp, Novell's Tuxedo, Transarc's Encina), 4GLs (e.g., Uniface, Cognos, and Focus), computer-aided design frameworks (e.g., Mentor Graphics' Falcon, Digital's Powerframe), computer-aided software engineering workbenches (e.g., HP's SoftBench, Texas Instruments' Composer by IEF, Andersen Consulting's Foundation, Digital's COHESIONworX), and system management workbenches (e.g., HP's OpenView, Tivoli's Management Environment, IBM's NetView).

Frameworks are a kind of middleware. For clarity, we therefore use the term "middleware services'' for underlying distributed system services and "middleware'' for middleware services and/or frameworks.

A framework's API may be a *profile* of APIs for a set of middleware services, or it may be a new API that simplifies the APIs of underlying middleware services that it abstracts. For example, a computer-aided design (CAD) or computer-aided software engineering (CASE) framework's API may include a presentation service, an invocation service (to call tools from the user interface), and a repository service (to store persistent data shared between tools); it could be simply the union of those services' APIs, or it could be abstracted by a new API that maps to those services. If a framework's API is different from that of the underlying services, it may be only a veneer that covers the underlying services with a common syntax. More often, it adds value by specializing the user interface, simplifying the API by maintaining shared context, or adding framework-private middleware services. For example, all TP monitors we know of add value in this way.

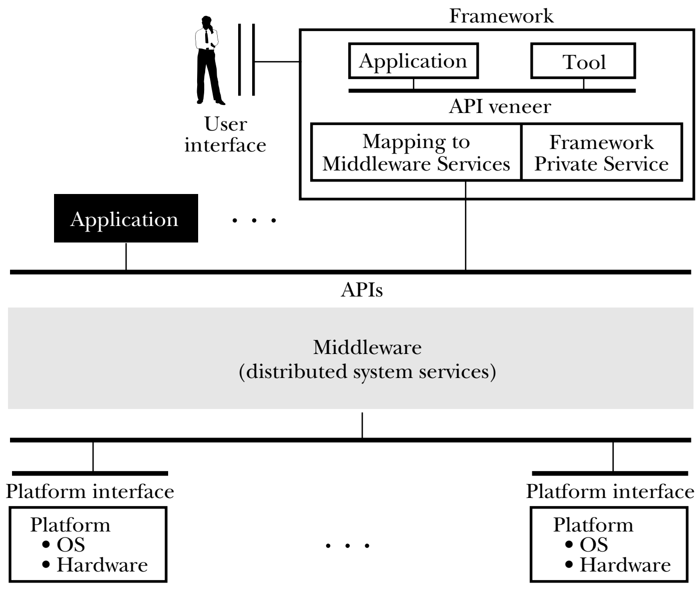


Figure 3. Framework architecture

Sometimes services grow up into frameworks. For example, most relational DBMSs have evolved to include rich toolsets accessible through a special user interface, retaining SQL as the API. Sometimes a set of services is sufficiently integrated to resemble a framework. For example, OSF's DCE integrates RPC, a naming service, an authentication service, a time service, a unique universal identifier service, and a file service. Although it has no special user interface (UI), and its API is just those of its component services, it does maintain context across calls (authenticated user id) and transparently invokes services to simplify certain operations, such as transparently invoking the name service to find a server that can process an RPC call.

Since a framework layers on middleware services, a framework provider is a customer of middleware services. By the same token, an application that layers on a framework is the framework's customer, and only indirectly the middleware service's customer. Given the wide variety and complexity of middleware services, a substantial and growing fraction of applications depend on frameworks to simplify their underlying middleware environment rather than directly accessing middleware services. This is analogous to the past trend of applications to move away from direct use of platform services and rely more heavily on more abstract middleware services.

A framework may have its own UI, which is a specialization of the GUIs of the underlying platforms it runs on and which has a special look and feel. For example, an office system framework may specialize the GUI to provide the appearance of a desktop, with icons and layout suitable for office users and applications. If a framework supports multiple GUIs (such as Motif, Microsoft Windows, and Macintosh), then this GUI specialization makes it easier for users to move between GUIs (e.g., between their Windows laptop and Unix workstation).

One way a framework can simplify its API is by maintaining context across calls to different services. For example, in most frameworks, service calls require a user identifier (for access control) and a device identifier (for communications binding) as parameters. However, the framework can maintain these identifiers as context for the application and not require them as parameters, thereby simplifying the API. For example, the CASE framework COHESIONworX maintains a context containing an authenticated user name, a display name, and a work area consisting of file directories and pointers to objects. Maintaining context is not a unique property of frameworks; a service can maintain context too, though that context is usually local to the one service.

For example, DCE RPC maintains an authenticated user id across multiple RPC calls.

A framework can also simplify its API by exploiting a work model that does not require all of the features of the middleware services it uses. For example, a CASE environment may offer a simplified interface for software configuration management that does not expose all of the features of the underlying repository manager. This improves ease of use and increases the number of services that are transparent with respect to the APIs.

Often, a framework offers a middleware service that is private to the framework, usually because the framework requires the service but no standard middleware service is available. For example, many TP monitors use a private implementation of RPC. However, with the availability of the OSF DCE, which includes an RPC, some of these TP monitors are replacing their proprietary RPC implementations with the standard DCE RPC middleware, such as IBM's CICS/6000, Transarc's Encina, and Digital's ACMSxp. As another example, CASE and CAD frameworks often offer framework-specific services for versioning and configuration management. These services may be replaced in the future by standard middleware if a repository technology emerges as such a standard.

Frameworks usually include *tools*, which are generic applications that make the framework easier to use. Tools may be designed for end users, programmers, or system managers. They may be provided by the framework's vendor or other parties. They may be designed to be used with a particular framework or with a variety of frameworks. Examples of tools include editors, help facilities, forms managers, compilers, script interpreters, debuggers, performance monitors, and software installation managers. Although in principle a framework does not need to have tools, in practice they all do, to make them sufficiently attractive to human users.

A tool is part of a framework if it is integrated through data, control, and/or presentation. In data integration, a tool shares (usually persistent) data with other tools that are part of its framework. This requires that the tools agree on the object model (the abstract model in which data formats are defined) and on the format of the shared data. Sometimes the format is defined by one tool that owns and exports the data; other tools can reference that data provided they can cope with the owning tool's format, such as a relational database catalog. Other times the data is equally shared by two or more tools, which is easiest to arrange if all tools are supported by one vendor. For example, CASE frameworks often contain a repository that is shared by tools for analysis and design and for program generation, such as Andersen Consulting's Foundation and Texas Instruments' Composer by IEF. Similarly, a 4GL typically has a data dictionary that is shared by its forms manager, query language, and application programming language.

Whether the data is owned by one tool or equally shared by many tools, tools may share online access to the fine-grained data (as in the previous examples of database catalog, CASE framework repository, and 4GL dictionary), or they may transfer the data in bulk from tool to tool (for example, using the CASE Data Interchange Format between CASE tools or the Express exchange format between CAD tools). If the data is accessed online, then the tools may have to agree on protocol, that is, on the set of operations allowed on the data and the allowed sequences of those operations.

In control integration, a user or tool operating within a framework can invoke another tool, and the called tool can exchange data and control signals appropriately with its caller. This involves making the tool's interface known to the framework, invoking the tool (or creating an instance of the tool, if necessary), and exchanging messages with the tool. For example, in Digital's COHESIONworX, HP's SoftBench, and Sun's SPARCworks, each tool in the CASE framework can send messages, which are forwarded to all other tools that have declared an interest in messages of that type. Sometimes it's useful if the tool executes on a different system from the one that the framework is executing, in which case control integration requires a remote invocation service. For example, Digital's COHESIONworX uses a CORBA implementation (Digital's Object Broker) to invoke tools on remote systems.

In presentation integration, a tool shares the display with other tools executing in the framework. Preferably, it has the same look and feel as those other tools, usually achieved by using the same GUI and adopting standard usage conventions, such as offering semantics similar to those of abstract operations (e.g., cut, paste, drag, drop). The look and feel may be guided by a metaphor, such as a virtual desktop, which leads to usage conventions for screen space, control panel layout, etc. Sometimes presentation integration drives the requirements for data and control integration. For example, if a spreadsheet is invoked from a compound document editor, both data integration (of the spreadsheet's contents) and control integration (for launching the spreadsheet application) are needed, but it was the presentation integration that even allowed the user to express the concept.

Pragmatically, presentation and control integration are usually more urgently needed than data integration, because most users need multiple tools and do not want to deal with tool-specific look and feel and invocation techniques. While there are often great benefits to data integration, tools are still useful and easy to use without this integration.

Presentation and control integration are often easier to implement than data integration. For control integration, each tool can be independently integrated by defining its interfaces and providing access to those interfaces through a middleware communications service. This work is localized at the tool's interface and requires no modification of the body of the tool. The same is true for presentation integration, by intercepting display I/O operations and translating them into operations on a common GUI. By contrast, data integration requires that all tool vendors agree on a common format for the data they share. Since data accesses are scattered throughout the tool's implementation, changing data formats is often tedious and expensive. One can localize the tool changes by creating a database view of shared data that allows the tool to access shared data in its native format. However, the shared data is often stored in a data manager that doesn't support views or supports views that are not updatable, making the views approach nonviable.

The distinctions among data, presentation, and control integration are blurred somewhat by taking an object-oriented view of the interfaces a tool calls. In all cases, the tool is invoking methods on objects, so the distinctions among calling a presentation interface, invoking a tool, and accessing data disappear. However, the different effects of the three types of integration on user capabilities remain, as does the critical technical problem of data integration. That is, the tools that share data must agree on the format of each data type they share, which is equally difficult whether it is manifested in a method name or in method parameters.

TP Monitors—An Example

To illustrate middleware concepts, we will look at one type of framework: TP monitors [1]. The main function of a TP monitor is to coordinate the flow of requests between terminals or other devices and application programs that can process these requests. A *request* is a message that asks the system to execute a transaction. The application that executes the transaction usually accesses resource managers, such as database and communications systems. A typical TP monitor includes functions for transaction management, transactional interprogram communications, queuing, and forms and menu management (see Figure 4).

Transaction management involves support for operations that start, commit, and abort a transaction. It also has interfaces to resource managers (e.g., database systems) that are accessed by a transaction, so a resource manager can tell the transaction manager when it is accessed by a transaction and the transaction manager can tell resource managers when the transaction has committed or aborted. A transaction manager implements the two-phase commit protocol, to ensure that all or none of the resource managers accessed by a transaction commit (see [2], chap. 7).

Transactional interprogram communications support the propagation of a transaction's context when one program calls another. This allows the called program to access resources on behalf of the same transaction as the caller. The communications paradigm may be peer-to-peer (i.e., send-message, receive-message) or RPC (i.e., send-request, receive-reply).

A queue manager is a resource manager that supports (usually) persistent storage of data to move between transactions. Its basic operations are enqueue and dequeue. A distributed queue manager can be invoked remotely and may provide system management operations to forward elements from one queue to another.

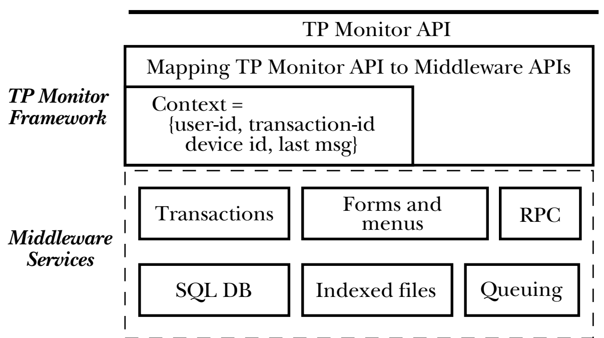
A forms manager supports operations to send and receive forms to and from display devices. It has a development system for defining forms and translating them into an internal format and a run-time system for interpreting the content of a form in an application.

Early TP monitors implemented all of these functions as framework-specific services, relying only on platform services. Today, many of these framework-specific services are available as off-the-shelf middleware. TP monitors being built today make use of such off-the-shelf services, such as record managers, transaction managers, and queue managers. These services may also be used directly by applications or by other middleware services (e.g., a DBMS) or frameworks (e.g., an office system framework).

All of the preceding services are the subject of efforts to standardize their APIs and protocols. For example, X/Open is working on standard APIs for transaction management and transactional communication, and ISO is working on a protocol standard for queuing.

Most TP monitors were highly dependent on the platform for which they were built. Today's monitors, layered on middleware services, are more portable. They are designed to be dependent mostly on the middleware they use, not on the platform that sits below the middleware. They have selected middle-

Figure 4. TP monitor architecture



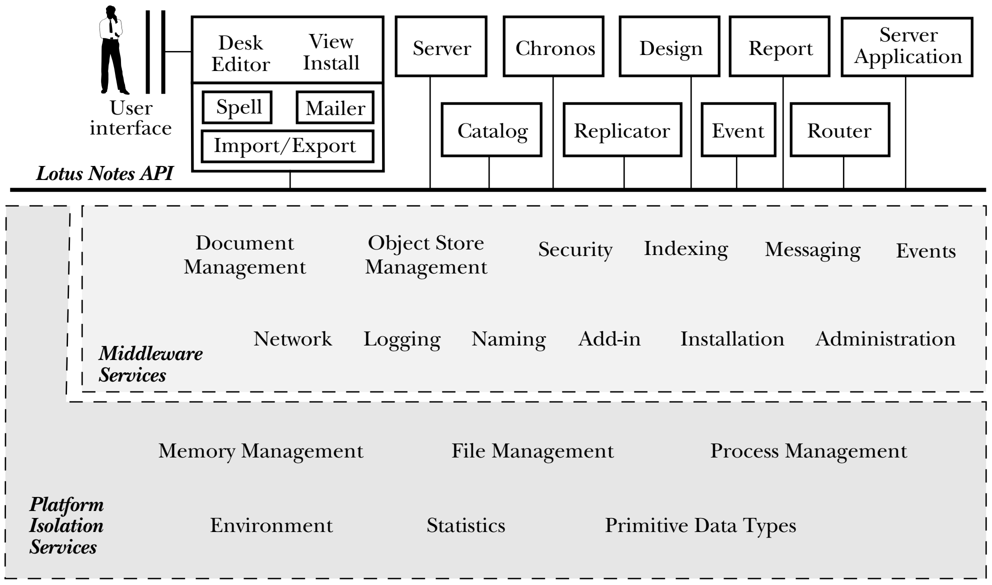


Figure 5. Architecture of Lotus Notes R3.0

ware services that are themselves portable and have been ported to many platforms, such as OSF DCE. Therefore, such TP monitors can be ported to a variety of platforms. For example, IBM has built a new implementation of its CICS TP monitor layered on transaction middleware from Transarc Corp. and has announced that the implementation will be ported to non-IBM Unix systems.

A TP monitor integrates the services so they are accessible via a simplified and uniform API. One way it simplifies the API is by maintaining context to avoid passing certain parameters. For example, a TP monitor typically maintains context about the current request, transaction, and user. Most application functions need not specify this information. Rather, the TP monitor fills in the necessary parameters when translating an application call into a call on the underlying middleware service. For example, an application may ask to enqueue a message, and the TP monitor would add as parameters the current transaction identifier (so the queue manager can tell the transaction manager it was accessed by the transaction) and user identifier (so the queue manager can check the user's authorization).

A TP monitor may specialize the user interface. For example, it may have a stylized interface for login, display of errors, and display and interaction with menus.

A TP monitor generally incorporates tools for application programming. For example, it may include a data dictionary for sharing record definitions between the forms manager, application programs, and database system. Digital used its Common Data Dictionary/Repository to integrate its TP Monitor (ACMS), relational DBMS (Rdb), forms manager (DECforms), and compilers in this way. This is an example of data integration. It also may include a CASE framework for simplifying the invocation and use of tools for compilation, software configuration management, and testing, as in Digital's DECadmire and TP Workcenter. This is an example of control and presentation integration.

A TP monitor also incorporates tools for system management. These tools allow one to display the state of a transaction or request, to determine what components are currently unavailable, and to monitor performance and tune performance parameters. System management may be implemented in its own framework, which ties together the system management facilities of the TP monitor with the platform's and database system's system management functions, so all resources can be managed in the same way from the same device.

Most customers buy a complete TP system from one system vendor, including the TP monitor, database system, and platform. The system vendor may or may not be the author of the TP monitor software. Still, the system vendor is responsible for ensuring that the complete software complement has suitable performance, reliability, usability, etc.

Lotus Notes

Another popular framework product that illustrates our points is Lotus Notes. It supports the development and execution of applications that create, access, track, share, and organize information represented as documents. We describe some of the key features of Lotus Notes in terms of our middleware model.

Applications are stored in databases on servers across networks of occasionally connected computers. Each database contains documents, forms (which define the design of multimedia documents), and views (which provide, for example, different subsets and orders of documents in databases). The majority of application development can be done by nonprogrammers using forms, views, and a formula language similar to those in spreadsheets. The databases are managed with little centralized control. That is, each Lotus Notes server (i.e., an installation of Lotus Notes on one system) independently defines its databases. Middleware services are provided to track and control these independent but related databases.

Lotus Notes R3.0 is structured in three layers: a platform isolation layer, a set of middleware services accessed through an integrated API, and a set of tools (see Figure 5). The platform isolation layer is a set of services for memory, file, process, and environment management. To enable portability of higher layers, these services have the same semantics across various versions of many platforms, such as Microsoft Windows, Apple Macintosh, IBM OS/2, Novell Netware, and several variants of Unix. Data type functions are provided to encapsulate the behavior of primitive data types.

Functions are also available to manage different platform representations and enable binary compatibility of applications, databases, and network messages. Though not strictly a part of the platform isolation layer, there is a network component that insulates higher layers from network architectures and differences in transport-level messaging and a platform-independent UI that insulates higher layers from differences in GUIs and window managers. Isolating a framework or middleware service from a platform using such services is common practice. For example, most portable database systems and portable TP monitors are built this way.

Many framework-specific, platform-independent middleware services sit above the platform isolation layer. These services are accessed via C functions that define the Lotus Notes API. Middleware services include the following:

- object management service, which supports simple data types and bulk information storage;
- document management service, which uses the object management service and names documents, organizes documents, and supports the formula language (a script language for building user applications);
- security service, which supports encryption, digital signatures, and discretionary access control. It uses services in security middleware from RSA Data Security;
- indexing, which provides indexed access to document content (using the content-based retrieval engine from Verity Corp.) and to summary data (i.e., document descriptors);
- messaging, which supports mail transports and mail addressing, including addressing to groups of names and hierarchically organized names;
- events, which support dynamic notification via a mail message or log entry;
- logging, for auditing events on persistent storage
- naming, which supports client-server binding and integrates with transports and security services, much like DCE (Lotus Notes pre-dates DCE);
- system management functions, including add-in (for server application configuration management), installation (of a complete "Lotus Notes server"), and administration (which supports management control from a remote console).

Some traditional middleware functions are performed by independent server processes, which one could classify as either tools or services, such as:

- Replicator—The naming system defines connections between two servers and defines schedules for reconciling their common databases. For each inter-server connection, the replicator copies new and updated entries in each database to the corresponding database(s) on the other server. In the event of a conflict (e.g., independent updates to the same object in both servers), it arbitrates by selecting a "winner" entry and flagging the likely loser.
- Catalog—The catalog produces a global directory of all databases in a Notes network. It is a discovery process that updates a special catalog database with information about local databases. Replication among the catalog replicas results in a global directory.
- Router—The router transfers mail messages between servers.
- Design—Databases are self-describing, in that they contain definitions of all application components as well as the data. Some databases, called templates, contain only definitions. The design process propagates design information in templates or databases between servers, much like the replicator.
- Chronos—This is a scheduler that executes formulas (in the formula language) in the background.
- Event, report—These are administration programs.
- Server—This server does job initiation and scheduling for the preceding servers. There are other built-in server programs, and users can write their own and register them with the add-in middleware service.

A variety of other tools are supported, including Editor for manipulating forms, View for defining and navigating views, and Desk for organizing access to applications. Each tool also includes private services for specific functions, such as spell checking, sending messages, and document format conversions. Lotus Notes is typical of new types of framework in that it uses framework-specific middleware services and a framework-unique API. If other frameworks of this type become popular, we would expect to see common middleware services between those frameworks and a standard API.

Integrating Middleware Services

An important way to add value to a set of middleware services is by integrating them so they work well as a coherent system. For example, one can ensure that they use a common naming architecture, have compatible performance characteristics, support the same international character sets, and execute on the same platforms. These integrated services are often encapsulated in a framework, but they need not be. For example, the OSF DCE is an integrated set of services that is not a framework. Making services work well together is what distinguishes an integrated system from a set of commodity services. The activity of creating such an integrated system out of independently engineered piece-parts is called *system engineering*.

Applications, middleware, and systems can be measured in a variety of dimensions, including usability, distributability, integration, conformance to standards, extensibility, internationalizability, manageability, performance, portability, reliability, scalability, and security. We call these *pervasive attributes*, since they can apply to the system as a whole, not just to the system's components.

Users want to see their applications rate highly on pervasive attributes. Application programmers can help accomplish this by layering their applications on middleware that rates highly on these attributes. System engineers can improve things further by ensuring a set of middleware uniformly attains certain pervasive attributes (by using common naming, compatible performance characteristics, etc.). They may do this by defining building codes for applications to use middleware services in a common way (e.g., common naming conventions), by influencing the design of middleware services (e.g., require that they use a particular security service to authenticate their callers), or by selecting services that are compatible (e.g., trading off performance for portability of some services to ensure the system as a whole has good performance).

Today, system engineering is mostly an ad hoc activity. For some pervasive attributes, such as performance and reliability, there are techniques for measuring, analyzing, and implementing systems to meet specified goals. For most attributes there is little theory or technique to apply, other than common sense and an orderly engineering process. Since the trend is to increase the use of off-the-shelf components to build software systems, more and better techniques for system engineering are urgently needed. Developing such techniques is a major opportunity and challenge for computer systems researchers.

In addition to integrating middleware services into a system, it is important to characterize the result of that integration. For the performance attribute, it is common practice to characterize the system by its behavior on benchmarks. For some attributes, one can characterize the system relative to specific functional capability (e.g., the U.S. Department of Defense "Orange Book" security labels: A1, B2, and so forth). For other attributes, such characterization is more difficult, since there are no generally agreed-upon metrics (e.g., extensibility or ease of use). Developing such metrics is another major research opportunity.

Trends

Although the concept of middleware will be with us for a long time, the specific components that constitute middleware will change over time. Earlier, we explained that some OS services will migrate up to become middleware and some middleware services will migrate down into the OS. Another strong driver of middleware evolution is new application areas, such as mobile computing, groupware, and multimedia. New applications usually have some new requirements that are not met by existing middleware services. Initially, vendors build frameworks to meet these requirements. Over the longer term, when a successful framework-specific service generates demand outside the context of that framework (to be used directly or in other frameworks), it is usually made available as an independent piece of middleware.

There is already too much diversity of middleware for many customers and application developers to cope with. Customers and standards bodies are responding to this problem by developing profiles, which include a subset of middleware services that cover a limited set of essential functions without duplication (i.e., only one service is selected of each type). The X/Open Portability Guide is one especially well-known profile [7]. Unfortunately, different profiles select different services, putting a heavy burden on vendors that want to sell to customers that require different profiles. This hurts users, since vendors dilute their resources by investing in different profiles. Standards groups are becoming more sensitive to this problem and are working harder to coordinate their efforts to avoid a proliferation of competing standards. In the end, the market will sort this out when certain components and profiles become low-cost commodities, making it hard for offbeat components and profiles to compete.

While profiles can simplify a set of middleware, they can also lead to integration problems. Independently designed middleware services may be hard to use together unless certain usage guidelines are adopted, such as common name format and common context (e.g., user and session identifiers). In addition, not all popular implementations of such services may be able to coexist on a platform without some reengineering. For example, they may generate naming conflicts or may require different versions of underlying OS or communications services. Therefore, when defining a profile, one needs a profile-level architecture that addresses these issues. Unfortunately, all too often, profile definers leave this work to vendors and customers to sort out.

Vendors too are responding to the complexity of middleware. They have formed consortia, often jointly with large users, to identify profiles that both vendors and customers need. X/Open and the Object Management Group are examples. Consortia are also working to integrate sets of middleware services, which can then be used as a single component. For example, the OSF DCE integrates RPC with directory, time, security, and file services [6]. OSF is pursuing a similar style effort with its Distributed Management Environment (DME) [3]. Individual vendors are publishing their preferred sets of middleware interfaces, to tell application developers what they can count on. For example, Digital is doing this with its Network Application Support (NAS) [5] and Microsoft with its Windows Open Services Architecture (WOSA).

The complexity of current middleware is untenable over the long term. Therefore, when developing a new middleware service, a vendor must immediately embark on a path to make its service a de facto standard. This seemingly works against the vendor's best interest by prematurely commoditizing the technology. However, application vendors won't rely on a middleware service unless they're confident it will become a de facto standard. Thus, the vendor's only (unappealing) alternative to early standardization is to wait until another vendor follows this path, rendering the first vendor's technology obsolete. Some technologies are defined by independent standards bodies. These standards are truly open and may therefore be implemented by multiple vendors. Since each vendor's implementation supports the same standard functions, vendors can compete by better attainment of pervasive attributes or by extending the functionality in nonstandard, high-value ways. In the latter case, they must now pursue the standardization route for the extensions, or again, application vendors will resist using them.

Large enterprises are already relying on middleware to support their current approximations of an information utility. The trends of simplifying middleware and expanding its functionality into new application areas are likely to increase this reliance in the future.

Glossary of Acronyms

ANSI	American National Standards Institute
API	Application Programming Interface
CAD	Computer-Aided Design
CASE	Computer-Aided Software Engineering
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DBMS	Database Management System
GOSIP	Government Open System Interconnect Profile
GUI	Graphical User Interface
ISO	International Organization for Standardization
ODBC	Open Database Connectivity
OMG	Object Management Group
OS	Operating System
OSF	Open Software Foundation
OSI	Open System Interconnect
RPC	Remote Procedure Call
SQL	Structured Query Language
TP	Transaction Processing
WOSA	Windows Open Services Architecture
4GL	Fourth Generation Language

Acknowledgments

This article arose from several years of work with dozens of engineers on Digital's NAS middleware architecture and products. I am indebted to the late David Stone, who created management focus on the problems addressed here and who encouraged me to pursue them. I thank Wendy Caswell, Scott Davis, Hans Gyllstrom, Chip Nylander, Barry Rubinson, John Miles Smith, Mark Storm, and especially Leo Laverdure for many stimulating discussions about middleware issues and for helping to develop some of the key abstractions in this article. I am grateful to Ken Moore of Iris Associates, who collaborated on the description of Lotus Notes, and to the following people for many suggested improvements to this article: Ed Balkovich, Mark Bramhall, Michael Brodie, John Colonna-Romano, Judy Hall, Meichun Hsu, Hans de Jong, Murray Mazer, and Calton Pu.

References

1. Bernstein, P.A. Transaction processing monitors. *Commun. ACM 33,* 11 (Nov. 1990), 75–86.
2. Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.
3. Chappell, D. The OSF distributed management environment. *ConneXions 6,* 10 (Oct. 1992), 10–15.
4. King, S.S. Middleware. *Data Communications.* (Mar. 1992), 58–67.
5. Laverdure, L., Colonna-Romano, J., Srite, P. *Network Application Support Architecture Reference Manual.* Digital Press, 1993
6. Rosenberry, W., Kenney, D., and Fisher, G. *Understanding DCE.* O'Reilly, Sebastapol, Calif., 1992.
7. *X/Open Portability Guide.* The X/Open Company, Reading, England.

About the Author:

PHILIP A. BERNSTEIN is an architect at Microsoft Corporation. He currently works on repository technology in support of CASE tools. **Author's Present Address:** One Microsoft Way, Redmond, WA 98052-6399; email: philbe@microsoft.com