Middleware
An Architecture for Distributed System Services[1]

Philip A. Bernstein[2]

Digital Equipment Corporation
Cambridge Research Lab

CRL 93/6 March 2, 1993

To help solve heterogeneity and distributed computing problems, vendors are offering distributed system services that have standard programming interfaces and protocols. Standard programming interfaces make it easier to port applications to a variety of platforms, giving the customer some vendor-independence. Standard protocols enable programs to interoperate. These distributed system services are called *middleware*, because they sit ‘‘in the middle,’’ layering above the operating system and networking software and below industry-specific applications. This paper classifies different kinds of middleware, describes their properties, and explains their evolution. The goal is to provide a common vocabulary and conceptual model for understanding today’s and tomorrow’s distributed system software.

CR Categories: D.4.7

Keywords and Phrases: distributed system, framework, information utility, interoperability, middleware, portability, system engineering, transaction processing monitor.

[1]The following trademarks are referenced in this paper: Alpha, AXP, NAS, Open/VMS, PATHWORKS are trademarks of Digital Equipment Corporation. AIX and IBM are registered trademarks of the IBM Corporation. Windows is a trademark of Microsoft Corporation. OSF is a registered trademark of the Open Software Foundation. SPARCstation is a trademark of Sun Microsystems. X/Open is a trademark of X/Open Company Ltd.

[2]Author address: Digital Equipment Corporation, One Kendall Square - Building 700 Cambridge, MA 02139. Internet: pbernstein@crl.dec.com

INTRODUCTION

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 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 to find other users and to find and connect to services of interest. Servers on the wide area network typically support database access, 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 (weather, stock prices, etc.), 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 in the format they need it.

Today's enterprise computing facilities are only an approximation to 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 is uneven. For example, local area servers are often isolated from the wide area network. An appliance can access files and printers on its local server, but often not those on the servers of other local area networks. Sometimes, an application available on one local area server is not available on other servers, because other departments use servers 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 only gain access to a wide area server if a local area server was explicitly programmed for access to the wide area server. 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. And so on.

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 by supporting standard programming interfaces and protocols. Standard programming interfaces make it easier to port applications to a variety of server types, giving the customer some vendor-independence. Increasingly, this is important to server vendors themselves, since customers buy applications, not servers; customers will choose any server that can run the applications they want. 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.

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,* because they sit ''in the middle,'' layering above the OS and networking software and below industry-specific applications.

Like large enterprises, application developers also have heterogeneity and distribution problems for vendors to solve. Developers need to layer their applications 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, thereby allowing 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 to 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 the OS's record-oriented file system and on a remote procedure call mechanism rather than transport-level messaging (e.g., send-message, receive-message). In general, middleware is replacing the non-distributed functions of OSs by distributed functions that use the network. For many applications, the programming interface provided by middleware defines the application's computing environment.

Legacy applications, built before portable middleware became popular, can also benefit from middleware services. One can encapsulate the application as a set of application functions, and use middleware to provide remote access through high-level communications services or provide an advanced user interface through high-level presentation services. Or, one can replace some internal interfaces by functions that call middleware services instead. For example, one can replace application-specific communications functions by middleware communications services. This saves maintenance by relying more on the middleware vendor, and often improves functionality, since the middleware vendor has greater resources to expend on these distributed system functions than the application developer.

This paper classifies different kinds of middleware, describes their properties, and explains their evolution. The goal is to provide a common vocabulary and conceptual model for understanding today's and tomorrow's distributed system software. The paper has five main sections:

- Middleware Services - the distributed system services at the core of middleware
- Frameworks - software environments consisting of many middleware services, often tailored to specific application domains
- TP Monitors, An Example - an important class of framework product that illustrates the concepts of the paper
- System Engineering - the task of integrating middleware into a coherent system
- Trends - where middleware products and architectures are headed.

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 fig. 1). By *platform,* we mean a set of low-level services and processing elements defined by a processor architecture and an OS's application programming interface (API), such as Intel 80486 and Win-32, SUN SPARCstation and SUN OS, IBM RS6000 and AIX, and Alpha AXP and OpenVMS.

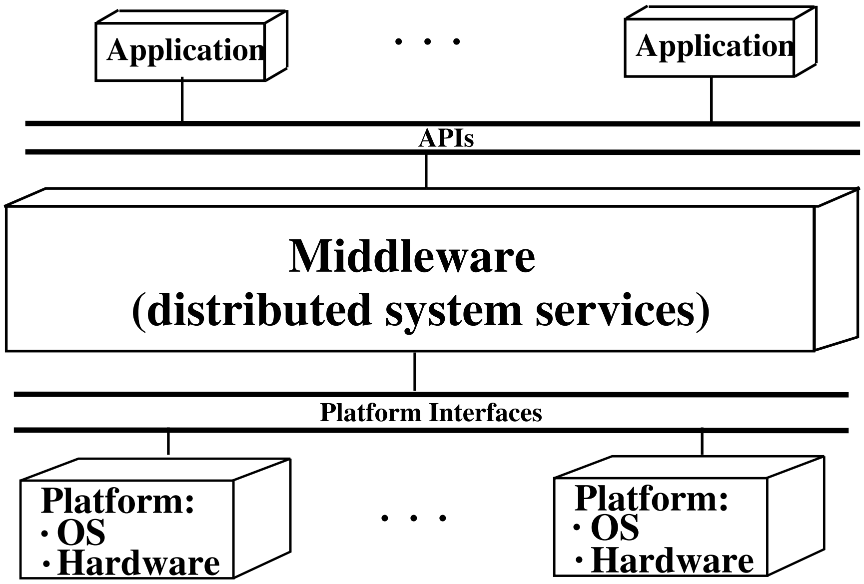

Figure 1 Middleware

A middleware service is defined by the APIs and protocols it supports. It may have multiple implementations that conform to its interface and protocol specifications.

Most middleware services are distributed. That is, a 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.

Most middleware services run on multiple platforms, thereby enhancing the platform coverage of applications that depend on these services. 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 they are ‘‘able to be ported to another platform with modest and predictable effort.’’

Ideally, a middleware service supports a standard protocol, or at least a published one. 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, some database system products have this property.

A middleware service is *transparent* with respect to an API if it can be accessed via that API without modifying that API. Non-transparent middleware requires a new API. Transparent middleware 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.

It is useful to consider middleware services as a separate category of system software, because many software components fit the definition now, and many more will fit in the future. However, our decision whether to classify a service 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 methods,’’ products separate from the OS. Now, they are usually bundled with the OS.

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, accounting, configuration manager, software installation manager, fault detector, recovery coordinator, authentication service, auditing service, encryption service, access controller.

Not all of these services currently are 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 (i.e., presentation, computation, etc.). They are just a convenient way of grouping the services.

The main purpose of middleware services is to help solve many of the problems discussed in the Introduction. 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. Middleware services also arise from the need to 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 middleware services use proprietary APIs (usually making applications dependent on a single vendor's product), proprietary and unpublished protocols (making it difficult for different vendors to build interoperable implementations), and/or are not available on many important platforms (limiting the customer's ability to connect heterogeneous systems). Even when a middleware service is state-of-the-art, an application developer that depends on it has now added a new risk to manage, the risk that the service does not keep pace with technology. For example, many applications that layered on network (e.g., CODASYL) databases have had to be rewritten to benefit from relational database systems that have replaced them as the *de facto* standard.

Second, the large 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 fig. 2). It typically includes a set of middleware services and software tools, and may include a specialized API and user interface. Some popular types of frameworks include office system environments, transaction processing monitors, computer-aided design frameworks, computer-aided software engineering workbenches, and system management workbenches.

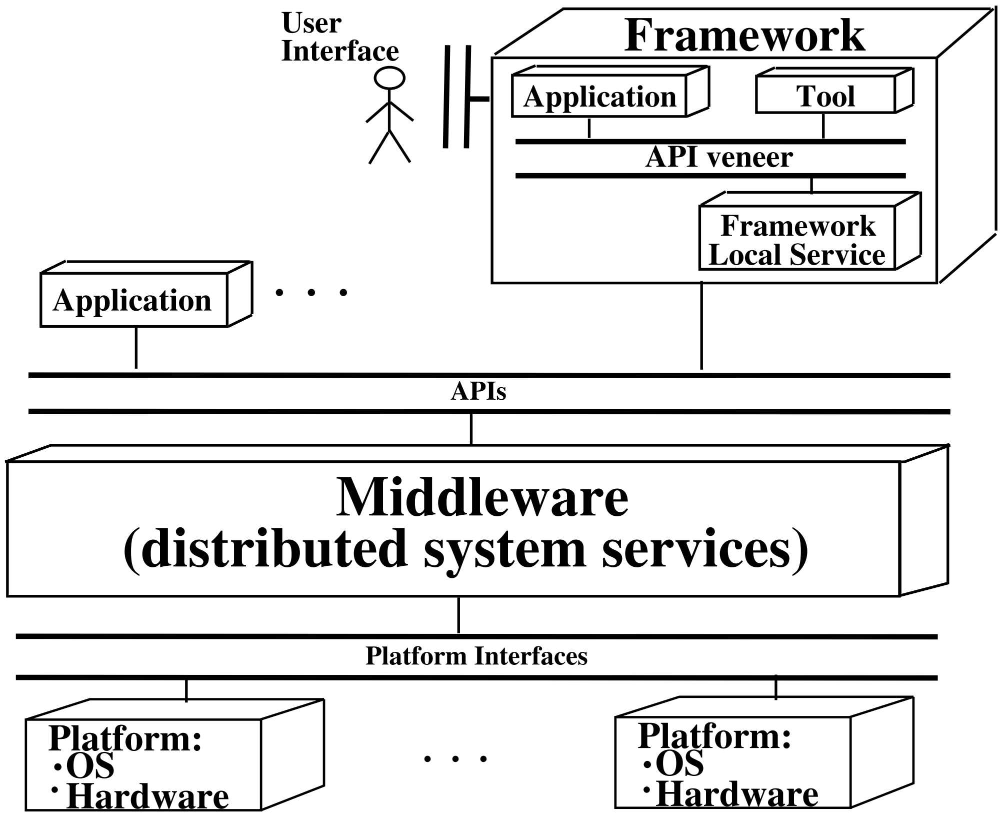

Figure 2 Framework Architecture

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 some of the APIs of underlying middleware services that it abstracts. For example, a CAD or CASE framework's API may consist of a particular presentation service, a particular invocation service (to call tools from the user interface), and a particular 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 significant value by specializing the user interface, simplifying the API by maintaining shared context, or adding framework-specific middleware services.

Since a framework layers on middleware, a framework provider is a customer of middleware. By the same token, an application that layers on a framework is the framework's customer, and only indirectly the middleware's customer. Given the wide variety and complexity of middleware services, a substantial and growing fraction of applications use frameworks 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 middleware services.

A framework may have its own user interface, which is a specialization of the graphical user interfaces (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.

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. 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.

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 transaction processing (TP) monitors use a private implementation of remote procedure call (RPC). However, with the availability of the OSF Distributed Computing Environment (DCE), which includes an RPC, some of these TP monitors are replacing their proprietary RPC implementations with the standard DCE middleware. 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. Example tools include editors, help facilities, forms managers, compilers, debuggers, performance monitors, and software installation managers. Although strictly speaking 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. Other times, the data is equally shared by two or more tools, in which case the tool vendors must negotiate a data format that they all will use. In both cases, tools may share on-line access to the fine-grained data, or they may transfer the data in bulk from tool to tool. If the data is accessed on-line, then the tools may have to agree on protocol, that is, on the set of operations allowed on the data and allowed sequences of those operations.

In control integration, a user or tool that is 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. Sometimes it's useful if the tool executes on a different system than that on which the framework is executing, in which case control integration requires a remote invocation service.

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 by using the same graphical user interface as those tools and by adopting standard usage conventions, such as offering similar semantics to 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 (the spreadsheet's contents) and control (launching the spreadsheet application) integration 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 technique. By contrast, while there are often great benefits to data integration, tools are still useful and easy-to-use without this integration. Also, presentation and control integration are often easier to implement than data integration. For control integration, each tool can be independently integrated by following general guidelines. The same is true for presentation integration, though the ease of integration depends on the tools' use of 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 format is often tedious and expensive.

TP Monitors -- An Example

To illustrate middleware concepts, we will look at one type of framework — transaction processing (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 inter-program communications, queuing, and forms and menu management (see fig. 3).

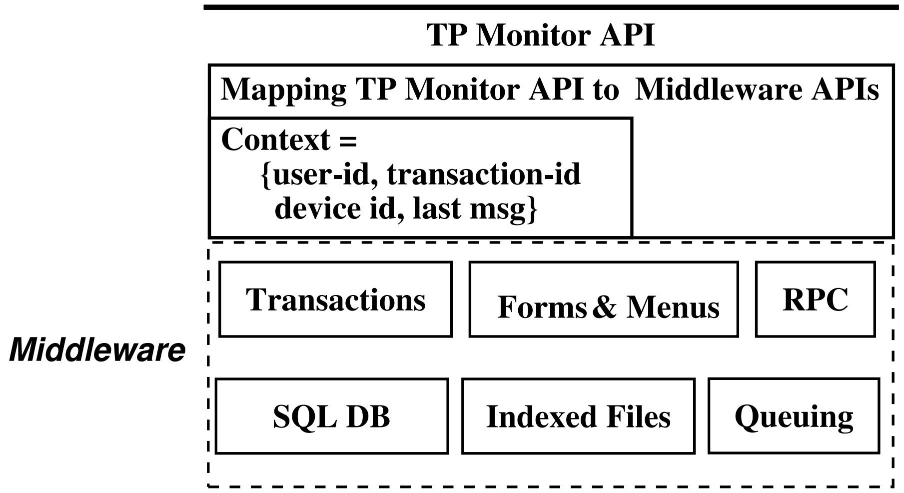

Figure 3 TP Monitor Architecture

Transaction management involves support for operations to 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 so 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 Chapter 7 of [2]).

Transactional inter-program communications supports 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 remote procedure call (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 manager 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 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 (e.g. a database system) or frameworks (e.g., an office system framework).

All of the above 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, ISO is working on a protocol standard for queuing, and CODASYL is working on an API standard for forms management.

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 middleware 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're 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. This is an example of data integration. It also may include a computer-aided software engineering (CASE) framework for simplifying the invocation and use of tools for compilation, software configuration management, and testing. 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 that 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.

System Engineering

Introduction

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 Distributed Computing Environment (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 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 for 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 the computer systems research community.

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 Dept. of Defense ''Orange Book'' security labels: A1, B2, etc.). 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.

Pervasive Attributes

As a first step to developing metrics, we propose the following definitions for the pervasive attributes.

Usability is the extent to which a system or component helps users do their jobs efficiently and effectively, where a ''user'' could be a system manager, programmer, or end-user. Usability can be improved by, for example, improving system behavior relative to other pervasive attributes (e.g., reliability or performance) or improving user interfaces (e.g., simplifying commands or making more effective use of graphics). In a sense, usability is the master attribute, since improving a system relative to any pervasive attribute also improves its usability.

Distributability is the ability of clients and components of a service, tool, or application to execute across multiple hardware components of a network with acceptable cost and efficiency. Ideally, the distribution is transparent, so the application accesses the service in the same way whether the service is local or remote, and if remote, whether it is accessed via a local area or wide area network. It should work across space and time; an application and service are distributed in time if they can exchange information without executing at the same time, e.g., by exchanging messages via a queue. It should work in all phases of the system life cycle (high level design, programming, testing, deployment, execution, etc.).

Integration is the ability of applications to work together in a consistent manner to perform tasks for users of an information system. Integration includes both interoperability and uniformity aspects. *Interoperability* is the extent to which a set of components invokes operations and exchanges information effectively. *Uniformity* is the extent to which a set of components is constant with respect to a set of attributes (e.g., equally distributable, reliable, or secure). Interoperability alone is not sufficient for integration, because it can be attained by gateways that are specific to each combination of components being integrated. Due to lack of uniformity, gateway-based integration often is hard to use and performs poorly.

Conformance to standards is the extent to which software is based on *de jure* or *de facto* standards. By *standard,* customers usually mean “widely available from multiple vendors” or “sold in high volume and therefore inexpensive.” They usually prefer that the standard be defined by an international standard body, but this isn’t required. Customers like standard interfaces and protocols not only because they solve portability and interoperability problems, but also because they promote price competition among vendors that conform, turning computing components into commodity products. System software developers like standards because each standard creates a market for components they can buy and sell. Implementations of standards are an important example of software reusability. Even when these implementations are not purchased as off-the-shelf products, a large vendor often builds such an implementation to avoid having each software product build their own (often non-standard) implementation.

Extensibility is the ease with which a system can be adapted to meet new requirements. This includes the ability to add or modify a program or data type without causing unwanted side effects or requiring changes to existing programs and data.

Internationalizability is the ability to develop applications that can display information in the languages and formats appropriate for all the countries and cultures in which the applications are used. A fully internationalized application does not require re-engineering to support a new language or culture or to support multilingual data (mixtures of languages).

Manageability is the extent to which system managers can economically configure, monitor, diagnose, maintain, and control the resources of a computing environment. Ideally, all resources should be managed in the same way from anywhere in the distributed system across the whole lifecycle of those resources.

Performance is the ability to produce timely results with acceptable cost. Performance factors include response time, efficiency, utilization, and throughput.

Portability is the ease with which software can be moved from one platform to another. This involves designing platform-independent interfaces (to avoid modifying interfaces for each platform) and isolating platform-dependent components (to avoid reimplementing most components when porting to a new platform).

Reliability is the extent to which expected results occur on repeated trials, including the availability of the system to process information when needed. Reliability is measured by mean time between failures (MTBF), and availability by MTBF/(MTBF + MTTR), where MTTR is mean time to repair.

Scalability is the ability to solve both small- and large-scale problems. Some measures of problem size are amount of resources required (e.g., processors, disks, memory), number of users, database size, and input rate.

Security is the protection of information and computer resources from unauthorized use. This involves support for authentication to identify the source of each request, access control to determine if the requester is authorized to perform the requested operation, and auditing to have evidence of every attempt at authentication and request execution.

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 into the OS. Another strong driver of middleware evolution is new applications 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. 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 have to 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 re-engineering. 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).

Long term, the complexity of current middleware is untenable. Therefore, when developing a new middleware service, a vendor must immediately embark on a path to make their 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 non-standard, 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 to simplify middleware and expand its functionality into new application areas are likely to increase this reliance in the future.

Acknowledgments

This paper arose from several years of work with dozens of engineers on Digital's NAS middleware architecture and products. I thank Wendy Caswell, Scott Davis, Hans Gyllstrom, Chip Nylander, Barry Rubinson, John Miles Smith, David Stone, 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 also grateful to the following people for many suggested improvements to this manuscript: Ed Balkovich, Mark Bramhall, John Colonna-Romano, Judy Hall, Mei Hsu, Hans de Jong, and Murray Mazer.

References

1. Bernstein, P. A. Transaction Processing Monitors. *CACM* 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, Steven S. Middleware. *Data Communications,* March 1992, 58-67.
5. Laverdure, L., Colonna-Romano, J., Srite, P. *Network Application Support Architecture Reference Manual*, Digital Press, to appear.
6. Rosenberry, W., Kenney, D, and Fisher , G. *Understanding DCE.* O'Reilly & Assoc., Sebastapol, California, 1992.
7. *X/Open Portability Guide*. The X/Open Company Ltd., Reading, U.K.