An Overview of Middleware

Steve Vinoski*

IONA Technologies, Waltham MA 02451, USA
vinoski@iona.com
http://www.iona.com/hyplan/vinoski/

Abstract. For a variety of reasons, middleware has become a critical substrate for numerous distributed applications. Middleware provides portability, protects applications from the inherent complexities of networking and distribution, supplies generally useful horizontal services (such as directory services, security, and transactions), and shields developers from system-level concerns so they can focus on the logic of their applications. This paper presents an overview of middleware, including its origins, its fundamental aspects, the different evolutionary paths it has taken, the effects that middleware standards have had on that evolution, and where middleware appears to be headed in the future.

1 Introduction

The term "middleware" refers to software that resides between an application and the operating system, network, or database underneath it. Middleware has evolved from simple beginnings—hiding database and network details from applications—into sophisticated systems that handle many important details for distributed applications. Today, middleware serves as a critical part of numerous applications and systems, even in traditionally cautious and exacting vertical industries such as telecommunications, finance, insurance, manufacturing, and government.

The fact that middleware exists at all is rather unsurprising. Its reason for being is to provide a layer between an application and its underlying platform that helps protect the application from the details and quirks of that platform [1]. In essence, it separates concerns by providing useful abstractions that help reduce coupling between software layers. Separating concerns is important; just as operating systems keep users from having to understand the inner workings of their computing hardware just to be able to run applications, middleware keeps developers from having to be experts in specialized areas (such as networking, distributed systems, security, transactions, and fault tolerance) just so they can build robust and reliable applications.

This paper provides an overview of the origins, history, fundamentals, and the various evolutionary paths of middleware. It concludes with a discussion of the directions in which middleware is likely to evolve in the future.

* The author would like to thank Douglas C. Schmidt and Michi Henning for their helpful reviews of drafts of this paper.

A. Llamosí and A. Strohmeier (Eds.): Ada-Europe 2004, LNCS 3063, pp. 35–51, 2004.

2 Middleware Origins

Numerous developments over the history of computing have contributed to modern-day middleware, but its origins can be found in on-line transaction processing (OLTP). In the early 1960s, for example, American Airlines and IBM jointly created the SABRE airline reservation system. This system enabled American to handle their airline reservations on mainframe computers, replacing a system based largely on human agents keeping track of flight reservations on paper index cards, thereby vastly increasing the scalability, reliability, and correctness of their reservation system. Today, the SABRE spin-off is worth more than the airline itself [2].

Though technically not middleware, the original SABRE system provided the foundations for the separation of transaction processing (TP) support from both the application and the operating system. IBM's experiences with SABRE and with developing other reservation systems with other airlines led them in the late 1960s and early 1970s to create several OLTP systems. One of these was the Airline Control Program (ACP), later renamed to the Transaction Processing Facility (TPF). TPF is an operating system built specifically to manage high volumes of transactions for applications. IBM also produced the Customer Information Control System (CICS) [3] and the Information Management System (IMS), which were generally the first systems known as TP monitors [4]. Because of their exceptional reliability, TPF, CICS, and IMS are still in use today for many mission-critical OLTP systems. In the decades that have followed the introduction of these first TP monitors, numerous other successful TP monitors for a variety of different platforms have been developed, including BEA's Tuxedo [5] and Digital Equipment Corporation's Application Control and Management System (ACMS).

TP monitors generally manage computing resources on behalf of transactional applications, and provide services such as database connection management, load balancing, and process monitoring on their behalf. General-purpose operating systems also manage computing resources for applications, but they do so at a basic level and usually provide little assistance, if any, for specific styles of applications. TP monitors, on the other hand, are tuned to provide management and services specifically for transactional applications. Many of the application services that modern middleware provides, as well as the resource management functions it performs, can trace their origins to features of TP monitor systems.

Over time, computing systems have evolved from centralized mainframes to decentralized distributed systems, and along the way they have become increasingly heterogeneous. Heterogeneity is inherent in distributed systems not only because of the never-ending march of technology (which creates continual change as systems are updated and replaced), but also because creation and administration of decentralized systems is itself also decentralized. As a result, different parts of the network will over time consist of different types of machines, operating systems, and applications, and the rate of upgrades and replacements for these components will often differ in various parts the network. This is obviously

true of the Internet, due to its immense size, but this phenomenon also occurs even in small enterprise computing systems that consist of only tens of machines.

As computing systems have become more diverse and heterogeneous, middleware has come to the rescue, restoring a sort of "virtual homogeneity" to the system. When middleware is developed to be portable to a wide variety of platforms, it can provide the same common interfaces and services to applications regardless of where they might execute. It is not uncommon, for example, for the same middleware package to provide the same interfaces and services to embedded applications, applications that run on desktop systems, and to mainframe applications, despite the tremendous differences in these underlying platforms.

3 Middleware Fundamentals 

The general role of middleware is to abstract away the details of underlying facilities and platforms, thereby isolating applications from them. Just as networking protocol stacks can be decomposed into multiple layers, such as the physical, data-link, network, transport, session, presentation, and application layers, so too can middleware be decomposed into the multiple layers [1] shown in Figure 1.[1] Host infrastructure middleware (such as Java Virtual Machines and ACE [6,7]) provides mechanisms to create reusable event demultiplexing, interprocess communication, concurrency, and synchronization objects. Distribution middleware (such as CORBA [8] and Java RMI [9]) defines higher-level distributed programming models. The reusable APIs and objects of these models automate and extend the native OS mechanisms encapsulated by host infrastructure middleware so that applications can invoke operations on target objects without hard-coding dependencies on their location, programming language, OS platform, communication protocols and interconnects, or hardware. Common middleware services (such as directory, transaction, security, and event services) augment distribution middleware by defining higher-level domain-independent reusable services that allow application developers to concentrate on programming business logic, without the need to write the "plumbing" code required to develop distributed applications via lower-level middleware directly. Finally, domain-specific middleware services are tailored to the requirements of particular domains, such as telecommunications, e-commerce, health care, process automation, or aerospace.

Regardless of which layers of middleware are used, good middleware not only makes applications more reliable by shielding them from unnecessary complexity, but it also can make them more scalable. In particular, for applications that must scale well (*e.g.*, in terms of numbers of systems, numbers of simultaneous users, or numbers of requests or operations), middleware reduces the complexities that tend to arise in several areas. These areas are described below.

[1] Figure 1 is provided courtesy of Douglas C. Schmidt.

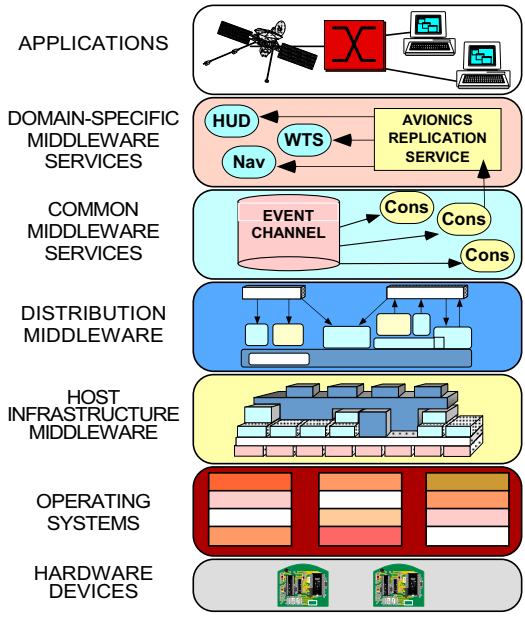

Fig. 1. Middleware Layers in Context

3.1 Communication Support

Most middleware-based systems involve distribution, where applications interact with each other and with distributed services over a network. Modern operating systems typically support network programming interfaces (such as sockets), but these are usually low level and complicated to use. Middleware shields application developers from the complexity of low-level network programming details [6].

There are primarily two forms of middleware-based inter-application distributed communication: *remote procedure call* (RPC) [10] and *messaging*. Each of these is described in further detail below.

Remote Procedure Call. The work on structured programming in the 1970s and 1980s made developers aware of the importance of properly modularizing their applications [11]. Developers grew to understand the benefits of properly dividing their applications into procedures so as to encapsulate units of work. The quality of a procedure was judged by its cohesiveness—the higher, the better—and how minimal its coupling was to its callers and to any procedures it called.

A procedure call reflects a change in computing context, and an RPC is simply the result of abstracting that change in context beyond the limits of a single CPU. RPC allows an application, called the *client*, to invoke services provided by a separate application, called the *server*, via what appears in the client source code to be just an ordinary procedure call. Remote procedures can reside within servers running on the same machine as the client but in separate processes, or they can reside on a wholly separate machine across the network.

The intent of RPC is to hide from the client the details of the distributed aspects of invocations of remote procedures. Some of these details include the following:

• *Data transfer.* A procedure normally has one or more parameters and might also return a value. Passing these values over a network between the client and the server requires that they be converted from their programming language representation into a form suitable for network transmission. Conversion from a programming language representation into a network transmission form is called *marshaling*, and the opposite conversion is called *demarshaling* or *unmarshaling*.

There are numerous complexities and subtleties associated with marshaling. For example, exchanging data between little-endian and big-endian systems requires byte swapping. Some protocols require marshaled data to be aligned in certain ways, such as 4- or 8-byte boundaries. Marshaling complicated data structures, such as cyclic graphs, requires the marshaling subsystem to ensure that graph nodes are marshaled only once when sending, and that links between nodes are properly reconstituted upon demarshaling.

One of the goals of RPC is to provide the client with the illusion that all of its procedure calls, even the remote ones, are just normal invocations made within its own address space. Marshaling must therefore be transparent to the application.[2] Middleware normally achieves this transparency by hiding the marshaling behind stubs or proxies [12] that it supplies to the application. A client invoking a remote procedure actually invokes a local proxy, which in turn performs the marshaling and network operations required to send the request to the actual remote procedure in the server. The local proxy also turns the server's remote response into a normal local procedure call return.

• *Network programming.* To invoke a remote request, one or more network messages must be sent from the client application to the remote server application hosting the target procedure. These messages must contain not only the marshaled data expected by the remote procedure, but must also contain enough information to allow the server application to determine precisely which of its procedures is the target of the invocation. These messages also normally contain header information that indicates the type of message being sent (*e.g.*, a request, a reply, an error, etc.). Together, the header, the information about the target, and the marshaled data make up all the information needed to perform the RPC. The rules regarding which sequences of message types are allowed between sender and receiver (*e.g.*, a receiver may send only a reply or an error in response to a request) collectively define the *RPC protocol*.

RPC protocols can be run over both connection-oriented and connectionless network protocols. Since RPC requires strict request-response semantics, however, when it is performed over connectionless protocols, such as the User Datagram Protocol (UDP), the middleware must take care of dealing with lost

[2] Note, however, that complete local/remote transparency is not possible, *cf.* [13] and the discussion of networking failures on page 41.

and duplicate packets, packet ordering issues, etc., to shield applications from these networking details. For this reason, most modern middleware communications rely on connection-oriented protocols, typically the Transmission Control Protocol (TCP), to avoid handling the low-level networking details associated with UDP.[3] UDP is useful, however, for tasks where lost or duplicate packets are acceptable, such as some forms of logging and event communication.

To actually make an RPC, of course, one or more network transmissions of some sort must occur between the client and server. Middleware that uses connection-oriented transmission protocols must hide all the details of establishing connections from the application. This is normally done by having the application provide some sort of *network handle* to the middleware when it invokes an RPC. The handle contains enough networking information, such as the host and port of the remote target application, to allow the middleware to connect properly. Such connections can either be established *implicitly*, where the middleware transparently creates the connection when the application first invokes a remote procedure using that handle, or *explicitly*, where the application calls a specific middleware API to establish the connection.

For server applications making their procedures available for remote invocation, network programming is even harder. Not only must such applications listen for connections and establish them when requested, but they must be able to do this for numerous callers. Writing scalable and robust server-side code that can handle many simultaneous connections and requests from clients, without unduly sacrificing performance, requires a great deal of expertise. Numerous publications document the details of this expertise [6,14].

Since the APIs offered by operating systems for network access tend to be extremely low-level and obtuse, network programming is quite tedious and error-prone. Polluting applications with these kinds of calls can mean that the application logic is intermixed with networking logic, making the application as a whole very brittle, hard to maintain, and unreliable. Middleware alleviates this issue and restores reliability by taking care of connection establishment and management in a manner that is largely transparent to the application.

• *Failures.* Distribution introduces chances for *partial failure* into a networked application. Unlike a monolithic application that executes in a single address space, where failure is either all or nothing, distributed applications often fail partially. For example, a client application might invoke an RPC, but the machine hosting the server application might fail while the client is waiting for a response. Alternatively, the network between the client and server might partition while the client is waiting. In cases like these, parts of the overall distributed application are left running while other parts have failed. Determining such failure modes and recovering from them can be extremely hard.

Due to partial failure, RPC protocols must provide the ability for the client and server middleware to indicate errors to each other. For example, if in making

[3] In fact, middleware that attempts to support reliable connection-oriented communications over UDP ends up essentially reimplementing TCP.

a request the client middleware somehow violates the RPC protocol, the server must be able to indicate this in an error reply. Middleware must also be able to set timeouts to prevent an application from hanging in the event of a networking problem or a problem with another associated distributed application.

Sometimes middleware can shield applications from the errors inherent in distributed computing. For example, should a transient condition arise in a network that causes a remote invocation to fail, the middleware can sometimes hide this from the application by simply transparently retrying the invocation. If the condition has somehow been corrected in the interim, the retry could succeed, with the result that the application is blissfully unaware that any error ever occurred.

Unfortunately, hiding errors in this manner is not always possible. For example, consider the case where a client invokes an RPC on a server. If a partial failure occurs that prevents the client middleware from knowing whether the server actually executed the procedure call or not, the client middleware must inform the client application that the invocation failed because the target remote procedure might not be *idempotent*, *i.e.*, invoking it more than once might have deleterious effects, such as making multiple withdrawals from a bank account where only one was intended. In cases like these, the middleware has no choice but to return error conditions to the application because only the application can know whether it is safe to retry the operation (unless the middleware has included explicit support for marking remote procedures as being idempotent). Since there is little overlap between the failure modes of local procedure calls and RPCs, local calls and RPCs cannot transparently be treated as being the same [13]. If applications are not properly coded to handle these types of error conditions, their reliability will suffer.

In terms of these three areas that RPC addresses—data transfer, network programming, and failures—note that other approaches that have evolved from RPC, specifically distributed object approaches and remote method invocations, are very similar to RPC, as they too aim to hide these same details. RPC and distributed objects differ mainly in their programming models, in much the same way that C differs from C++.

Messaging. While RPC approaches try to hide network programming details from the application developer as much as possible, messaging takes almost the opposite approach: messaging systems tend to directly and explicitly expose their presence and their APIs to applications. Messaging approaches do not mimic programming language procedure calls or method invocations. Messaging applications therefore need not bother trying to pretend that they are confined to a single address space. Instead, they specifically target inter-application communication.

Messaging APIs tend to be minimal, typically providing a queueing abstraction that applications use to put messages onto and get messages from a queue. Messaging systems often do not provide marshaling support, meaning that applications must perform their own message formatting and interpretation.

Unlike RPCs, which are inherently *synchronous* because they cause the client to wait until the server responds, messaging is *asynchronous*— often called "fire and forget"—because applications simply post messages to queues and continue processing without waiting synchronously for a response. Asynchrony has the primary advantage of providing *loose coupling* because applications work with queues, not directly with other applications. In a client-server system, if a client calls a server that happens to be down, the client is in trouble, as there is little it can do to continue operating correctly. In a messaging system, conversely, messages can be stored persistently in queues until applications become available to retrieve them.

It is possible to implement RPC-like semantics in a messaging system, and vice-versa. For example, RPC can be achieved in a messaging system via the use of two queues, one for request messages and one for replies, while messaging-like semantics can be integrated with RPC systems by implementing queues behind asynchronous proxies.

3.2 Concurrency Support

Many middleware-based applications must be highly scalable, typically in terms of numbers of requests or messages processed per second. This requires maximizing the amount of concurrency in the system, so that it can continously operate on as many tasks as possible at the same time. Middleware systems employ a variety of techniques and patterns to enhance concurrency [14]. For example, employing multiple threads within a server process allows middleware communication subsystems to maximize both the handling of network connections and the handling of requests and messages that arrive on those connections. Depending on the server application characteristics, performance and scalability can depend heavily on the connection handling and request dispatching facilities that the middleware provides.

In addition to multi-threading, concurrency can also involve multiple processes. For example, running only a single instance of a heavily-used server is usually unacceptable not only because of limited throughput, but also because failure of that process means that the service becomes completely unavailable. Running concurrent process replicas allows middleware to balance request load among the replicas, thus providing better service to calling applications. It also allows middleware to divert incoming requests from a failed replica to a live one. Diverting requests in this manner is called *failover*. Quality middleware can perform load balancing and failover transparently for calling applications.

Due to the inherent difficulty of concurrent programming, some middleware systems try to hide its complexities. These approaches usually take the form of frameworks [7] based on patterns [14] that impose rules about what actions applications are allowed to take within certain contexts. Sometimes these rules disallow explicit application invocations of threading APIs, thus allowing the underlying middleware to completely control how the overall application utilizes threading. Such approaches can work for simple applications, but they are usually too inflexible for systems of even moderate complexity.

3.3 Common Middleware Services

In addition to leveraging the communication and concurrency support described above, distributed applications also typically use middleware services to address concerns that are independent of any particular application domain. Some of the most common middleware services are described below.

• *Directory services* allow applications to look up or discover distributed resources. For example, naming services allow applications to find resources based on their names, while trading services provide for discovery based on various resource properties. Directory services avoid the need to hard-code network addresses and other communication details into applications, since hard-coding them would result in a very brittle and hard-to-maintain system that would break anytime host addresses changed or applications were moved to new hosts. Directory services are often federated, meaning that the services are not centralized and are instead implemented across a number of hosts and linked to form a coherent distributed service.

• *Transaction services* help applications commit or roll back their transactions. Such services are critically important in distributed transactional systems, such as those used for financial applications or reservations systems. Middleware for this domain typically takes the form of distributed transaction managers that create and coordinate transactions together with resource managers that are local to each node involved in the transaction.

• *Security services* provide support for authentication and authorization across a distributed system. Such services often coordinate across multiple disparate security systems, supplying single sign-on capabilities that manage credentials transparently across the underlying systems on behalf of applications.

• *Management services* help monitor and maintain systems while they are running, typically in production. Each application in the distributed system is able to log error, warning, and informational messages regarding its operation via a distributed logging service, and to raise alerts to monitoring systems to notify system operators of problems.

• *Event services* allow applications to post event messages for reception by other applications. These can form the basis for the management alerts described above, or for any general publish/subscribe system. In RPC-style systems, event services allow for looser coupling between applications, and can even provide messaging-like persistent queueing capabilities.

• *Persistence services* assist applications with managing their non-volatile data. There are a variety of approaches to providing persistence for applications. Some are oriented around relational databases, others around object-oriented

databases, and still others around non-typical datastores such as persistent key-value pairs or even plain text files. Regardless of which approach is used, middleware persistence abstraction layers hide these underlying storage mechanisms from applications.

• *Load balancing* services direct incoming requests or messages to appropriate service application replicas capable of best handling the request at that point in time. Typically, balancing services track the load of individual service replicas and transparently forward each request or message to the particular replica that is least loaded.

• *Configuration services* add flexibility to applications by allowing middleware capabilities to be modified or augmented non-programmatically, often through management consoles. This type of service allows application behavior, performance, and scalability to be modified and tuned without requiring the application to be recompiled. In some cases, for example, an application can have its security or transactional capabilities enabled or disabled entirely via configuration.

The fact that middleware platforms supply such services relieves applications from having to supply them for themselves. Many of these services, especially those related to security and transactions, require significant expertise and resources to develop correctly. Applications that depend on such services can be smaller, easier to develop, and more flexible, but ultimately their reliability and robustness are only as good as that of the services they rely on.

4 Middleware Evolution

The evolution of middleware has been influenced by numerous developments and standards efforts. Some of these are described in the following sections.

4.1 Early Influences

As described above in section 2, middleware evolved out of early OLTP systems. Along the way, it has also been heavily influenced by programming language developments and by the proliferation of computer networks. By and large, many of today's middleware mechanisms (such as concurrency approaches and process control) have evolved from TP monitors, while middleware abstractions (such as interface definition languages and proxies) have evolved mainly from the area of programming languages.

The influence of programming languages on middleware is not surprising. Developing middleware applications has never been easy, and as a result much effort has been invested in developing abstractions at the programming language level to help ease the development of such applications. Abstractions such as RPC

and its descendant, distributed objects, both resulted directly from programming language research and development.

Systems such as Apollo's Network Computing Architecture (NCA) [15], Sun's Remote Procedure Call standard [16], and the Open Software Foundation's Distributed Computing Environment (DCE) [17] are all examples of successful RPC-oriented middleware that has been used for significant production applications. The success of these RPC systems was relatively limited, however, due in large part to their accidental complexity. In particular, the fact that they were tied mostly to the C programming language made them accessible only to expert developers.

4.2 Distributed Objects and Components 

At the same time these systems were being productized in the 1980s and early 1990s, significant research was occurring in the area of distributed objects. Distributed objects represented the confluence of two key areas of information technology: distributed systems and object-oriented (OO) design and programming. Techniques for developing distributed systems focus on integrating many computing devices to act as a coordinated computational resource. Likewise, techniques for developing OO systems focus on reducing software complexity by capturing successful patterns and creating reusable frameworks based on encapsulation, data abstraction, and polymorphism. Significant early distributed object-based and OO systems included Argus [18], Eden [19], Emerald [20], and COMANDOS [21].

As a result of research on these and other distributed object systems, combined with an evolution of the TP monitor into the concept of an "object monitor," the Common Object Request Broker Architecture (CORBA) was created by the Object Management Group (OMG) in 1991 [8]. CORBA is arguably the most successful middleware standard ever created. Just as a TP monitor controls and manages transactional processes, an object monitor, or *object request broker* (ORB), controls and manages distributed objects and invocations on their methods. A number of commercial, research, and open source ORBs have been implemented in the years since CORBA was first published. On top of these implementations, thousands and thousands of middleware applications have been successfully built and deployed into production. Today, CORBA-based applications serve as critical elements of systems in a wide variety of industries, especially telecommunications, finance, manufacturing, and defense.

From a technical point of view, the success of CORBA is due largely to the balance it provides between flexibility and stringency. CORBA is fairly stringent with respect to the definition of its object model and its invocation rules. Beyond that, it provides a great deal of flexibility for applications, allowing them to be built in a variety of ways. For example, it allows for applications to be built using generated code (in the form of object proxies) based on object interface definitions, which can provide for very high-performance systems, or built using dynamic approaches that can adapt to any object interface, which enables slower but extremely flexible applications. CORBA also supports both RPC

and messaging approaches, the latter through asynchronous proxies as well as through notification and event services. Adding to this flexibility is the fact that CORBA APIs are available in a variety of programming languages, including C, C++, Java, Python, COBOL, and PL/I. This flexibility is critical, given the fact that CORBA is most often used for integration "glue" between disparate systems. CORBA's flexibility has allowed it to be used in an extremely wide variety of applications.

CORBA directly addresses heterogeneous distributed systems using a layered architecture like the one shown in Figure 1. Unfortunately, handling heterogeneous systems sometimes requires approaches that represent the lowest common denominator of the individual systems involved. Indeed, this has been a common criticism of CORBA. For example, the CORBA Interface Definition Language (IDL) has similarities to C++, yet it excludes common OO features, such as method overloading, because of the difficulty of mapping them to non-OO languages such as C [22]. The fact that CORBA objects are defined in IDL—but implemented in an actual programming language—means that the developer is faced with a constant "impedance mismatch" between CORBA constructs and the native constructs of the implementation language. This impedance mismatch can add complexity to the development of CORBA applications [23].

Due to the limitations and trade-offs that dealing with heterogeneity requires at the application development level, some distributed system designers have opted to instead create platforms that are specific to a given programming language. For example, developers using Java Remote Method Invocation (RMI) [9] use normal Java constructs and approaches when developing RMI applications.

4.3 Enterprise Application Integration

While the evolution of distributed objects in the 1990s was marked by significant efforts to establish standards such as CORBA, the evolution of messaging-oriented middleware was practically devoid of standards efforts. In fact, today's most popular messaging systems, such as IBM MQ Series and TIBCO Rendezvous, are proprietary. Even so, messaging systems were widely deployed during this time, primarily because of the loose coupling they afforded, their ease of development with respect to RPC and distributed object systems, and the ease with which they allowed connections to be established between disparate systems.

In the mid to late 1990s, Enterprise Application Integration (EAI) evolved from message-oriented middleware. EAI is characterized as hub-oriented communication between disparate systems, where adapters convert the native formats of the systems being connected into the canonical protocols and formats of the EAI hub.

4.4 Component Models and Web Services

In the late 1990s, the growing popularity of Java, the explosive growth of the World Wide Web (WWW), and lessons learned from CORBA and messaging

middleware were all combined to form the Java 2 Enterprise Edition (J2EE) [24], a comprehensive component middleware platform. J2EE provides support for numerous application types, including distributed objects, components, web-based applications, and messaging systems. For example, it includes Enterprise Java Beans (EJB), which essentially provide a simpler form of the most common CORBA object approaches while adding support for message-driven beans. It also includes a variety of other middleware approaches: servlets and Java Server Pages (JSP) for building web-centric systems, JDBC[4] for database access, Java Message Service (JMS) for messaging support, and the Java Connector Architecture (JCA) for EAI-like functionality. J2EE aimed to introduce some simplicity into the middleware arena, especially with its heavy use of deployment descriptors and other configuration approaches that serve to separate deployment concerns from the actual application code. Unfortunately, despite its features and its homogeneous nature, J2EE remains quite complex. As a result, most uses of J2EE involve only JSPs or servlets and JDBC.

Throughout the development of CORBA, J2EE, and proprietary messaging systems, Microsoft was busy developing its Common Object Model (COM) [25] distributed objects system. COM was seen mainly as a competitor to CORBA, but it was specific to the Windows platform.[5] The main contribution COM made to the overall middleware picture—or more specifically, the tools that were built above COM—was ease of use. Even developers using simple programming languages, such as Visual Basic, could leverage COM's features and support, using them to easily create practical and working distributed applications.

In 1999 and 2000, the Web's influence on middleware started to become readily apparent with the publication of the initial version of SOAP [26]. Initially dubbed the *Simple Object Access Protocol,*[6] SOAP was the result of trying to create a system-agnostic protocol that could be used over the Web and yet still interface easily to non-SOAP middleware, including CORBA, COM, J2EE, and messaging middleware systems.

The introduction of SOAP spawned a new middleware category called *Web Services* [27]. In 2000 and 2001, the development of the SOAP standard in the World Wide Web Consortium (W3C) enjoyed an unprecedented level of industry support. Seemingly every vendor, large and small, was unified behind the creation of the SOAP standard. Given the 1990s "middleware wars" between between J2EE, CORBA, and COM, and between RPC and messaging, the unified support for SOAP was indeed groundbreaking. At the time, it seemed that SOAP and Web Services might finally provide the basis for broad industry agreement on middleware standards, and that ubiquitous application interoperability

[4] JDBC originally stood for "Java Database Connectivity." It is now a trademark of Sun Microsystems.

[5] Efforts to port COM to other operating systems besides Windows were made, but they never achieved commercial success.

[6] "Simple Object Access Protocol" was a poor name, given that SOAP is neither simple nor object-oriented. This name was eventually dropped in favor of the name *SOAP*.

and portability would soon follow. Not surprisingly, this was not to be, given the substantial inter-company political and business conflicts in this market. Despite broad agreement around SOAP and the Web Services Description Language (WSDL), which is used to define Web Services interfaces [28], significant disagreements remain over what Web Services are and how they are best applied [29,30].

5 The Future of Middleware

Web Services are clearly the immediate future of middleware. This does not mean, however, that Web Services will displace older approaches, such as J2EE and CORBA. Rather, Web Services complements these earlier successful middleware technologies.

Initial Web Services developments were heavily RPC-oriented, and were touted as possible replacements for more complicated EJBs or CORBA objects. However, the performance profile of Web Services is such that they are often several orders of magnitude slower than CORBA when used for fine-grained distributed resource access. As a result, the use of Web Services for RPC systems is now considered much less significant than using them for document-oriented or message-oriented systems.

Rather than trying to replace older approaches, today's Web Services developments are instead oriented around middleware integration [31], thereby adding value to existing messaging, J2EE, and CORBA middleware systems. WSDL allows developers to abstractly describe Web Service interfaces while also defining the concrete *bindings*—the protocols and transports—required at runtime to access the services. For example, projects such as the Apache Web Services Invocation Framework (WSIF; `http://ws.apache.org/wsif/`) aim to allow applications to access Web Services transparently via EJB, JMS, JCA, or as local Java objects. This move towards integration allows services implemented in these different technologies to exposed as Web Services and made available to a variety of client applications. Middleware integration is truly the sweet spot for Web Services applications for the foreseeable future.

Web Services standardization efforts continue, with various standards bodies working on standards for security, transactions, addressing, orchestration, choreography, reliable messaging, business processes, and management. Unfortunately, these standardization efforts are currently splintered across a number of standards bodies. Still, even splintered standards efforts are usually better than none. For example, because EAI has no associated standards, Web Services are quickly moving to wholly displace EAI because customers view standards as affording them at least some degree of freedom from vendor lock-in. EAI thus appears to be a middleware evolutionary dead end.

Middleware also has a bright future for distributed real-time and embedded (DRE) systems [32] This domain is traditionally extremely conservative because of the stringent requirements for predictability, size, and performance in DRE systems. The construction of middleware suitable for such systems is now well

understood and standardized (*e.g.*, via the Real-time CORBA and Real-time Java specifications), meaning that middleware benefits are now available for DRE systems. This, combined with continuing advances in hardware such as higher data bus and CPU clock speeds, smaller circuits, and larger memory capacities, promises to advance the capabilities of many real-time and embedded systems and allow DRE techniques to be applied to new areas.

Another recent middleware development, grid technology [33], is attempting to provide distributed infrastructure for "utility computing." The goal of grid technology is to create a computing grid that is as ubiquitous and invisible as the power grid. Users of grid middleware infrastructure hope to take advantage of continuous increases in compute power, network speed, and storage capacities to make distributed access to computing resources easier and faster. Companies such as IBM, HP, and Sun are working hard to achieve grid computing because they view it as a way to sell their next generation of high-end compute servers. Work is also being done to unite grid approaches with Web Services technologies to create the Open Grid Services Architecture (OGSA) [34], which promises to make grid distributed services available via Web Services interfaces.

Grid technologies count on the existence and ubiquity of very high speed networks to minimize latency when accessing remote resources. As a result, the potential for success of grids is currently unclear, as such networks are still under development in research testbeds and are not yet ubiquitously available in the commercial sector.

Commercial-off-the-shelf (COTS) middleware platforms have traditionally expected static connectivity, reliable communication channels, and relatively high bandwidth. Significant challenges remain, however, to design, optimize, and apply middleware for more flexible network environments, such as self-organizing peer-to-peer (P2P) networks, mobile settings, and highly resource-constrained sensor networks. For example, hiding network topologies and other deployment details from networked applications becomes harder (and often undesirable) in wireless sensor networks since applications and middleware often need to adapt according to changes in location, connectivity, bandwidth, and battery power. Concerted R&D efforts [35] are therefore essential to devising new middleware solutions and capabilities that can fulfill the requirements of these emerging network technologies and next-generation applications.

6 Concluding Remarks

This paper has presented a broad overview of middleware, describing its evolution from transaction processing systems into RPC systems, message queuing systems, distributed object systems, and web-based service applications. Covering the entire middleware spectrum would require significantly more pages than this format allows, but this paper has attempted to accent the most significant developments in the history of middleware and show the effects of those developments on middleware evolution. Due to its separation of concerns, middleware will continue to provide performance, scalability, portability, and reliability for

applications while insulating them from many accidental and inherent complexities of underlying operating systems, networks, and hardware.

References

1. R. Schantz and D. Schmidt. "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," *Encyclopedia of Software Engineering*, Wiley and Sons, 2002.
2. C. Zook and J. Allen. "Growth Outside the Core." *Harvard Business Review*, 81(12), December 2003, pp. 66–73.
3. B. Yelavich. "Customer Information Control System—An Evolving System Facility." *IBM Systems Journal*, 24(3/4), 1985, pp. 264–278.
4. P. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann, 1997.
5. J. Andrade, T. Dwyer, S. Felts, M. Carges, *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications*, Addison-Wesley, 1996.
6. D. Schmidt and S. Huston, *C++ Network Programming, Vol. 1: Mastering Complexity with ACE and Patterns*, Addison-Wesley, 2002.
7. D. Schmidt and S. Huston, *C++ Network Programming, Vol. 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, 2003.
8. Object Management Group. "The Common Object Request Broker: Architecture and Specification (CORBA)," OMG document number 91-12-1, 1991.
9. Javasoft, "Java Remote Method Invocation—Distributed Computing for Java, a White Paper," available online at `http://java.sun.com/marketing/collateral/javarmi.html`.
10. A. Birrell and B. Nelson. "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, 2(1), February 1984, pp. 114–122.
11. D. Parnas, "On The Criteria To Be Used In Decomposing Systems Into Modules," *Communications of the ACM*, 15(12), 1972, pp. 1053–1058.
12. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
13. J. Waldo, G. Wyant, A. Wollrath, S. Kendall, "A Note on Distributed Computing." Technical Report SMLI TR-94-29. Sun Microsystems Laboratories, Inc, 1994.
14. D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*, John Wiley & Sons, Ltd, 200.
15. L. Zahn, T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, & G. Wyant, *Network Computing Architecture*, Prentice-Hall, 1990.
16. Sun Microsystems, "RPC: Remote Procedure Call Protocol Specification," Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.
17. W. Rosenberry, D. Kenney, and G. Fischer. 1992. *Understanding DCE*, O'Reilly and Associates, Inc.
18. B. Liskov, "The Argus Language and System," In M. Paul and H.J. Siegert, editor, Distributed Systems—Methods and Tools for Specification, Lecture Notes in Computer Science no. 190, Springer Verlag, 1985, pp. 343–430.
19. G. Almes, A. Black, E. Lazowska and J. Noe, "The Eden System: A technical Review," *IEEE Transactions on Software Engineering*, SE-11(1), 1985, pp. 43–59.
20. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, SE-13(1), 1987, pp. 65–76.

21. C. Horn and S. Krakowiak. "Object Oriented Architecture for Distributed Office Systems." In ESPRIT '87: Achievements and Impact, North-Holland, 1987.
22. S. Vinoski. "Toward Integration: It's Just a Mapping Problem," *IEEE Internet Computing*, 7(3), May/June 2003, pp. 88–90.
23. M. Henning. "A New Approach to Object-Oriented Middleware," *IEEE Internet Computing*, 8(1), 2004, pp. 66–75.
24. Sun Microsystems. *J2EE 1.4 Platform Specification.* Available online at `http://java.sun.com/j2ee/`, November 2003.
25. D. Box, *Essential COM*, Addison-Wesley, 1998.
26. World Wide Web Consortium (W3C). 2003. "SOAP Version 1.2." Available online at `http://www.w3.org/TR/SOAP/`.
27. E. Newcomer, *Understanding Web services: XML, WSDL, SOAP, and UDDI*, Addison-Wesley, 2002.
28. World Wide Web Consortium. "Web Services Description Language(WSDL) Version 2.0 Part 1: Core Language," W3C Working Draft, 10 November 2003, available online at `http://www.w3.org/TR/wsdl20/`.
29. S. Vinoski, "Toward Integration: Web Services Interaction Models, Part 1: Current Practice," *IEEE Internet Computing*, 6(3), May/June 2002, pp. 89–91.
30. S. Vinoski. "Toward Integration: Web Services Interaction Models, Part 2: Putting the 'Web' Into Web Services," *IEEE Internet Computing*, 6(4), July/Aug 2002, pp. 90–92.
31. S. Vinoski, "Toward Integration: Integration With Web Services," *IEEE Internet Computing*, 7(6), Nov/Dec 2003, pp. 75–77.
32. D. C. Schmidt and F. Kuhns, "An Overview of the Real-time CORBA Specification," *IEEE Computer Magazine*, 33(6), June 2000, pp. 56–63.
33. I. Foster, "The Grid: A New Infrastructure for 21st Century Science," *Physics Today*, 55(2), February 2002, available online at `http://www.aip.org/pt/vol-55/iss-2/p42.html`.
34. I. Foster, C. Kesselman, J. Nick, S. Tuecke, "Grid Services for Distributed System Integration." *IEEE Computer*, 35(6), June 2002, pp. 37–46.
35. *IEEE Network* Special Issue on Middleware Technologies for Future Communication Networks, Edited by D. Schmidt, G. Blair, and A. Campbell, Vol 18, No. 1, January/February 2004.