

M

MIDDLEWARE FOR DISTRIBUTED SYSTEMS

MIDDLEWARE IS PART OF A BROAD SET OF INFORMATION TECHNOLOGY TRENDS

Middleware represents the confluence of two key areas of information technology (IT): distributed systems and advanced software engineering. Techniques for developing distributed systems focus on integrating many computing devices to act as a coordinated computational resource. Likewise, software engineering techniques for developing component-based systems focus on reducing software complexity by capturing successful patterns of interactions and creating reusable frameworks for integrating these components. Middleware is the area of specialization dealing with providing environments for developing systems that can be distributed effectively over a myriad of topologies, computing devices, and communication networks. It aims to provide developers of networked applications with the necessary platforms and tools to (1) formalize and coordinate how parts of applications are composed and how they interoperate and (2) monitor, enable, and validate the (re)configuration of resources to ensure appropriate application end-to-end quality of service (QoS), even in the face of failures or attacks.

During the past few decades, we have benefited from the commoditization of hardware (such as CPUs and storage devices), operating systems (such as UNIX and Windows), and networking elements (such as IP routers). More recently, the maturation of software engineering focused programming languages (such as Java and C++), operating environments (such as POSIX and Java Virtual Machines), and enabling fundamental middleware based on previous middleware R&D (such as CORBA, Enterprise Java Beans, and SOAP/Web services) are helping to commoditize many common-off-the-shelf (COTS) software components and architectural layers. The quality of COTS software has generally lagged behind hardware, and more facets of middleware are being conceived as the complexity of application requirements increases, which has yielded variations in maturity and capability across the layers needed to build working systems. Nonetheless, improvements in software frameworks (1), patterns (2,3), component models (4), and development processes (5) have encapsulated the knowledge that enables COTS software to be developed, combined, and used in an increasing number of real-world applications, such as e-commerce websites, avionics mission computing, command and control systems, financial services, and integrated distributed sensing, to name but a few.

Some notable successes in middleware for distributed systems include:

- Distributed Object Computing (DOC) middleware (6–10) (such as CORBA, Java RMI, SOAP), which pro-

vides a support base for objects that can be dispersed throughout a network, with clients invoking operations on remote target objects to achieve application goals. Much of the network-oriented code is tool generated using a form of interface definition language and compiler.

- Component middleware (11) (such as Enterprise Java Beans, the CORBA Component Model, and .NET), which is a successor to DOC approaches, focused on composing relatively autonomous, mixed functionality software elements that can be distributed or collocated throughout a wide range of networks and interconnects, while extending the focus and tool support toward lifecycle activities such as assembling, configuring, and deploying distributed applications.
- World Wide Web middleware standards (such as web servers, HTTP protocols, and web services frameworks), which enable easily connecting web browsers with web pages that can be designed as portals to powerful information systems.
- Grid computing (12) (such as Globus), which enables scientists and high-performance computing researchers to collaborate on grand challenge problems, such as global climate change modeling.

Within these middleware frameworks, a wide variety of services are made available off-the-shelf to simplify application development. Aggregations of simple, middleware-mediated interactions form the basis of large-scale distributed systems.

MIDDLEWARE ADDRESSES KEY CHALLENGES OF DEVELOPING DISTRIBUTED SYSTEMS

Middleware is an important class of technology that is helping to decrease the cycle time, level of effort, and complexity associated with developing high-quality, flexible, and interoperable distributed systems. Increasingly, these types of systems are developed using reusable software (middleware) component services, rather than being implemented entirely from scratch for each use. When implemented properly, middleware can help to:

- Shield developers of distributed systems from low-level, tedious, and error-prone platform details, such as socket-level network programming.
- Amortize software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use.
- Provide a consistent set of higher-level network-oriented abstractions that are much closer to application and system requirements to simplify the development of distributed systems.

- Provide a wide array of developer-oriented services, such as logging and security that have proven necessary to operate effectively in a networked environment.

Middleware was invented in an attempt to help simplify the software development of distributed computing systems, and bring those capabilities within the reach of many more developers than the few experts at the time who could master the complexities of these environments (7). Complex system integration requirements were not being met from either the *application perspective*, in which it was too hard and not reusable, or the *network or host operating system perspectives*, which were necessarily concerned with providing the communication and endsystem resource management layers, respectively.

Over the past decade, middleware has emerged as a set of software service layers that help to solve the problems specifically associated with heterogeneity and interoperability. It has also contributed considerably to better environments for building distributed systems and managing their decentralized resources securely and dependably. Consequently, one of the major trends driving industry involves moving toward a multi-layered architecture (applications, middleware, network, and operating system infrastructure) that is oriented around application composition from reusable components and away from the more traditional architecture, where applications were developed directly atop the network and operating system abstractions. This middleware-centric, multi-layered architecture descends directly from the adoption of a network-centric viewpoint brought about by the emergence of the Internet and the componentization and commoditization of hardware and software.

Successes with early, primitive middleware, such as message passing and remote procedure calls, led to more ambitious efforts and expansion of the scope of these middleware-oriented activities, so we now see a number of distinct layers taking shape within the middleware itself, as discussed in the following section.

MIDDLEWARE HAS A LAYERED STRUCTURE, JUST LIKE NETWORKING PROTOCOLS

Just as networking protocol stacks are decomposed into multiple layers, such as the physical, data-link, network, and transport, so too are middleware abstractions being decomposed into multiple layers, such as those shown in Fig. 1.

Below, we describe each of these middleware layers and outline some of the technologies in each layer that have matured and found widespread use in COTS platforms and products in recent years.

Host Infrastructure Middleware

Host infrastructure middleware leverages common patterns (3) and best practices to encapsulate and enhance native OS communication and concurrency mechanisms to create reusable network programming components, such as reactors, acceptor-connectors, monitor objects, active objects, and component configurators (13,14). These components abstract away the peculiarities of individual operating systems, and help eliminate many tedious, error-prone, and nonportable aspects of developing and maintaining networked applications via low-level OS programming APIs, such as Sockets or POSIX pthreads. Widely used examples of host infrastructure middleware include:

- The Sun Java Virtual Machine (JVM) (15), which provides a platform-independent way of executing code by abstracting the differences between operating systems and CPU architectures. A JVM is responsible for interpreting Java bytecode and for translating the bytecode into an action or operating system call. It is the JVM's responsibility to encapsulate platform details within the portable bytecode interface, so that applications are shielded from disparate operating systems and CPU architectures on which Java software runs.

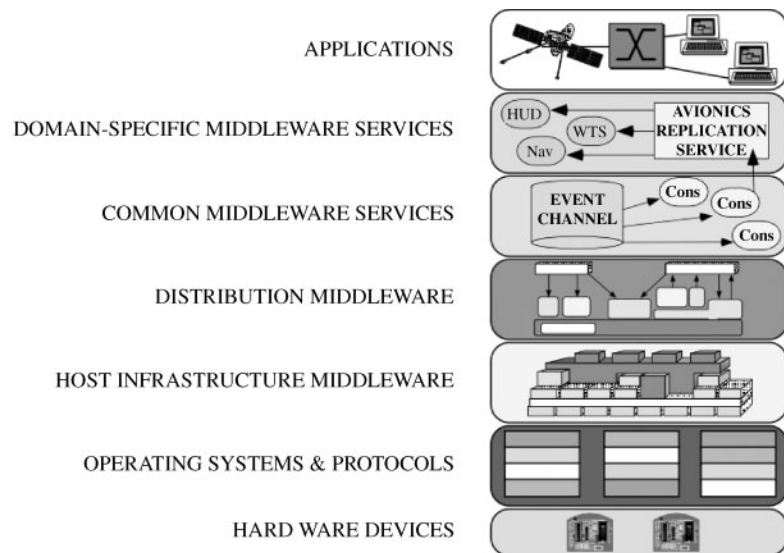


Figure 1. Layers of middleware and surrounding context.

- .NET (16) is Microsoft's platform for XML Web services, which are designed to connect information, devices, and people in a common, yet customizable way. The common language runtime (CLR) is the host infrastructure middleware foundation for Microsoft's .NET services. The CLR is similar to Sun's JVM (i.e., it provides an execution environment that manages running code and simplifies software development via automatic memory management mechanisms, cross-language integration, interoperability with existing code and systems, simplified deployment, and a security system).
- The Adaptive Communication Environment (ACE) (13,14) is a highly portable toolkit written in C++ that encapsulates native OS network programming capabilities, such as connection establishment, event demultiplexing, interprocess communication, (de)marshaling, concurrency, and synchronization. The primary difference between ACE, JVMs, and the .NET CLR is that ACE is always a compiled interface rather than an interpreted bytecode interface, which removes another level of indirection and helps to optimize runtime performance.
- Microsoft's Distributed Component Object Model (DCOM) (19), which is distribution middleware that enables software components to communicate over a network via remote component instantiation and method invocations. Unlike CORBA and Java RMI, which run on many OSs, DCOM is implemented primarily on Windows.
- SOAP (20), which is an emerging distribution middleware technology based on a lightweight and simple XML-based protocol that allows applications to exchange structured and typed information on the Web. SOAP is designed to enable automated Web services based on a shared and open Web infrastructure. SOAP applications can be written in a wide range of programming languages, used in combination with a variety of Internet protocols and formats (such as HTTP, SMTP, and MIME), and can support a wide range of applications from messaging systems to RPC.

Distribution Middleware

Distribution middleware defines higher-level distributed programming models whose reusable APIs and components automate and extend the native OS network programming capabilities encapsulated by host infrastructure middleware. Distribution middleware enables clients to program distributed systems much like stand-alone applications (i.e., by invoking operations on target objects without hard-coding dependencies on their location, programming language, OS platform, communication protocols and interconnects, and hardware). At the heart of distribution middleware are request brokers, such as:

- The OMG's Common Object Request Broker Architecture (CORBA) (6) and the CORBA Component Model (CCM) (17), which are open standards for distribution middleware that allows objects and components, respectively, to interoperate across networks regardless of the language in which they were written or the platform on which they are deployed. The OMG Real-time CORBA (RT-CORBA) specification (18) extends CORBA with features that allow real-time applications to reserve and manage CPU, memory, and networking resources.
- Sun's Java Remote Method Invocation (RMI) (10), which is distribution middleware that enables developers to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other JVMs, possibly on different hosts. RMI supports more sophisticated object interactions by using object serialization to marshal and unmarshal parameters as well as whole objects. This flexibility is made possible by Java's virtual machine architecture and is greatly simplified by using a single language.
- The OMG's CORBA Common Object Services (CORBAServices) (21), which provide domain-independent interfaces and capabilities that can be used by many distributed systems. The OMG CORBAServices specifications define a wide variety of these services, including event notification, logging, multimedia streaming, persistence, security, global time, real-time scheduling, fault tolerance, concurrency control, and transactions.
- Sun's Enterprise Java Beans (EJB) technology (22), which allows developers to create n-tier distributed systems by linking a number of pre-built software services—called “beans”—without having to write much code from scratch. As EJB is built on top of Java technology, EJB service components can only be implemented using the Java language. The CCM

Common Middleware Services

Common middleware services augment distribution middleware by defining higher-level domain-independent services that allow application developers to concentrate on programming business logic, without the need to write the “plumbing” code required to develop distributed systems by using lower-level middleware directly. For example, application developers no longer need to write code that handles naming, transactional behavior, security, database connection, because common middleware service providers bundle these tasks into reusable components. Whereas distribution middleware focuses largely on connecting the parts in support of an object-oriented distributed programming model, common middleware services focus on allocating, scheduling, coordinating, and managing various resources end-to-end throughout a distributed system using a component programming and scripting model. Developers can reuse these component services to manage global resources and perform common distribution tasks that would otherwise be implemented in an ad hoc manner within each application. The form and content of these services will continue to evolve as the requirements on the applications being constructed expand. Examples of common middleware services include:

(17) defines a superset of EJB capabilities that can be implemented using all the programming languages supported by CORBA.

- Microsoft's .NET Web services (16), which complements the lower-level middleware .NET capabilities and allows developers to package application logic into components that are accessed using standard higher-level Internet protocols above the transport layer, such as HTTP. The .NET Web services combine aspects of component-based development and Web technologies. Like components, .NET Web services provide black-box functionality that can be described and reused without concern for how a service is implemented. Unlike traditional component technologies, however, .NET Web services are not accessed using the object model-specific protocols defined by DCOM, Java RMI, or CORBA. Instead, XML Web services are accessed using Web protocols and data formats, such as HTTP and XML, respectively.

Domain-Specific Middleware Services

Domain-specific middleware services are tailored to the requirements of particular domains, such as telecom, e-commerce, health care, process automation, or aerospace. Unlike the other three middleware layers, which provide broadly reusable "horizontal" mechanisms and services, domain-specific middleware services are targeted at vertical markets. From a COTS perspective, domain-specific services are the least mature of the middleware layers today. This immaturity is due partly to the historical lack of distribution middleware and common middleware service *standards*, which are needed to provide a stable base upon which to create domain-specific services. As they embody knowledge of a domain, however, domain-specific middleware services have the most potential to increase system quality and decrease the cycle time and effort required to develop particular types of networked applications. Examples of domain-specific middleware services include the following:

- The OMG has convened a number of Domain Task Forces that concentrate on standardizing domain-specific middleware services. These task forces vary from the *Electronic Commerce Domain Task Force*, whose charter is to define and promote the specification of OMG distributed object technologies for the development and use of electronic commerce and electronic market systems, to the *Life Science Research Domain Task Force*, who do similar work in the area of life science, maturing the OMG specifications to improve the quality and utility of software and information systems used in life sciences research. There are also OMG Domain Task Forces for the health-care, telecom, command and control, and process automation domains.
- The Siemens Medical Solutions Group has developed *syngo*. (See <http://www.syngo.com>), which is both an integrated collection of domain-specific middleware services as well as an open and dynamically extensible application server platform for medical imaging tasks

and applications, including ultrasound, mammography, radiography, magnetic resonance, patient monitoring systems, and life support systems. The *syngo* middleware services allow health-care facilities to integrate diagnostic imaging and other radiological, cardiological, and hospital services via a black-box application template framework based on advanced patterns for communication, concurrency, and configuration for business and presentation logic supporting a common look and feel throughout the medical domain.

OVERARCHING BENEFITS OF MIDDLEWARE

The various layers of middleware described in the previous section provide essential capabilities for developing and deploying distributed systems. This section summarizes the benefits of middleware over traditional non-middleware approaches.

Growing Focus on Integration Rather than on Programming

This visible shift in focus is perhaps the major accomplishment of currently deployed middleware. Middleware originated because the problems relating to integration and construction by composing parts were not being met by *applications*, which at best were customized for a single use; *networks*, which were necessarily concerned with providing the communication layer; or *host operating systems*, which were focused primarily on a single, self-contained unit of resources. In contrast, middleware has a fundamental integration focus, which stems from incorporating the perspectives of both OSs and programming model concepts into organizing and controlling the composition of separately developed components across host boundaries. Every middleware technology has within it some type of request broker functionality that initiates and manages intercomponent interactions.

Distribution middleware, such as CORBA, Java RMI, or SOAP, makes it easy and straightforward to connect separate pieces of software together, largely independent of their location, connectivity mechanism, and the technology used to develop them. These capabilities allow middleware to amortize software lifecycle efforts by leveraging previous development expertise and reifying implementations of key patterns into more encompassing reusable frameworks and components. As middleware continues to mature and incorporate additional needed services, next-generation applications will increasingly be assembled by modeling, integrating, and scripting domain-specific and common service components, rather than by being programmed from scratch or requiring significant customization or augmentation to off-the-shelf component implementations.

Focus on End-to-End Support and Integration, Not Just Individual Components

There is now widespread recognition that effective development of large-scale distributed systems requires the use of COTS infrastructure and service components. Moreover,

the usability of the resulting products depends heavily on the weaving of the properties of the whole as derived from its parts. In its most useful forms, middleware provides the end-to-end perspective extending across elements applicable to the network substrate, the platform OSs and system services, the programming system in which they are developed, the applications themselves, and the middleware that integrates all these elements together.

The Increased Viability of Open Systems Architectures and Open-Source Availability

By their very nature, distributed systems developed by composing separate components are more open than systems conceived and developed as monolithic entities. The focus on interfaces for integrating and controlling the component parts leads naturally to *standard* interfaces, which, in turn, yields the potential for multiple choices for component implementations and open engineering concepts. Standards organizations, such as the OMG, The Open Group, Grid Forum, and the W3C, have fostered the cooperative efforts needed to bring together groups of users and vendors to define domain-specific functionality that overlays open integrating architectures, forming a basis for industry-wide use of some software components. Once a common, open structure exists, it becomes feasible for a wide variety of participants to contribute to the off-the-shelf availability of additional parts needed to construct complete systems. As few companies today can afford significant investments in internally funded R&D, it is increasingly important for the IT industry to leverage externally funded R&D sources, such as government investment. In this context, standards-based middleware serves as a common platform to help concentrate the results of R&D efforts and ensure smooth transition conduits from research groups into production systems.

For example, research conducted under the DARPA Quorum, PCES, and ARMS programs focused heavily on CORBA open systems middleware. These programs yielded many results that transitioned into standardized service definitions and implementations for CORBA's real-time (9,18), fault-tolerant (23,24), and components (17) specifications and productization efforts. In this case, focused government R&D efforts leveraged their results by exporting them into, and combining them with, other on going public and private activities that also used a standards-based open middleware substrate. Before the viability of common middleware platforms, these same results would have been buried within a custom or proprietary system, serving only as the existence proof, not as the basis for incorporating into a larger whole.

Advanced Common Infrastructure Sustaining Continuous Innovation

Middleware supporting component integration and reuse is a key technology to help amortize software lifecycle costs by leveraging previous development expertise (e.g., component middleware helps to abstract commonly reused low-level OS concurrency and networking details away into higher-level, more easily used artifacts). Likewise, middleware also focus efforts to improve software quality and

performance by combining aspects of a larger solution together (e.g., component middleware combines fault tolerance for domain-specific elements with real-time QoS properties).

When developers need not worry as much about low-level details, they are freed to focus on more strategic, larger scope, application-centric specializations concerns. Ultimately, this higher-level focus will result in software-intensive distributed system components that apply reusable middleware to get smaller, faster, cheaper, and better at a predictable pace, just as computing and networking hardware do today, which, in turn, will enable the next generation of better and cheaper approaches to what are now carefully crafted custom solutions, which are often inflexible and proprietary. The result will be a new technological paradigm where developers can leverage *frequently used common components*, which come with steady innovation cycles resulting from a multi-user basis, in conjunction with *custom domain-specific components*, which allow appropriate mixing of multi-user low cost and custom development for competitive advantage.

KEY CHALLENGES AND OPPORTUNITIES FOR NEXT-GENERATION MIDDLEWARE

This section presents some of the challenges and opportunities for next-generation middleware. One such challenge is in supporting new trends toward distributed "systems of systems," which include many interdependent levels, such as network/bus interconnects, embedded local and geographically distant remote endsystems, and multiple layers of common and domain-specific middleware. The desirable properties of these systems of systems, both individually and as a whole, include predictability, controllability, and adaptability of operating characteristics with respect to such features as time, quantity of information, accuracy, confidence, and synchronization. All these issues become highly volatile in systems of systems, because of the dynamic interplay of the many interconnected parts. These parts are often constructed in a similar way from smaller parts.

Many 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 because applications and middleware often need to adapt according to changes in location, connectivity, bandwidth, and battery power. Concerted R&D efforts are therefore essential to devise new middleware solutions and capabilities that can fulfill the requirements of these emerging network technologies and next-generation applications.

There are significant limitations today with regard to building the types of large-scale complex distributed systems outlined above that have increasingly more stringent

requirements and more volatile environments. We are also discovering that more things need to be integrated over conditions that more closely resemble a dynamically changing Internet than they do a stable backplane. One problem is that the playing field is changing constantly, in terms of both resources and expectations. We no longer have the luxury of being able to design systems to perform highly specific functions and then expect them to have life cycles of 20 years with minimal change. In fact, we more routinely expect systems to behave differently under different conditions and complain when they just as routinely do not. These changes have raised a number of issues, such as end-to-end-oriented adaptive QoS, and construction of systems by composing off-the-shelf parts, many of which have promising solutions involving significant new middleware-based capabilities and services.

To address the many competing design forces and runtime QoS demands, a comprehensive methodology and environment is required to dependably compose large, complex, interoperable distributed systems from reusable components. Moreover, the components themselves must be sensitive to the environments in which they are packaged. Ultimately, what is desired is to take components that are built independently by different organizations at different times and assemble them to create a complete system. In the longer run, this complete system becomes a component embedded in still larger systems of systems. Given the complexity of this undertaking, various tools and techniques are needed to configure and reconfigure these systems so they can adapt to a wider variety of situations.

An essential part of what is needed to build the type of systems outlined above is the integration and extension of ideas that have been found traditionally in network management, data management, distributed operating systems, and object-oriented programming languages. But the goal for next-generation middleware is not simply to build a better network or better security in isolation, but rather to pull these capabilities together and deliver them to applications in ways that enable them to realize this model of adaptive behavior with tradeoffs between the various QoS attributes. The payoff will be reusable middleware that significantly simplifies the building of applications for systems of systems environments. The remainder of this section describes points of emphasis that are embedded within that challenge to achieve the desired payoff:

Reducing the Cost and Increasing the Interoperability of Using Heterogeneous Environments

Today, it is still the case that it costs quite a bit more in complexity and effort to operate in a truly heterogeneous environment, although nowhere near what it used to cost. Although it is now relatively easy to pull together distributed systems in heterogeneous environments, there remain substantial recurring downstream costs, particularly for complex and long-lived distributed systems of systems. Although homogeneous environments are simpler to develop and operate, they often do not reflect the long-run market reality, and they tend to leave open more avenues for catastrophic failure. We must, therefore,

remove the remaining impediments associated with integrating and interoperating among systems composed from heterogeneous components. Much progress has been made in this area, although at the host infrastructure middleware level more needs to be done to shield developers and end users from the accidental complexities of heterogeneous platforms and environments. In addition, interoperability concerns have largely focused on data interoperability and invocation interoperability. Little work has focused on mechanisms for controlling the overall behavior of integrated systems, which is needed to provide “control interoperability.” There are requirements for interoperable distributed control capabilities, perhaps initially as increased flexibility in externally controlling individual resources, after which approaches can be developed to aggregate these into acceptable global behavior.

Dynamic and Adaptive QoS Management

It is important to avoid “all or nothing” point solutions. Systems today often work well as long as they receive all the resources for which they were designed in a timely fashion, but fail completely under the slightest anomaly. There is little flexibility in their behavior (i.e., most of the adaptation is pushed to end users or administrators). Instead of hard failure or indefinite waiting, what is required is either *reconfiguration* to reacquire the needed resources automatically or *graceful degradation* if they are not available. Reconfiguration and operating under less than optimal conditions both have two points of focus: individual and aggregate behavior. To manage the increasingly stringent QoS demands of next-generation applications operating under changing conditions, middleware is becoming more adaptive and reflective. *Adaptive middleware* (25) is software whose functional and QoS-related properties can be modified either (1) *statically* (e.g., to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies) or (2) *dynamically* (e.g., to optimize system responses to changing environments or requirements, such as changing component interconnections, power levels, CPU/network bandwidth, latency/jitter, and dependability needs).

In mission-critical distributed systems, adaptive middleware must make such modifications dependably (i.e., while meeting stringent end-to-end QoS requirements). Reflective middleware (26) techniques make the internal organization of systems, as well as the mechanisms used in their construction, both visible and manipulable for middleware and application programs to inspect and modify at run time. Thus, reflective middleware supports more advanced adaptive behavior and more dynamic strategies keyed to current circumstances (i.e., necessary adaptations can be performed autonomously based on conditions within the system, in the system’s environment, or in system QoS policies defined by end users).

Advanced System Engineering Tools

Advanced middleware by itself will not deliver the capabilities envisioned for next-generation distributed systems. We must also advance the state of the system engineering

tools that come with these advanced environments used to build and evaluate large-scale mission-critical distributed systems. This area of research specifically addresses the immediate need for system engineering tools to augment advanced middleware solutions. A sample of such tools might include:

- *Design time tools*, to assist system developers in understanding their designs, in an effort to avoid costly changes after systems are already in place (which is partially obviated by the late binding for some QoS decisions referenced earlier).
- *Interactive tuning tools*, to overcome the challenges associated with the need for individual pieces of the system to work together in a seamless manner.
- *Composability tools*, to analyze resulting QoS from combining two or more individual components.
- *Modeling tools for developing system models* as adjunct means (both online and offline) to monitor and understand resource management, in order to reduce the costs associated with trial and error.
- *Debugging tools*, to address inevitable problems that develop at run time.

Reliability, Trust, Validation, and Assurance

The dynamically changing behaviors we envision for next-generation middleware-mediated systems of systems are quite different from what we currently build, use, and have gained some degrees of confidence in. Considerable effort must, therefore, be focused on validating the correct functioning of the adaptive behavior and on understanding the properties of large-scale systems that try to change their behavior according to their own assessment of current conditions before they can be deployed. But even before that, long-standing issues of adequate reliability and trust factored into our methodologies and designs using off-the-shelf components have not reached full maturity and common usage, and must therefore continue to improve. The current strategies organized around anticipation of long lifecycles with minimal change and exhaustive test case analysis are clearly inadequate for next-generation dynamic distributed systems of systems with stringent QoS requirements.

TAKING STOCK OF TECHNICAL PROGRESS ON MIDDLEWARE FOR DISTRIBUTED SYSTEMS

The increased maturation of, and reliance on, middleware for distributed systems stems from two fundamental trends that influence the way we conceive and construct new computing and information systems. The first is that IT of all forms is becoming highly commoditized (i.e., hardware and software artifacts are getting faster, cheaper, and better at a relatively predictable rate). The second is the growing acceptance of a network-centric paradigm, where distributed systems with a range of QoS needs are constructed by integrating separate components connected by various forms of reusable communication services. The nature of the interconnection ranges from the very small

and tightly coupled, such as embedded avionics mission computing systems, to the very large and loosely coupled, such as global telecommunications systems.

The interplay of these two trends has yielded new software architectural concepts and services embodied by middleware. The success of middleware has added new layers of infrastructure software to the familiar OS, programming language, networking, and database offerings of the previous generation. These layers are interposed between applications and commonly available hardware and software infrastructure to make it feasible, easier, and more cost effective to develop and evolve systems via reusable software. The past decade has yielded significant progress in middleware, which has stemmed, in large part, from the following:

Years of Iteration, Refinement, and Successful Use. The use of middleware is not new (27,28). Middleware concepts emerged alongside experimentation with the early Internet (and even its predecessor the ARPAnet), and middleware systems have been continuously operational since the mid-1980s. Over that period of time, the ideas, designs, and (most importantly) the software that incarnates those ideas have had a chance to be tried and refined (for those that worked), and discarded or redirected (for those that did not). This iterative technology development process takes a good deal of time to get right and be accepted by user communities and a good deal of patience to stay the course. When this process is successful, it often results in *standards* that codify the boundaries, and *patterns and frameworks* that reify the knowledge of how to apply these technologies, as described in the following subsections.

The Maturation of Open Standards and Open Source. Over the past decade, middleware standards have been established and have matured considerably, particularly with respect to mission-critical distributed systems that possess stringent QoS requirements. For instance, the OMG has adopted the following specifications in recent years: (1) *Minimum CORBA* (29), which removes nonessential features from the full OMG CORBA specification to reduce footprint so that CORBA can be used in memory-constrained embedded systems; (2) *Real-time CORBA* (18), which includes features that enable applications to reserve and manage network, CPU, and memory resources more predictably end-to-end; (3) *CORBA Messaging* (30), which exports additional QoS policies, such as timeouts, request priorities, and queuing disciplines, to applications; and (4) *Fault-tolerant CORBA* (23), which uses entity redundancy of objects to support replication, fault detection, and failure recovery. Robust implementations of these CORBA capabilities and services are now available from multiple suppliers, many of whom have adopted open-source business models. Moreover, the scope of open systems is extending to an even wider range of applications with the advent of emerging standards, such as the Real-Time Specification for Java (31), and the Distributed Real-Time Specification for Java (32).

The Dissemination of Patterns and Frameworks. Also during the past decade, a substantial amount of R&D effort has

focused on developing *patterns* and *frameworks* as a means to promote the transition and reuse of successful middleware technology. Patterns capture successful solutions to commonly occurring software problems that occur in a particular context (2,3). Patterns can simplify the design, construction, and performance tuning of middleware and applications by codifying the accumulated expertise of developers who have confronted similar problems before. Patterns also raise the level of discourse in describing software design and programming activities. Frameworks are concrete realizations of groups of related patterns (1). Well-designed frameworks reify patterns in terms of functionality provided by the middleware itself, as well as functionality provided by an application. A framework also integrates various approaches to problems where there are no a priori, context-independent, optimal solutions. Middleware frameworks (14) can include strategized selection and optimization patterns so that multiple independently developed capabilities can be integrated and configured automatically to meet the functional and QoS requirements of particular applications.

In the brief space of this article, we can only summarize and lend perspective to the many activities, past and present, that contribute to making middleware technology an area of exciting current development, along with considerable opportunity and unsolved challenging R&D problems. We have provided references to other sources to obtain additional information about ongoing activities in this area. We have also provided a more detailed discussion and organization for a collection of activities that we believe represent the most promising future directions for middleware. The ultimate goals of these activities are to:

1. Reliably and repeatably construct and compose distributed systems that can meet and adapt to more diverse, changing requirements/environments, and
2. Enable the affordable construction and composition of the large numbers of these systems that society will demand, each precisely tailored to specific domains.

To accomplish these goals, we must overcome not only the technical challenges, but also the educational and transitional challenges, and eventually master and simplify the immense complexity associated with these environments, as we integrate an ever-growing number of hardware and software components together via advanced middleware.

FURTHER READING

Quality Objects Toolkit v3.0 User's Guide, chapter 9. Available: <http://www.dist-systems.bbn.com/tech/QuO/release/latest/docs/usr/doc/quo-3.0/html/QuO30UsersGuide.htm>.

Sun Microsystems, "Jini Connection Technology", 1999. Available: <http://www.sun.com/jini/index.html>.

B. Sabata, S. Chatterjee, M. Davis, J. Sydir, T. Lawrence, Taxonomy for QoS specifications, *Proceedings of Workshop on Object-oriented Real-time Dependable Systems (WORDS 97)*, February 1997.

BIBLIOGRAPHY

1. R. Johnson, Frameworks = Patterns + Components, *CACM*, **40**(10), 1997.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.
3. D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, New York: Wiley, 2000.
4. C. Szyperski, *Component Software- Beyond Object-Oriented Programming*, Reading, M.A.: Addison-Wesley, 1998.
5. I. Jacobson, G. Booch, J. Rumbaugh, *Unified Software Development Process*, Reading, M.A.: Addison-Wesley, 1999.
6. Object Management Group, The Common Object Request Broker: Architecture and Specification Revision 3.0.2, OMG Technical Document, 2002.
7. R. Schantz, R. Thomas, and G. Bono, The architecture of the cronus distributed operating system, *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems*, Cambridge, M.A., 1986.
8. R. Gurwitz, M. Dean and R. Schantz, Programming support in the cronus distributed operating system, *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems*, Cambridge, MA, 1986.
9. D. Schmidt, D. Levine, and S. Mungee, The Design and Performance of the TAO Real-Time Object Request Broker, *Computer Communications Special Issue on Building Quality of Service into Distributed Systems*, **21** (4), 1998.
10. A. Wollrath, R. Riggs, J. Waldo, A distributed object model for the java system, *USENIX Computing Systems*, **9** (4), 1996.
11. G. Heineman and B. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Reading, MA: Addison-Wesley, 2001.
12. I. Foster, and K. Kesselman, *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann, 1999.
13. D. Schmidt, S. Huston, *C++ Network Programming Volume 1: Mastering Complexity with ACE and Patterns*, Reading, M.A.: Addison-Wesley, 2002.
14. D. Schmidt, S. Huston, *C++ Network Programming Volume 2: Systematic Reuse with ACE and Frameworks*, Reading, M.A.: Addison-Wesley, 2003.
15. T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Reading, MA: Addison-Wesley, 1997.
16. T. Thai and H. Lam, *.NET Framework Essentials*, Cambridge, M.A.: O'Reilly, 2001.
17. Object Management Group, CORBA Components, OMG Document formal/2002-06-65.
18. Object Management Group, Real-Time CORBA, OMG Document formal/02-08-02, 2002.
19. D. Box, *Essential COM*, Reading, MA: Addison-Wesley, 1997.
20. J. Snell, K. MacLeod, *Programming Web Applications with SOAP*, Cambridge, M.A.: O'Reilly, 2001.
21. Object Management Group, CORBAServices: Common Object Service Specification, OMG Document formal/98-12-31, edition, 1998.
22. A. Thomas, Enterprise Java Beans Technology, 1998. Available: http://java.sun.com/products/ejb/white_paper.html.
23. Object Management Group, *Fault Tolerance CORBA Using Entity Redundancy RFP*, OMG Document orbos/98-04-01 edition, 1998.

24. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, B. Bakken, M. Berman, D. Karr, R. Schantz, AQUA: An adaptive architecture that provides dependable distributed objects, *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 245–253.
25. J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications. *Proceedings of the 21st IEEE International Conference on Distributed Computing System*, Phoenix, AZ, 2001.
26. G.S. Blair, F. Costa, G. Coulson, H. Duran, et al., The design of a resource-aware reflective middleware architecture, *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection*, St.-Malo, France, Springer-Verlag, LNCS, Vol. **1616**, 1999.
27. R. Schantz, *BBN and the Defense Advanced Research Projects Agency*, Prepared as a Case Study for America's Basic Research: Prosperity Through Discovery, A Policy Statement by the Research and Policy Committee of the Committee for Economic Development (CED), June 1998, Available: <http://www.dist-systems.bbn.com/papers/1998/CaseStudy>.
28. P. Bernstein, Middleware, A model for distributed system service, *CACM*, **39**: 2, 1996.
29. Object Management Group, Minimum CORBA, OMG Document formal/00-10-59, 2000.
30. Object Management Group, CORBA Messaging Specification, OMG Document orbos/98-05-05, 1998.
31. G. Bollella and J. Gosling, The real-time specification for java, *Computer*, June 2000.
32. D. Jensen, Distributed Real-Time Specification for Java, 2000 Available: java.sun.com/aboutJava/communityprocess/jsr/jsr_050_drt.html.

RICHARD E. SCHANTZ
BBN Technologies
Cambridge, MA
DOUGLAS C. SCHMIDT
Electrical Engineering &
Computer Science Dept.
Vanderbilt University
Nashville, TN