Douglas C. Schmidt

Middleware for Real-Time and Embedded Systems

Off-the-shelf middleware technology is being adapted for such mission-critical dynamic domains as online financial trading, distributed process control, even submarine information systems, avionics mission computing, radar processing, and software-defined radios.

Distributed real-time and embedded (DRE) systems play an increasingly important role in modern application domains. The many types of DRE systems all have one thing in common: the right answer delivered too late becomes the wrong answer. Providing the right answer at the right time is crucial for, say, life-critical military DRE systems, such as those defending ships against missile attacks or controlling unmanned combat air vehicles through wireless links. The right answer is also crucial for safety-critical civilian DRE systems, such as those regulating the temperature of coolant in nuclear reactors and maintaining the safe operation of steel manufacturing machinery.

The affordability of certain types of distributed systems, such as two- and three-tier business systems, can often be enhanced through commercial-off-the-shelf (COTS) technologies. However, today's efforts aimed at integrating COTS into mission-critical DRE systems focus mainly on initial nonrecurring acquisition costs and do not reduce recurring software life-cycle costs. Likewise, many COTS products lack support for controlling key quality of service (QoS) properties, including predictable latency, jitter, and throughput, as well as scalability, dependability, and security. The inability to control them with sufficient confidence compromises DRE system adaptability and assurability; minor perturbations in conventional COTS products can cause failures leading to loss of life and property.

Historically, conventional COTS software has been unsuitable for mission-critical DRE systems due to its being characterized by either of two measures: flexible and standard but incapable of guaranteeing stringent QoS demands, thus limiting system assurability; or partially QoS-enabled but inflexible and nonstandard, thus limiting system adaptability and affordability. One result is that the rapid progress developing COTS software for mainstream business systems is not as broadly applicable for mission-critical DRE systems. Until this limitation is resolved, DRE system integrators and end users will be unable to take advantage of advances in COTS software in a dependable, timely, and cost-effective manner.

Here, I describe key R&D efforts helping create the new generation of assurable, adaptable, affordable COTS technologies to meet the stringent demands of mission-critical DRE systems. Although COTS software in DRE systems has been limited in scope and

TERRY MIURA

domain, future prospects are much brighter as a result of this work.

Technical Challenges and Solution Approaches

Some of the most challenging requirements for new and planned DRE systems can be characterized as follows:

- Multiple QoS properties must be satisfied in real time;
- Different levels of service are appropriate under different configurations, environmental conditions, and costs;
- The levels of service in one dimension must be coordinated with and/or traded-off against the levels of service in other dimensions to meet mission needs; and
- The need for autonomous and time-critical application behavior necessitates a flexible distributed system substrate that adapts robustly to dynamic changes in mission requirements and environmental conditions.

Although conventional COTS software cannot meet all these requirements, today's economic and organizational constraints—along with increasingly complex requirements and competitive pressure—prevent developers from building complex DRE system software entirely from scratch. Thus, programmers have a pressing need to develop, validate, and ultimately standardize a new generation of adaptive and reflective middleware technologies supporting stringent DRE system functionality and QoS requirements [1].

Middleware is reusable system software that functionally bridges the gap between the end-to-end functional requirements and mission doctrine of applications and the lower-level underlying operating systems and network protocol stacks [7]. Middleware therefore provides capabilities whose quality and QoS are critical to DRE systems.

The functional and QoS-related properties of adaptive middleware software [5] can be modified in two ways:

Statically. The capabilities of specific platforms are leveraged to enable functional subsetting and minimize hardware and software infrastructure dependencies, thus reducing memory footprint; or

Dynamically. System responses are optimized to changing environments or requirements, including component interconnections, power levels, CPU/network bandwidth, latency/jitter, and dependability needs.

In mission-critical DRE systems, adaptive middleware must make these modifications dependably while meeting stringent end-to-end QoS requirements.

Since they embody knowledge of a domain, domain-specific middleware services have potential for increasing the quality and decreasing the cycle time and effort to develop classes of DRE systems.

Reflective middleware goes a step further, permitting the automated examination of the middleware's capabilities and automated adjustment to optimize them [1]. Reflective middleware therefore supports more advanced adaptations that can be performed autonomously based on conditions within the DRE system, in the system's environment, or in the system's policies, as defined by operators and administrators.

Middleware Layers and R&D Efforts

Just as networking protocol stacks can be decomposed into multiple layers, middleware can be decomposed into multiple layers (see Figure 1). Each such layer is described in the following sections, along with related R&D efforts toward helping the middleware meet the stringent QoS demands of DRE systems.

Host infrastructure middleware. Host infrastructure middleware encapsulates and enhances native operating system communication and concurrency mechanisms to create portable and reusable network programming components, including reactors, acceptor-connectors, monitor objects, active objects, and component configurators [10]. These components abstract away the accidental incompatibilities of individual operating systems, helping eliminate many tedious, error-prone, and nonportable aspects of networked applications; they do so via low-level operating system programming APIs, such as Sockets and POSIX Pthreads.

An example of host infrastructure middleware R&D relevant for DRE systems is the Open Virtual Machine (OVM) project being conducted at Purdue University, the University of Maryland, and the State University of New York, Oswego, as part of the Defense Advanced Research Projects Agency Information Technology Office's Program Composition for Embedded Systems (PCES) program (see www.ovmj.org). OVM is an open-source real-time Java virtual machine that implements the Real-Time Specification for Java (RTSJ) [2]. The RTSJ is a set of extensions to Java providing a largely platform-independent way to execute code by encapsulating the differences between real-time operating systems and CPU architectures. Key RTSJ features include scoped and immortal memory, real-time threads with enhanced scheduling support, asynchronous event handlers, and asynchronous transfer of control within a thread.

The OVM virtual machine is written entirely in Java, and its architecture emphasizes customizability and pluggable components. Its implementation strives to balance performance and flexibility, allowing users to customize implementation of such operations as message dispatch, synchronization, field access, and speed. OVM allows dynamic update of the implementation of instructions of a running virtual machine. Although RTSJ virtual machines like OVM and TimeSys Jtime are relatively new, they have generated tremendous interest from the R&D and DRE systems-integrator communities due to their potential for reducing the costs of software development and evolution.

Distribution middleware. Distribution middleware defines higher-level distributed programming models whose reusable APIs and mechanisms automate and extend the native operating system network programming capabilities encapsulated by host infrastructure middleware. Distribution middleware enables developers to program distributed applications much like standalone applications, that is, by invoking operations on target objects without hard-coding dependencies on their location, programming language, operating system platform, communication protocols and interconnects, or hardware characteristics. At the heart of distribution middleware are QoS-enabled object request brokers (ORBs), such as CORBA, COM+, and Java Remote Method Invocation. These ORBs allow objects to interoperate across networks regardless of the language in which they are written or the operating system platform on which they are deployed.

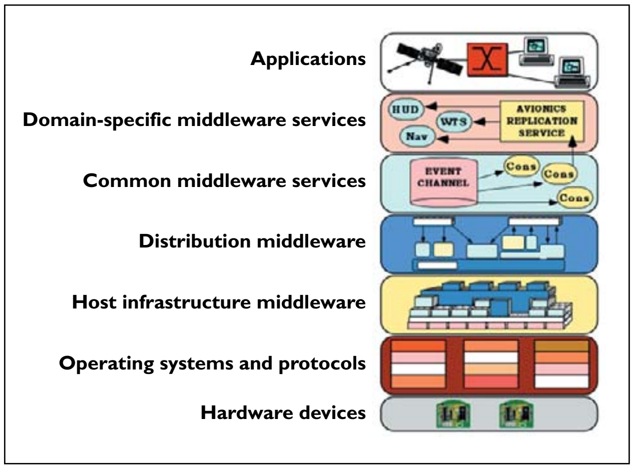
Applications
Domain-specific middleware services
HUD
WTS
Nav
AVIONICS REPLICATION SERVICE
Common middleware services
EVENT CHANNEL
Cons
Cons
Cons
Distribution middleware
Host infrastructure middleware
Operating systems and protocols
Hardware devices

Figure 1. Layers of middleware and their contexts.

An example of distribution middleware R&D relevant for DRE systems is the TAO project (see www.cs.wustl.edu/~schmidt/TAO.html) conducted by researchers at Washington University, St. Louis, and the University of California, Irvine, as part of the DARPA ITO Quorum program [9]. TAO is an open-source real-time CORBA ORB [6] that allows DRE applications to reserve and manage the following resources:

- Processor resources via thread pools, priority mechanisms, intra-process mutual exclusions (mutexes), and a global scheduling service for real-time systems with fixed priorities;
- Communication resources via protocol properties and explicit bindings to server objects using priority bands and private connections; and
- Memory resources via buffering requests in queues and bounding the size of thread pools.

TAO is implemented with reusable frameworks from the Adaptive Communication Environment (ACE) host infrastructure middleware toolkit (see www.cs.wustl.edu/~schmidt/ACE.html) [8]. ACE and TAO are mature examples of middleware transition from research prototype to DRE application, having been used in hundreds of DRE systems, including those in telecom network management and call processing, online trading services, avionics mission computing, software-defined radios, radar systems, surface mount “pick and place” systems, and hot-rolling steel mills.

Common middleware services. Common middleware services augment distribution middleware by defining higher-level domain-independent components that allow application developers to concentrate on programming application logic, without having to write the “plumbing” code needed to develop distributed applications, instead using lower-level middleware features directly. Whereas distribution middleware focuses largely on managing end-system resources in support of an object-oriented distributed programming model, common middleware services focus on allocating, scheduling, and coordinating various end-to-end resources throughout distributed systems via a component programming and scripting model. Developers reuse these services to manage global resources and perform recurring distribution tasks, including event notification, logging, persistence, real-time scheduling, fault tolerance, and transactions that would otherwise be implemented in an ad hoc manner by each application or integrator developer.

An example of common middleware services that are R&D relevant for DRE systems is the QuO project (see www.dist-systems.bbn.com/tech/QuO) by researchers at BBN Technologies as part of the DARPA ITO Quorum and PCES programs [5]. QuO is a set of open-source middleware services based on the layered middleware architecture outlined in Figure 2. The QuO architecture decouples DRE middleware and applications along two dimensions:

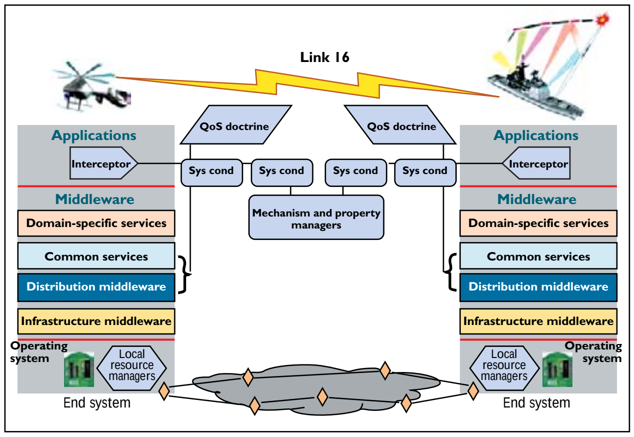

Figure 2. The QuO architecture.

Functional paths. Functional paths are flows of information between client and remote server applications. In distributed systems, middleware ensures this information is exchanged efficiently, predictably, scalably, dependably, and securely between remote peers. The information itself is largely application-specific, as determined by the functionality being provided (hence the term functional path).

QoS paths. QoS paths are responsible for determining how well the functional interactions behave end-to-end with respect to key DRE system QoS properties. Examples of such properties include: how and when resources are committed to client/server interactions at multiple levels of the system; the proper application and system behavior if available resources do not satisfy the expected resources; and the failure-detection and-recovery strategies necessary to meet end-to-end dependability requirements.

The QuO middleware is responsible for collecting, organizing, and disseminating QoS-related meta-information for monitoring and managing how well the functional interactions occur at multiple levels of DRE systems. It also enables the adaptive and reflective decision-making needed to support nonfunctional QoS properties robustly in the face of rapidly changing application requirements and environmental conditions, including local failures, transient overloads, and dynamic functional and QoS reconfigurations.

Domain-specific middleware services. Domain-specific middleware services are tailored to the requirements of particular DRE system domains, such as avionics mission computing, radar processing, online financial trading, and distributed process control. Unlike the previous three middleware layers, which provide broadly reusable "horizontal" mechanisms and services, domain-specific middleware services target vertical markets. From the COTS and R&D perspectives, domain-specific services are the least mature of the middleware layers, due in part to the historical lack of distribution middleware and middleware service standards needed for a stable base on which to create domain-specific middleware services. However, since they embody knowledge of a domain, domain-specific middleware services have the most potential for increasing the quality and decreasing the cycle time and effort integrators need to develop particular classes of DRE systems.

An example of domain-specific middleware services that are R&D relevant for DRE systems is the Boeing Bold Stroke architecture [11], which has been used as the open experimentation platform in many DARPA ITO programs. Bold Stroke is an open architecture for mission-computing avionics capabilities, such as navigation, heads-up display management, weapons targeting and release, and airframe sensor processing. The domain-specific middleware services in Bold Stroke are layered on COTS processors (PowerPC), network interconnects (VME), operating systems (VxWorks), infrastructure middleware (ACE), distribution middleware (TAO), and common middleware services (QuO and the CORBA Event Service).

Recent Progress, Future Needs

Significant progress has been made over the past five years in DRE middleware R&D and deployment, stemming largely from the following advances:

Years of research, iteration, refinement, use. Middleware and the related Distributed Object Computing (DOC) middleware are not new [7]. Middleware concepts coevolved from experimentation with the early Internet (even from its predecessor, the ARPAnet); DOC middleware systems have been operational since the mid-1980s following the advent of BBN's Cronus and Corbus systems. Since then, the ideas, designs, and, most important, the software incarnating these ideas have been tried and refined (for those that worked) and discarded or redirected (for those that didn't). This iterative development process takes a good deal of funding and time to get right and be accepted by user communities, as well as a good deal of patience to stay the course. When the process succeeds, it often results in standards (codifying the boundaries) and patterns and frameworks (reifying the knowledge of how to apply the technologies):

PRIOR TO OPEN MIDDLEWARE PLATFORMS, SUCH R&D RESULTS WOULD HAVE BEEN BURIED IN CUSTOM OR PROPRIETARY SYSTEMS, RATHER THAN SERVING AS THE BASIS FOR RESHAPING THE R&D AND INTEGRATOR COMMUNITIES.

Maturation of standards. Over the past decade, middleware standards have matured considerably with respect to DRE requirements. For instance, the OMG has adopted the following specifications:

Minimum CORBA. Removes nonessential features from the full OMG CORBA specification to reduce memory footprint, rendering CORBA usable in memory-constrained embedded systems;
Real-time CORBA. Includes features allowing applications to reserve and manage network, CPU, and memory resources predictably end-to-end; and
CORBA messaging. Exports to applications additional QoS policies, such as timeouts, request priorities, and queueing disciplines.
Fault-tolerant CORBA. Uses objects' entity redundancy to support replication, fault detection, and failure recovery.

Multiple interoperable and robust implementations of these CORBA capabilities and services are available today. Moreover, emerging standards like Dynamic Scheduling Real-Time CORBA, the Real-Time Specification for Java, and the Distributed Real-Time Specification for Java are extending the scope of open standards to a wider range of DRE applications.

Patterns and frameworks. A substantial amount of R&D effort has focused on the following means of promoting the development and reuse of high-quality middleware technology:

Patterns. Patterns codify design expertise, thus providing time-proven solutions to commonly occurring software problems in certain contexts [3, 10]. Patterns simplify the design, construction, and performance-tuning of DRE applications by codifying the accumulated expertise of developers who have successfully confronted similar problems before. Patterns also elevate the level of discourse in describing software development activities to focus on strategic architecture and design issues, rather than just tactical programming and representation details.
Frameworks. Frameworks are concrete realizations of groups of related patterns [4]. Well-designed frameworks reify patterns in terms of functionality provided by the middleware itself, as well as functionality provided by applications. Frameworks also integrate various approaches to problems where there are no a priori context-independent, optimal solutions. Middleware frameworks, such as ACE, OVM, QuO, and TAO, can include strategized selection and optimization patterns; multiple independently developed capabilities can then be integrated and configured automatically to meet the functional and QoS requirements of individual DRE applications.

Sustained government R&D investment. Much of the pioneering R&D effort on middleware patterns and frameworks has been conducted in the two DARPA programs mentioned earlier—Quorum and PCES. Each focuses on CORBA and Java open-systems middleware, yielding many results that have transitioned from research prototypes into standardized service definitions and implementations for the real-time and fault-tolerant CORBA specification and commercialization efforts. Quorum and PCES are examples of how focused government R&D efforts leverage their results by exporting them into and combining them with other ongoing public and private activities that also use a common open middleware substrate. Prior to the viability of standards-based open middleware platforms, such R&D results would have been buried in custom or proprietary systems, serving only as an existence proof, rather than as the basis for fundamentally reshaping the R&D and integrator communities.

Standards-based middleware has also now been demonstrated and is deployed today in a number of mission-critical DRE systems, including avionics mission computing, software-defined radios, and submarine information systems. However, since COTS middleware technology is not yet mature enough to cover large-scale, dynamically changing systems, these middleware applications have been confined to relatively small-scale statically configured DRE systems.

To satisfy the highly application- and mission-specific QoS requirements in network-centric DRE "system of systems" environments, considerable additional R&D is still required to enhance middleware, particularly when it involves common and domain-specific services. If these efforts succeed, future middleware technologies will be able to control individual and aggregate resources used by multiple system components at multiple system levels to dependably manage communication bandwidth, security, and the scheduling and allocation of DRE system artifacts.

Middleware is a strategic element in developing DRE systems, bridging the gap between application programs and the underlying operating systems and network protocol stacks to provide reusable services critical to these systems. The economic payoffs of middleware R&D efforts stem from moving standardization up several levels of abstraction by DRE software technology artifacts, such as middleware frameworks, protocols, service components, and patterns, so they are available for COTS acquisition and customization. Given the proper advanced R&D context and an effective process for commercializing R&D results, the COTS middleware market will inevitably adapt, adopt, and implement the robust hardware and software capabilities needed for mission-critical DRE systems.

As a result of the R&D efforts discussed here, as well as other similar efforts, the next generation of middleware will be able to adapt to dynamically changing conditions in order to use the available computer and network infrastructure to the greatest degree possible in support of application needs. C

References

1. Blair, G. et al. The design of a resource-aware reflective middleware architecture. In *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection* (July 19–21, Saint-Malo, France). Springer-Verlag, 1999.
2. Bollella, G. and Gosling, J. The real-time specification for Java. *Comput. 33,* 6 (June 2000).
3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
4. Johnson, R. Frameworks = patterns + components. *Commun. ACM 40,* 10 (Oct. 1997).
5. Loyall, J., Gossett, J., Gill, C., Schantz, R., Zinky, J., Pal, P., Shapiro, R., Rodrigues, C., Atighetchi, M., and Karr, D. Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems* (Phoenix, AZ, Apr. 16–19, 2001).
6. Object Management Group. *The Common Object Request Broker: Architecture and Specification, rev. 2.6.* OMG technical document, Dec. 2001; see www.omg.org.
7. Schantz, R. and Schmidt, D. Middleware for distributed systems: Evolving the common structure for network-centric applications. In *Encyclopedia of Software Engineering,* J. Marciniak, and G. Telecki, Eds. John Wiley & Sons, Inc., New York, 2001.
8. Schmidt, D. and Huston, S. *C++ Network Programming: Mastering Complexity with ACE and Patterns.* Addison-Wesley, Reading, MA, 2002.
9. Schmidt, D., Levine, D., and Mungee, S. The design and performance of the TAO real-time object request broker. In a special issue on building quality of service into distributed systems. *Comput. Commun. 21,* 4 (1998).
10. Schmidt, D., Stal, M., Rohnert H., and Buschmann, F. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Inc., New York, 2000.
11. Sharp, D. Reducing avionics software cost through component-based product line development. In *Proceedings of the Software Technology Conference* (Salt Lake City, UT, Apr. 1998).

Douglas C. Schmidt (schmidt@uci.edu) is an associate professor in the Department of Electrical and Computer Engineering of the University of California, Irvine; for more on these DRE middleware R&D efforts, see www.cs.wustl.edu/~schmidt.