Middleware for Distributed Systems

Evolving the Common Structure for Network-centric Applications

Richard E. Schantz
BBN Technologies
10 Moulton Street
Cambridge, MA 02138, USA
schantz@bbn.com

Douglas C. Schmidt
Electrical & Computer Engineering Dept.
University of California, Irvine
Irvine, CA 92697-2625, USA
schmidt@uci.edu

1 Overview of Trends, Challenges, and Opportunities

Two fundamental trends influence the way we conceive and construct new computing and information systems. The first is that information technology 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 applications with a range of quality of service (QoS) needs are constructed by integrating separate components connected by various forms of communication services. The nature of this interconnection can range from

1. The very small and tightly coupled, such as avionics mission computing systems to
2. The very large and loosely coupled, such as global telecommunications systems.

The interplay of these two trends has yielded new architectural concepts and services embodying layers of *middleware*. 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 using reusable software. Middleware stems from recognizing the need for more advanced and capable support–beyond simple connectivity–to construct effective distributed systems. A significant portion of middleware-oriented R&D activities over the past decade have focused on

1. The identification, evolution, and expansion of our understanding of current middleware services in providing this style of development and
2. The need for defining additional middleware layers and capabilities to meet the challenges associated with constructing future network-centric systems.

These activities are expected to continue forward well into this decade to address the needs of next-generation distributed applications.

During the past decade we've also benefited from the commoditization of hardware (such as CPUs and storage devices) and networking elements (such as IP routers). More recently, the maturation of 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 .NET) are helping to commoditize many software components and architectural layers. The quality of commodity 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, recent improvements in frameworks [John97], patterns [Gam95, Bus96, Sch00b], and development processes [Beck00, RUP99] have encapsulated the knowledge that enables common off-the-shelf (COTS) software to be developed, combined, and used in an increasing number of real-world applications, such as e-commerce web sites, consumer electronics, avionics mission computing, hot rolling mills, command and control planning systems, backbone routers, and high-speed network switches.

The trends outlined above are now yielding additional middleware challenges and opportunities for organizations and developers, both in deploying current middleware-based solutions and in inventing and shaping new ones. To complete our overview, we summarize key challenges and emerging opportunities for moving forward, and outline the role that middleware plays in meeting these challenges.

- ***Growing focus on integration rather than on programming*** – There is an ongoing trend away from programming applications from scratch to integrating them by configuring and customizing reusable components and frameworks [John97]. While it is possible in theory to program applications from scratch, economic and organizational constraints–as well as increasingly complex requirements and competitive pressures–are making it infeasible to do so in practice. Many applications in the future will therefore be configured by integrating reusable commodity hardware and software

components that are implemented by different suppliers together with the common middleware substrate needed to make it all work harmoniously.

- ***Demand for end-to-end QoS support, not just component QoS*** – The need for autonomous and time-critical behavior in next-generation applications necessitates more flexible system infrastructure components that can adapt robustly to dynamic end-to-end changes in application requirements and environmental conditions. For example, next-generation applications will require the simultaneous satisfaction of multiple QoS properties, such as predictable latency/jitter/throughput, scalability, dependability, and security. Applications will also need different levels of QoS under different configurations, environmental conditions, and costs, and multiple QoS properties must be coordinated with and/or traded off against each other to achieve the intended application results. Improvements in current middleware QoS and better control over underlying hardware and software components–as well as additional middleware services to coordinate these–will all be needed.

- ***The increased viability of open systems*** – Shrinking profit margins and increasing shareholder pressure to cut costs are making it harder for companies to invest in long-term research that does not yield short-term pay offs. As a result, many companies can no longer afford the luxury of internal organizations that produce completely custom hardware and software components with proprietary QoS support. To fill this void, therefore, standards-based hardware and software researched and developed by third parties–and glued together by common middleware–is becoming increasingly strategic to many industries. This trend also requires companies to transition away from proprietary architectures to more open systems in order to reap the benefits of externally developed components, while still maintaining an ability to compete with domain-specific solutions that can be differentiated and customized. The refactoring of much domain-independent middleware into open-source releases based on open standards is spurring the adoption of common software substrates in many industries. It is also emphasizing the role of domain knowledge in selecting, organizing, and optimizing appropriate middleware components for requirements in particular application domains.

- ***Increased leverage for disruptive technologies leading to increased global competition*** – One consequence of the commoditization of larger bundled solutions based around middleware-integrated components is that industries long protected by high barriers to entry, such as telecom and aerospace, are more vulnerable to disruptive technologies [Chris98] and global competition, which drive prices to marginal cost. For example, advances in high-performance COTS hardware are being combined with real-time and fault tolerant middleware services to simplify the development of predictable and dependable network elements. Systems incorporating these network elements, ranging from PBXs to high-speed backbone routers–and ultimately carrier class switches and services built around these components–can now use standard hardware and software components that are less expensive than legacy proprietary systems, yet which are becoming nearly as dependable.

- ***Potential complexity cap for next-generation systems*** – Although current middleware solves a number of basic problems with distribution and heterogeneity, many challenging research problems remain. In particular, problems of scale, diversity of operating environments, and required level of trust in the sustained and correctly functioning operation of next-generation systems have the potential to outstrip what can be built. Without significantly improved capabilities in a number of areas, we may reach a point where the limits of our starting points put a ceiling on the size and levels of complexity for future systems. Without an investment in fundamental R&D to invent, develop, and popularize the new middleware capabilities needed to realistically and cost-effectively construct next-generation network-centric applications, the anticipated move towards large-scale distributed "systems of systems" in many domains may not materialize. Even if it does, it may do so with intolerably high risk because of inadequate COTS middleware support for proven, repeatable, and reliable solutions. The additional complexity forced into the realm of application development will only exacerbate the already high rate of project failures exhibited in complex distributed system domains.

The preceding discussion outlines the fundamental drivers that led to the emergence of middleware architectures and components in the previous decade, and will of necessity lead to more advanced middleware capabilities in this decade. We spend the rest of this paper exploring these topics in more depth, with detailed evaluations of where we are, and where we need to go, with respect to middleware.

2 How Middleware Addresses Distributed Application Challenges

Requirements for faster development cycles, decreased effort, and greater software reuse motivate the creation and use of *middleware* and *middleware-based architectures*. Middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to

1. Functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate and
2. Enable and simplify the integration of components developed by multiple technology suppliers.

When implemented properly, middleware can help to:

- Shield software developers 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 requirements in order to simplify the development of distributed and embedded 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.

Over the past decade, various technologies have been devised to alleviate many complexities associated with developing software for distributed applications. Their successes have added a new category of systems software to the familiar operating system, programming language, networking, and database offerings of the previous generation. Some of the most successful of these technologies have centered on *distributed object computing (DOC) middleware*. DOC is an advanced, mature, and field-tested middleware paradigm that supports flexible and adaptive behavior. DOC middleware architectures are composed of relatively autonomous software objects that can be distributed or collocated throughout a wide range of networks and interconnects. Clients invoke operations on target objects to perform interactions and invoke functionality needed to achieve application goals. Through these interactions, a wide variety of middleware-based services are made available off-the-shelf to simplify application development. Aggregations of these simple, middleware-mediated interactions form the basis of large-scale distributed system deployments.

2.1 Structure and Functionality of DOC Middleware

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 DOC middleware be decomposed into multiple layers, such as those shown in Figure 1.

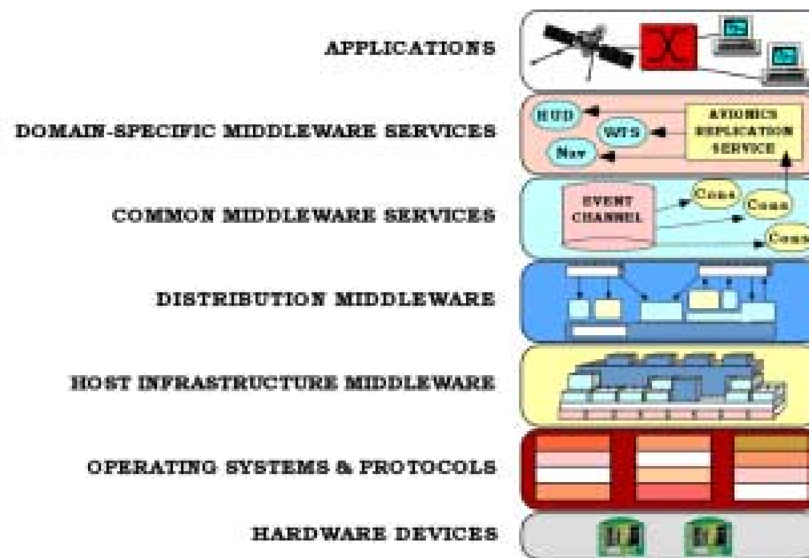

Figure 1. Layers of DOC Middleware and Surrounding Context

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

Host infrastructure middleware encapsulates and enhances native OS communication and concurrency mechanisms to create reusable network programming components, such as reactors, acceptor-connectors, monitor objects, active objects, and component configurators [Sch00b, Sch01]. These components abstract away the peculiarities of individual operating systems, and help eliminate many tedious, error-prone, and non-portable 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) [JVM97], 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.
- .NET [NET01] 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 upon which Microsoft's .NET services are built. The Microsoft 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) [Sch01] is a highly portable toolkit written in C++ that encapsulates native operating system (OS) network programming capabilities, such as connection establishment, event demultiplexing, interprocess communication, (de)marshaling, static and dynamic configuration of application components, 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.

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 applications 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) [Omg00], which is an open standard for distribution middleware that allows objects to interoperate across networks regardless of the language in which they were written or the platform on which they are deployed. In 1998 the OMG adopted the Real-time CORBA (RT-CORBA) specification [Sch00a], which 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) [Wol96], 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..
- Microsoft's Distributed Component Object Model (DCOM) [Box97], 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 operating systems, DCOM is implemented primarily on Windows platforms.
- SOAP [SOAP01] 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.

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 applications by using lower-level middleware directly. For example, application developers no longer need to write code that handles transactional behavior, security, database connection pooling or threading, because common middleware service providers bundle these tasks into reusable components. 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 resources 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:

- The OMG's CORBA Common Object Services (CORBAservices) [Omg98b], which provide domain-independent interfaces and capabilities that can be used by many DOC applications. 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 [Tho98], 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. Since EJB is built on top of Java technology, EJB service components can only be implemented using the Java language. The CORBA Component Model (CCM) [Omg99] defines a superset of EJB capabilities that can be implemented using all the programming languages supported by CORBA.
- Microsoft's .NET Web services [NET01], which complements the lower-level middleware .NET capabilities, 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 the Hypertext Transfer Protocol (HTTP) and eXtensible Markup Language (XML), respectively.

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 DOC 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. Since 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 healthcare, telecom, command and control, and process automation domains.
- The Siemens Medical Engineering Group has developed *syngo*® 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, flouroscopy, angiography, computer tomography, magnetic resonance, nuclear medicine, therapy systems, cardiac systems, patient monitoring systems, life support systems, and imaging- and diagnostic-workstations. The *syngo*® middleware services allow healthcare facilities to integrate diagnostic imaging and other radiological, cardiological and hospital services via a blackbox application template framework based on advanced patterns for communication, concurrency, and configuration for both business logic and presentation logic supporting a common look and feel throughout the medical domain.
- The Boeing Bold Stroke [Sha98, Doe99] architecture uses COTS hardware and middleware to produce a non-proprietary, standards-based component architecture for military avionics mission computing capabilities, such as navigation, display management, sensor management and situational awareness, data link management, and weapons control. A driving objective of Bold Stroke was to support reusable product line applications, leading to a highly configurable application component model and supporting middleware services. Associated products ranging from single processor systems with $O(10^5)$ lines of source code to multi-processor systems with $O(10^6)$ lines of code have shown dramatic affordability and schedule improvements and have been flight tested successfully. The domain-specific middleware services in Bold Stroke are layered upon common middleware services (the CORBA Event Service), distribution middleware (Real-time CORBA), and host infrastructure middleware (ACE), and have been demonstrated to be highly portable for different COTS operating systems (e.g. VxWorks), interconnects (e.g. VME), and processors (e.g. PowerPC).

2.2 Benefits of DOC Middleware

Middleware in general–and DOC middleware in particular–provides essential capabilities for developing distributed applications. In this section we summarize its improvements over traditional non-middleware oriented approaches, using the challenges and opportunities described in Section 1 as a guide:

- ***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 either
  1. Applications, which at best were customized for a single use,
  2. Networks, which were necessarily concerned with providing the communication layer, or
  3. 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 operating systems and programming model concepts into organizing and controlling the composition of separately developed components across host boundaries. Every DOC middleware technology has within it some type of request broker functionality that initiates and manages inter-component 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 technology used to develop them. These capabilities allow DOC middleware to amortize software life-cycle efforts by leveraging previous development expertise and reifying implementations of key patterns into more encompassing reusable frameworks and components. As DOC middleware continues to mature and incorporates additional needed services, next-generation applications will increasingly be assembled by modeling, integrating, and scripting domain-specific and common service components, rather than by
  1. Being programmed either entirely from scratch or
  2. Requiring significant customization or augmentation to off-the-shelf component implementations.

- ***Demand for end-to-end QoS support, not just component QoS*** – This area represents the next great wave of evolution for advanced DOC middleware. There is now widespread recognition that effective development of large-scale distributed applications requires the use of COTS infrastructure and service components. Moreover, the usability of the resulting products depends heavily on the properties of the whole as derived from its parts. This type of environment requires *visible*, *predictable*, *flexible*, and *integrated* resource management strategies within and between the pieces.

  Despite the ease of connectivity provided by middleware, however, constructing integrated systems remains hard since it requires significant customization of non-functional QoS properties, such as predictable latency/jitter/throughput, scalability, dependability, and security. In their most useful forms, these properties extend end-to-end and thus have elements applicable to
  - The network substrate
  - The platform operating systems and system services
  - The programming system in which they are developed
  - The applications themselves and
  - The middleware that integrates all these elements together.

  Two basic premises underlying the push towards end-to-end QoS support mediated by middleware are that:
  1. Different levels of service are possible and desirable under different conditions and costs and
  2. The level of service in one property must be coordinated with and/or traded off against the level of service in another to achieve the intended overall results.

  To manage the increasingly stringent QoS demands of next-generation applications, middleware is becoming more adaptive and reflective. *Adaptive middleware* [Loy01] is software whose functional and QoS-related properties can be modified either:
  - *Statically*, *e.g.,* to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies or
  - *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 systems, adaptive middleware must make such modifications dependably, *i.e.,* while meeting stringent end-to-end QoS requirements. *Reflective middleware* [Bla99] goes further to permit automated examination of the capabilities it offers, and to permit automated adjustment to optimize those capabilities. Reflective 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 runtime. 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.

- ***The increased viability of open systems*** architectures and open-source availability – By their very nature, 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. In turn, this yields the potential for multiple choices for component implementations, and open engineering concepts. Standards organizations such as the OMG and The Open Group 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. Since few companies today can afford significant investments in internally funded R&D, it is increasingly important for the information technology industry to leverage externally funded R&D sources, such as government investment. In this context, standards-based DOC 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 program [Dar99] focused heavily on CORBA open systems middleware. Quorum yielded many results that transitioned into standardized service definitions and implementations for Real-time [OMG00B, Sch98A] and Fault-tolerant [Omg98a, Cuk98] CORBA specification 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. Prior to 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.

- ***Increased leverage for disruptive technologies leading to increased global competition*** – Middleware supporting component integration and reuse is a key technology to help amortize software life-cycle costs by:
  1. Leveraging previous development expertise, *e.g.,* DOC middleware helps to abstract commonly reused low-level OS concurrency and networking details away into higher-level, more easily used artifacts and
  2. Focusing on efforts to improve software quality and performance, *e.g.,* DOC middleware combines various aspects of a larger solution together, *e.g.,* fault tolerance for domain-specific objects with real-time QoS properties.

  When developers needn't worry as much about low-level details, they are freed to focus on more strategic, larger scope, application-centric specializations concerns, such as distributed resource management and end-to-end dependability. 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. And that, 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 economy where developers can leverage:
  1. Frequently used common components, which come with steady innovation cycles resulting from a multi-user basis, in conjunction with
  2. Custom domain-specific capabilities, which allow appropriate mixing of multi-user low cost and custom development for competitive advantage.

- ***Potential complexity cap for next-generation complex systems*** – As today's technology transitions run their course, the systemic reduction in long-term R&D activities runs the risk of limiting the complexity of next-generation systems that can be developed and integrated using COTS hardware and software components. The advent of open DOC middleware standards, such as CORBA and Java-based technologies, is hastening industry consolidation towards portable and interoperable sets of COTS products that are readily available for purchase or open-source acquisition. These products are still deficient and/or immature, however, in their ability to handle some of the most important attributes needed to support future systems. Key attributes include end-to-end QoS, dynamic property tradeoffs, extreme scaling (large and small), highly mobile environments, and a variety of other inherent complexities. Complicating this situation over the past decade has been the steady flow of faculty, staff, and graduate students out of universities and research labs and into startup companies and other industrial positions. While this migration helped fuel the global economic boom in the late '90s, it does not bode well for long-term technology innovation.

  As distributed systems grow in complexity, it may not be possible to sustain the composition and integration perspective we have achieved with current middleware platforms without continued R&D. Even worse, we may plunge ahead with an inadequate knowledge base, reverting to a myriad of high-risk independent solutions to common problems. Ultimately, premium value and competitive advantage will accrue to those who master the patterns and pattern languages

[Sch00b] necessary to integrate COTS hardware and DOC middleware to develop low cost, complex distributed systems with superior domain-specific attributes that cannot be bought off-the-shelf at any given point in time.

3 The Road Ahead

The past decade has yielded significant progress in DOC middleware, which has stemmed in large part from the following trends:

- ***Years of iteration, refinement, and successful use*** – The use of middleware and DOC middleware is not new [Sch86, Sch98, Ber96]. Middleware concepts emerged alongside experimentation with the early Internet (and even its predecessor the ARPAnet), and DOC middleware systems have been continuously operational since the mid 1980's. 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 didn't). 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 bullets.

- ***The maturation of standards*** – Over the past decade, middleware standards have been established and have matured considerably, particularly with respect to distributed real-time and embedded (DRE) systems. For instance, the OMG has adopted the following specifications in the past three years:
  - *Minimum CORBA*, which removes non-essential features from the full OMG CORBA specification to reduce footprint so that CORBA can be used in memory-constrained embedded systems
  - *Real-time CORBA*, which includes features that allow applications to reserve and manage network, CPU, and memory resources predictably end-to-end
  - *CORBA Messaging*, which exports additional QoS policies, such as timeouts, request priorities, and queueing disciplines, to applications and
  - *Fault-tolerant CORBA*, 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 vendors. Moreover, the scope of open systems is extending to an even wider range of applications with the advent of emerging standards, such as Dynamic Scheduling Real-Time CORBA [Omg01], the Real-Time Specification for Java [Bol00], and the Distributed Real-Time Specification for Java..
- ***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 development and reuse of successful middleware technology. Patterns capture successful solutions to commonly occurring software problems that arise in a particular context [Gam95]. 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 [John97]. 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 [Sch01] 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.

The following section presents an analysis of the challenges and opportunities for next-generation middleware and outlines the research orientation required to overcome the challenges and realize the opportunities.

3.1 Challenges and Opportunities for Next-generation Middleware

An increasing number of next-generation applications will be developed as distributed "systems of systems," which include many interdependent levels, such as network/bus interconnects, local and remote endsystems, and multiple layers of common and domain-specific middleware. The desirable properties of these systems of systems include predictability, controllability, and adaptability of operating characteristics for applications with respect to such features as time, quantity of information, accuracy, confidence, and synchronization. All these issues become highly volatile in systems of systems, due to the dynamic interplay of the many interconnected parts. These parts are often constructed in a similar way from smaller parts.

To address the many competing design forces and runtime QoS demands, a comprehensive methodology and environment is required to dependably compose large, complex, interoperable DOC applications 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 hierarchically 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. The payoff will be reusable DOC middleware that significantly simplifies the building of applications for systems of systems environments. The following points of emphasis are embedded within that challenge to achieve the payoff:

- ***Toward more universal use of common middleware*** – Today, it is too often the case that a substantial percentage of the effort expended to develop applications goes into building *ad hoc* and proprietary middleware substitutes, or additions for missing middleware functionality. As a result, subsequent composition of these *ad hoc* capabilities is either infeasible or prohibitively expensive. One reason why redevelopment persists is that it is still often relatively easy to pull together a minimalist *ad hoc* solution, which remains largely invisible to all except the developers. Unfortunately, this approach can yield substantial recurring downstream costs, particularly for complex and long-lived distributed systems of systems.

  Part of the answer to these problems involves educating developers about software lifecycle issues beyond simply interconnecting individual components. In addition, there are many different operating environments, so providing a complete support base for each takes time, funding, and a substantial user community to focus the investment and effort needed. To avoid this often repeated and often wasted effort, more of these "completion" capabilities must be available off-the-shelf, through middleware, operating systems, and other common services. Ideally, these COTS capabilities can eliminate, or at least minimize, the necessity for significant custom augmentation to available packages just to get started. Moreover, we must 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.

- ***Common solutions handling both variability and control*** – 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. Moreover, there is a need for interoperability of control and management mechanisms.

  As with the adoption of middleware-based packages and shielding applications from heterogeneity, there has been much progress in the area of interoperability. Thus far, however, interoperability concerns have 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 control capabilities to appear in individual resources first, after which approaches can be developed to aggregate these into acceptable global behavior.

Before outlining the research that will enable these capabilities, it is important to understand the role and goals of middleware within an entire system. As illustrated in Section 2.1, middleware resides between applications and the underlying OS, networks, and computing hardware. As such, one of its most immediate goals is to augment those interfaces with QoS attributes. It is important to have a clear understanding of the QoS information so that it becomes possible to:

1. Identify the users' requirements at any particular point in time and
2. Understand whether or not these requirements are being (or even can be) met.

It is also essential to aggregate these requirements, making it possible to form decisions, policies, and mechanisms that begin to address a more global information management organization. Meeting these requirements will require flexibility on the parts of both the application components and the resource management strategies used across heterogeneous systems of systems. A key direction for addressing these needs is through the concepts associated with managing adaptive behavior, recognizing that not all requirements can be met all of the time, yet still ensuring predictable and controllable end-to-end behavior.

Within these general goals, there are pragmatic considerations, including incorporating the interfaces to various building blocks that are already in place for the networks, operating systems, security, and data management infrastructure, all of which continue to evolve independently. Ultimately, there are two different types of resources that must be considered:

1. Those that will be fabricated as part of application development and
2. Those that are provided and can be considered part of the substrate currently available.

While not much can be done in the short-term to change the direction of the hardware and software substrate that's installed today, a reasonable approach is to provide the needed services at higher levels of (middleware-based) abstraction. This architecture will enable new components to have properties that can be more easily included into the controllable applications and integrated with each other, leaving less lower-level complexity for application developers to address and thereby reducing system development and ownership costs. Consequently, the goal of 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. As the evolution of the underlying system components change to become more controllable, we can expect a refactoring of the implementations underlying the enforcement of adaptive control.

3.2 Research Orientation

The following two concepts are central to the type of QoS tradeoffs described in Section 3.1 above:

- *Contracts* – Information must be gathered for particular applications or application families regarding user requirements, resource requirements, and system conditions. Multiple system behaviors must be made available based on what is best under the various conditions. This information provides the basis for the contracts between users and the underlying system substrate. These contracts provide not only the means to specify the degree of assurance of a certain level of service, but also provide a well-defined middleware abstraction to improve the visibility of changes in the mandated behavior.
- *Graceful degradation* – Mechanisms must also be developed to monitor the system and enforce contracts, providing feedback loops so that application services can degrade gracefully (or augment) as conditions change, according to a prearranged contract governing that activity. The initial challenge here is to establish the idea in developers' and users' minds that multiple behaviors are both feasible and desirable. The next step is to put into place the additional middleware support–including connecting to lower level network and operating system enforcement mechanisms–necessary to provide the right behavior effectively and efficiently given current system conditions.

Although it is possible to satisfy contracts and achieve graceful degradation to a limited degree in a limited range of systems today [Kar01], much R&D work remains. The research orientation needed to deliver these goals can be divided into the seven areas described below:

1. ***Individual QoS Requirements*** – Individual QoS deals with developing the mechanisms relating to the end-to-end QoS needs from the perspective of a single user or application. The specification requirements include multiple contracts, negotiation, and domain specificity. Multiple contracts are needed to handle requirements that change over time and to associate several contracts with a single perspective, each governing a portion of an activity. Different users running the same application may have different QoS requirements emphasizing different benefits and tradeoffs, often depending on current configuration. Even the same user running the same application at different times may have different QoS requirements, *e.g.,* depending on current mode of operation and other external factors. Such dynamic behavior must be taken into account and introduced seamlessly into next-generation distributed systems.

   General negotiation capabilities that offer convenient mechanisms to enter into and control a negotiated behavior (as contrasted with the service being negotiated) need to be available as COTS middleware packages. The most effective way for such negotiation-based adaptation mechanisms to become an integral part of QoS is for them to be "user friendly," *e.g.,* requiring a user or administrator to simply provide a list of preferences. This is an area that is likely to become domain-specific and even user-specific. Other challenges that must be addressed as part of delivering QoS to individual applications include:

   - Translation of requests for service among and between the various entities on the distributed end-to-end path
   - Managing the definition and selection of appropriate application functionality and system resource tradeoffs within a "fuzzy" environment and
   - Maintaining the appropriate behavior under composability.

   Translation addresses the fact that complex systems of systems are being built in layers. At various levels in a layered architecture the user-oriented QoS must be translated into requests for other resources at a lower level. The challenge is how to accomplish this translation from user requirements to system services. A logical place to begin is at the application/middleware boundary, which closely relates to the problem of matching application resources to appropriate

distributed system resources. As system resources change in significant ways, either due to anomalies or load, tradeoffs between QoS attributes (such as timeliness, precision, and accuracy [TPA97]) may need to be (re)evaluated to ensure an effective level of QoS, given the circumstances. Mechanisms need to be developed to identify and perform these tradeoffs at the appropriate time. Last, but certainly not least, a theory of effectively composing systems from individual components in a way that maintains application-centric end-to-end properties needs to be developed, along with efficient implementable realizations of the theory.

2. ***Runtime Requirements*** – From a system lifecycle perspective, decisions for managing QoS are made at design time, at configuration/deployment time, and/or at runtime. Of these, the runtime requirements are the most challenging since they have the shortest time scales for decision-making, and collectively we have the least experience with developing appropriate solutions. They are also the area most closely related to advanced middleware concepts. This area of research addresses the need for runtime monitoring, feedback, and transition mechanisms to change application and system behavior, *e.g.,* through dynamic reconfiguration, orchestrating degraded behavior, or even off-line recompilation. The primary requirements here are *measurement*, *reporting*, *control*, *feedback*, and *stability*. Each of these plays a significant role in delivering end-to-end QoS, not only for an individual application, but also for an aggregate system. A key part of a runtime environment centers on a permanent and highly tunable measurement and resource status service [Quo01] as a common middleware service, oriented to various granularities for different time epochs and with abstractions and aggregations appropriate to its use for runtime adaptation.

   In addition to providing the capabilities for enabling graceful degradation, these same underlying mechanisms also hold the promise to provide flexibility that supports a variety of possible behaviors, without changing the basic implementation structure of applications. This reflective flexibility diminishes the importance of many initial design decisions by offering late- and runtime-binding options to accommodate actual operating environments at the time of deployment, instead of only anticipated operating environments at design time. In addition, it anticipates changes in these bindings to accommodate new behavior.

3. ***Aggregate Requirements*** – This area of research deals with the system view of collecting necessary information over the set of resources across the system, and providing resource management mechanisms and policies that are aligned with the goals of the system as a whole. While middleware itself cannot manage system-level resources directly (except through interfaces provided by lower level resource management and enforcement mechanisms), it can provide the coordinating mechanisms and policies that drive the individual resource managers into domain-wide coherence. With regards to such resource management, policies need to be in place to guide the decision-making process and the mechanisms to carry out these policy decisions.

   Areas of particular R&D interest include:
   - *Reservations*, which allow resources to be reserved to assure certain levels of service
   - *Admission control mechanisms*, which allow or reject certain users access to system resources
   - *Enforcement mechanisms* with appropriate scale, granularity and performance and
   - *Coordinated strategies and policies* to allocate distributed resources that optimize various properties.

   Moreover, policy decisions need to be made to allow for varying levels of QoS, including whether each application receives guaranteed, best-effort, conditional, or statistical levels of service. Managing property composition is essential for delivering individual QoS for component based applications, and is of even greater concern in the aggregate case, particularly in the form of layered resource management within and across domains.

4. ***Integration Requirements*** – Integration requirements address the need to develop interfaces with key building blocks used for system construction, including the OS, network management, security, and data management. Many of these areas have partial QoS solutions underway from their individual perspectives. The problem today is that these partial results must be integrated into a common interface so that users and application developers can tap into each, identify which viewpoint will be dominant under which conditions, and support the tradeoff management across the boundaries to get the right mix of attributes. Currently, object-oriented tools working with DOC middleware provide end-to-end syntactic interoperation, and relatively seamless linkage across the networks and subsystems. There is no *managed* QoS, however, making these tools and middleware useful only for resource rich, best-effort environments.

   To meet varying requirements for integrated behavior, advanced tools and mechanisms are needed that permit requests for *different* levels of attributes with different tradeoffs governing this interoperation. The system would then either provide the requested end-to-end QoS, reconfigure to provide it, or indicate the inability to deliver that level of service, perhaps offering to support an alternative QoS, or triggering application-level adaptation. For all of this to work together properly, multiple dimensions of the QoS requests must be understood within a common framework to translate and communicate those requests and services at each relevant interface. Advanced integration middleware provides this common framework to enable the right mix of underlying capabilities.

5. ***Adaptivity Requirements*** – Many of the advanced capabilities in next-generation information environments will require adaptive behavior to meet user expectations and smooth the imbalances between demands and changing environments. Adaptive behavior can be enabled through the appropriate organization and interoperation of the capabilities of the previous four areas. There are two fundamental types of adaptation required:
   1. Changes beneath the applications to continue to meet the required service levels despite changes in resource availability and
   2. Changes at the application level to either react to currently available levels of service or request new ones under changed circumstances.

   In both instances, the system must determine if it needs to (or can) reallocate resources or change strategies to achieve the desired QoS. Applications need to be built in such a way that they can change their QoS demands as the conditions under which they operate change. Mechanisms for reconfiguration need to be put into place to implement new levels of QoS as required, mindful of both the individual and the aggregate points of view, and the conflicts that they may represent.

   Part of the effort required to achieve these goals involves continuously gathering and instantaneously analyzing pertinent resource information collected as mentioned above. A complementary part is providing the algorithms and control mechanisms needed to deal with rapidly changing demands and resource availability profiles and configuring these mechanisms with varying service strategies and policies tuned for different environments. Ideally, such changes can be dynamic and flexible in handling a wide range of conditions, occur intelligently in an automated manner, and can handle complex issues arising from composition of adaptable components. Coordinating the tools and methodologies for these capabilities into an effective adaptive middleware should be a high R&D priority.

6. ***System Engineering Tools*** – Advanced middleware by itself will not deliver the capabilities envisioned for next-generation embedded environments. We must also advance the state of the system engineering tools that come with these advanced environments used to build complex distributed computing 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 (this 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.

7. ***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, longstanding 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 life cycles with minimal change and exhaustive test case analysis are clearly inadequate for next-generation dynamic distributed systems of systems with stringent QoS requirements.

4 Concluding Remarks

Distributed object computing (DOC) middleware is an important technology that is helping to decrease the cycle-time, level of effort, and complexity associated with developing high-quality, flexible, and interoperable distributed and embedded applications. Increasingly, these types of applications are developed using reusable software (middleware) component services, rather than being implemented entirely from scratch for each use. DOC middleware was invented in an attempt to help simplify the software development of network-centric 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. Complex system integration requirements were not being met from either the *application perspective*, where it was too difficult 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 network-centric applications and managing their distributed resources effectively. Consequently, one of the major trends driving industry involves moving toward a multi-layered architecture (applications, middleware, network and operating system infrastructure), which 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 led to more ambitious efforts and expansion of the scope of these middleware-oriented activities, so we now see a number of distinct layers of the middleware itself taking shape. The result has been a deeper understanding of the large and growing issues and potential solutions in the space between:

1. Complex distributed application requirements and
2. The simpler infrastructure provided by bundling existing network systems, operating systems, and programming languages.

There are significant limitations with regards to building these more complex systems today. For example, applications have increasingly more stringent requirements. We are also discovering that more things need to be integrated over conditions that more closely resemble a volatile, 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.

In the brief space of this article, we can do little more than 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 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. Downstream, the goals of these activities are to:

1. Reliably and repeatably construct and compose network-centric 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 DOC middleware.

5 Acknowledgements

We would like to thank to Don Hinton, Joe Loyall, Jeff Parsons, Andrew Sutton, and Franklin Webber for comments that helped to improve this article. Thanks also to members of the Cronus, ACE, TAO, and QuO user communities who have helped to shape our thinking on DOC middleware for over a decade.

6 References

[Beck00] Beck K., *eXtreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, MA, 2000.

[Ber96] Bernstein, P., "Middleware, A Model for Distributed System Service"", *Communications of the ACM*, 39:2, February 1996.

[Bla99] Blair, G.S., 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.

[Bol00] Bollella, G., Gosling, J. "The Real-Time Specification for Java," *Computer*, June 2000.

[Box97] Box D., *Essential COM*, Addison-Wesley, Reading, MA, 1997.

[Bus96] Buschmann, F., Meunier R., Rohnert H., Sommerlad P., Stal M., *Pattern-Oriented Software Architecture- A System of Patterns*, Wiley and Sons, 1996

[Chris98] Christensen C., *The Innovator's Dilemma: When New Technology Causes Great Firms to Fail*, 1997.

[Cuk98] Cukier, M., Ren J., Sabnis C., Henke D., Pistole J., Sanders W., Bakken B., Berman M., Karr D. Schantz R., "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects ", *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245-253, October 1998.

[Dar99] DARPA, *The Quorum Program*, http://www.darpa.mil/ito/research/quorum/index.html, 1999.

[Doe99] Doerr B., Venturella T., Jha R., Gill C., Schmidt D. "Adaptive Scheduling for Real-time, Embedded Information Systems," *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Louis, Missouri, October 1999.

[Gam95] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[John97] Johnson R., "Frameworks = Patterns + Components", *Communications of the ACM*, Volume 40, Number 10, October, 1997.

[JVM97] Lindholm T., Yellin F., *The Java Virtual Machine Specification*, Addison-Wesley, Reading, MA, 1997.

[Kar01] Karr DA, Rodrigues C, Loyall JP, Schantz RE, Krishnamurthy Y, Pyarali I, Schmidt DC. "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," *Proceedings of the International Symposium on Distributed Objects and Applications*, September 18-20, 2001, Rome, Italy.

[Loy01] Loyall JL, Gossett JM, Gill CD, Schantz RE, Zinky JA, Pal P, Shapiro R, Rodrigues C, Atighetchi M, Karr D. "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications". *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, April 16-19, 2001, Phoenix, Arizona.

[NET01] Thai T., Lam H., *.NET Framework Essentials*, O'Reilly, 2001.

[Omg98a] Object Management Group, "Fault Tolerance CORBA Using Entity Redundancy RFP", OMG Document orbos/98-04-01 edition, 1998.

[Omg98b] Object Management Group, "CORBAServcies: Common Object Service Specification," OMG Technical Document formal/98-12-31.

[Omg99] Object Management Group, "CORBA Component Model Joint Revised Submission," OMG Document orbos/99-07-01.

[Omg00] Object Management Group, "The Common Object Request Broker: Architecture and Specification Revision 2.4, OMG Technical Document formal/00-11-07", October 2000.

[Omg00A] Object Management Group. "Minimum CORBA," OMG Document formal/00-10-59, October 2000.

[Omg00B] Object Management Group. "Real-Time CORBA," OMG Document formal/00-10-60, October 2000.

[Omg01] Object Management Group, "Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission," OMG Document orbos/2001-04-01.

[Quo01] *Quality Objects Toolkit v3.0 User's Guide*, chapter 9, available as http://www.dist-systems.bbn.com/tech/QuO/release/latest/docs/usr/doc/quo-3.0/html/QuO30UsersGuide.htm

[RUP99] Jacobson I., Booch G., and Rumbaugh J., *Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Sch86] Schantz, R., Thomas R., Bono G., "The Architecture of the Cronus Distributed Operating System", *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS-6)*, Cambridge, Massachusetts, May 1986.

[Sch98] Schantz, RE, "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 (also available as: http://www.dist-systems.bbn.com/papers/1998/CaseStudy).

[Sch98A] Schmidt D., Levine D., Mungee S. "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), pp. 294—324, 1998.

[Sch00a] Schmidt D., Kuhns F., "An Overview of the Real-time CORBA Specification," *IEEE Computer Magazine*, June, 2000.

[Sch00b] Schmidt D., Stal M., Rohnert H., Buschmann F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley and Sons, 2000.

[Sch01] Schmidt D., Huston S., *C++ Network Programming: Resolving Complexity with ACE and Patterns*, Addison-Wesley, Reading, MA, 2001.

[Sha98] Sharp, David C., “Reducing Avionics Software Cost Through Component Based Product Line Development”, *Software Technology Conference*, April 1998.

[SOAP01] Snell J., MacLeod K., *Programming Web Applications with SOAP*, O’Reilly, 2001.

[Ste99] Sterne, D.F., G.W. Tally, C.D. McDonell et al, “Scalable Access Control for Distributed Object Systems”, *Proceedings of the 8th Usenix Security Symposium*, August,1999.

[Sun99] Sun Microsystems, “Jini Connection Technology”, http://www.sun.com/jini/index.html, 1999.

[TPA97] Sabata B., Chatterjee S., Davis M., Sydir J., Lawrence T., “Taxonomy for QoS Specifications,” *Proceedings of Workshop on Object-oriented Real-time Dependable Systems (WORDS 97)*, February 1997.

[Tho98] Thomas, Anne “Enterprise JavaBeans Technology”, http://java.sun.com/products/ejb/white_paper.html, Dec. 1998

[Wol96] Wollrath A., Riggs R., Waldo J. “A Distributed Object Model for the Java System,” *USENIX Computing Systems*, 9(4), 1996.