Research Advances in Middleware for Distributed Systems: State of the Art

Richard E. Schantz
BBN Technologies
schantz@bbn.com

Douglas C. Schmidt
Electrical & Computer Engineering Dept.
University of California, Irvine
schmidt@uci.edu

1 INTRODUCTION

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 these interconnections can range from very small and tightly coupled systems, such as avionics mission computing systems, to very large and loosely coupled systems, such as global telecommunications systems and so-called "grid" computing.

The interplay of these two trends has yielded new architectural concepts and services embodying layers of *middleware*. Middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to:

1. Make it feasible, easier, and more cost effective to develop and evolve distributed systems
2. Coordinate how parts of applications are connected and how they interoperate and

3. Enable and simplify the integration of components developed by multiple technology suppliers.

The growing importance of middleware stems from the recognition of 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 therefore focused on

1. Identifying, evolving, and expanding our understanding of current middleware services to support the network-centric paradigm and
2. 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 systems.

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

- ***Years of iteration, refinement, and successful use*** – The use of middleware and middleware oriented system architectures is not new [Sch86, Sch98, Ber96]. Middleware concepts emerged alongside experimentation with the early Internet (and even its predecessor the ARPAnet) and systems based on middleware have been operational continuously 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 frameworks, components, and patterns that reify the knowledge of how to apply these technologies, along with standards that codify the boundaries of these technologies, as described next.
- ***The dissemination of middleware frameworks, components, and patterns*** – During the past decade, a substantial amount of R&D effort has focused on developing *frameworks, components,* and *patterns* 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 and components are concrete realizations of groups of related patterns [John97]. Well-designed frameworks reify patterns in terms of functionality provided by components in the middleware itself, as well as functionality provided by an application. Frameworks also integrate various approaches to problems where there are no *a priori*, context-independent, optimal solutions. Middleware frameworks [Sch02] and component toolkits 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 maturation of middleware standards*** – Also over the past decade, standards for middleware frameworks and components have been established and have matured considerably. For instance, the Object Management Group (OMG) has defined the following specifications for CORBA [Omg00] in the past several years:

    - *CORBA Component Model*, which standardizes component implementation, packaging, and deployment to simplify server programming and configuration
    - *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.

These middleware specifications go well beyond the interconnection standards conceived originally and reflect attention to more detailed issues beyond basic connectivity and remote procedure calls. Robust implementations of these CORBA capabilities and services are now available from multiple suppliers, some using open-source business models and some using traditional business models.

In addition to CORBA, some other notable successes to date in the domain of middleware services, frameworks, components, and standards include:

- Java 2 Enterprise Edition (J2EE) [Tho98] and .NET *[NET01]*, which have introduced advanced software engineering capabilities to the mainstream IT community and which incorporate various levels of middleware as part of the overall development process, albeit with only partial support for performance critical and embedded solutions.
- Akamai et al, which have legitimized a form of middleware service as a viable business, albeit using proprietary and closed, non-user programmable solutions.
- Napster, which demonstrated the power of having a powerful, commercial-off-the-shelf (COTS) middleware infrastructure to start from in quickly (weeks/months) developing a very capable system, albeit without much concern for system lifecycle and software engineering practices, i.e., it is one of a kind.
- WWW, where the world wide web middleware/standards led to easily connecting independently developed browsers and web pages, albeit also the world wide *wait*, because there was no system engineering or attention paid to enforcing end-to-end quality of service issues.
- The Global Grid, which is enabling scientists and high performance computing researchers to collaborate on grand challenge problems, such as global climate change modeling, albeit using architectures and tools that are not yet aligned with mainstream IT COTS middleware.

These competing and complementary forms of and approaches to middleware based solutions represent simultaneously a healthy and robust technical area of continuing innovation, and a source of confusion due to the multiple forms of similar capabilities, patterns, and architectures.

2 ADDRESSING DISTRIBUTED APPLICATION CHALLENGES WITH MIDDLEWARE

Requirements for faster development cycles, decreased effort, and greater software reuse motivate the creation and use of middleware and middleware-based architectures. When implemented properly, middleware helps 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 reuseable, off-the-shelf developer-oriented services, such as naming, logging and security that have proven necessary to operate effectively in a networked environment.

Over the past decade, various middleware technologies have been devised to alleviate many complexities associated with developing software for distributed applications. Their successes have added middleware as a new category of systems software to complement the familiar operating system, programming language, networking, and database offerings of the previous generation. Some of the most successful middleware technologies have centered on *distributed object computing (DOC)*. DOC is an advanced, mature, and field-tested middleware connectivity paradigm that also 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 are increasingly forming the basis of large-scale distributed system deployments.

2.1 The 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.

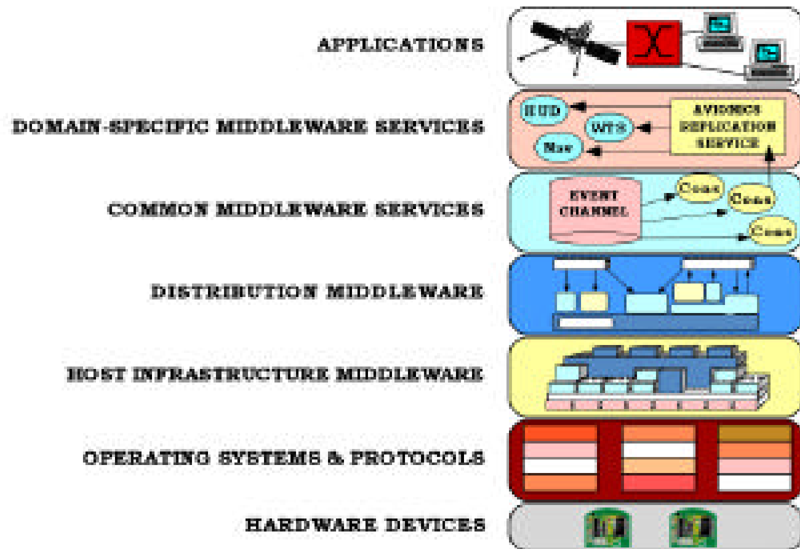

Figure 1. Layers of DOC Middleware and Surrounding Context

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.

2.1.1 Host infrastructure middleware

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]. 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) is a freely available, highly portable toolkit that shields applications from differences between native OS programming capabilities, such as file handling, connection establishment, event demultiplexing, interprocess communication, (de)marshaling, concurrency, and synchronization. ACE provides an OS adaptation layer and wrapper facades [Sch02] that encapsulate OS file system, concurrency, and network programming mechanisms. ACE also provides reusable frameworks [Sch03] that handle network programming tasks, such as synchronous and asynchronous event handling, service configuration and initialization, concurrency control, connection management, and hierarchical service integration.

The primary differences between ACE, JVMs, and the .NET CLR are that (1) ACE is always a compiled C++ interface, rather than an interpreted bytecode interface, which removes a level of indirection and helps to optimize runtime performance, (2) ACE is open-source, so it's possible to subset it or modify it to meet a wide variety of needs, and (3) ACE runs on more OS and hardware platforms than JVMs and CLR.

2.1.2 Distribution middleware

Distribution middleware defines higher-level distributed programming models whose reusable APIs and components automate and extend the native OS network programming capabilities encapsulated by host infrastructure middleware. Distribution middleware enables clients to program distributed 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 allows objects to interoperate across networks regardless of the language in which they were written or the platform on which they are deployed.
- Sun's Java Remote Method Invocation (RMI) [Wol96], which 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 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.

An example of distribution middleware R&D is the TAO project [Sch98a] conducted by researchers at Washington University, St. Louis, the University of California, Irvine, and Vanderbilt University as part several DARPA programs. TAO is an open-source Real-time CORBA ORB that allows distributed real-time and embedded (DRE) applications to reserve and manage

- *Processor resources* via thread pools, priority mechanisms, intra-process mutual exclusion mechanisms, 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 ACE [Sch02, Sch03] host infrastructure middleware toolkit. ACE and TAO are mature examples of middleware R&D transition, having been used in hundreds of DRE systems, including 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 mills.

2.1.3 Common middleware services

Common middleware services augment distribution middleware by defining higher-level domain-independent services that allow application developers to concentrate on programming business logic, without the need to write the "plumbing" code required to develop distributed 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 Java 2 Enterprise Edition (J2EE) technology [Tho98], which allows developers to create n-tier distributed systems by linking a number of pre-built software services—called "Javabeans"—without having to write much code from scratch. Since J2EE is built on top of Java technology, J2EE service components can only be implemented using the Java language. The CORBA Component Model (CCM)

[Omg99] defines a superset of J2EE 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.

2.1.4 Domain-specific middleware services

Domain-specific middleware services are tailored to the requirements of particular domains, such as telecom, e-commerce, health care, process automation, or aerospace. Unlike the other three 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(R), 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(R) 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 is 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 The Benefits of DOC Middleware

Middleware in general–and DOC middleware in particular–provides essential capabilities for developing an increasingly large class of distributed applications. In this section we summarize some of the

improvements and areas of focus in which middleware oriented approaches are having significant impact.

2.2.1 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

- Applications, which at best were customized for a single use,
- Networks, which were necessarily concerned with providing the communication layer, or
- Host operating systems, which were focused primarily on a single, self-contained unit of resources.

In contrast, middleware has a fundamental integration focus, which stems from incorporating the perspectives of both 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 being programmed either entirely from scratch or requiring significant customization or augmentation to off-the-shelf component implementations.

2.2.2 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. This in turn yields the potential for multiple choices for component implementations and to 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 [Quorum99] 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.

2.2.3 Increased leverage for disruptive technologies leading to increased global competition

Middleware that supports 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 frequently used common components, which come with steady innovation cycles resulting from a multi-user basis, in conjunction with custom domain-specific capabilities, which allow appropriate mixing of multi-user low cost and custom development for competitive advantage.

2.2.4 Growing focus on real-time embedded environments integrating computational and real world physical assets

Historically, conventional COTS software has been unsuitable for use in mission-critical distributed systems due to its either being flexible and standard, but incapable of guaranteeing stringent QoS demands (which restricts assurability) or partially QoS-enabled, but inflexible and non-standard (which restricts adaptability and affordability). As a result, the rapid progress in COTS software for mainstream desktop business information technology (IT) has not yet become as broadly applicable for mission-critical distributed systems. However, progress is being made today in the laboratory, in technology transition, in COTS products, and in standards. Although off-the-shelf middleware technology has not yet matured to cover the realm of large-scale, dynamically changing systems, DOC middleware has been applied to relatively small-scale and statically configured embedded systems [Sha98, NAS94]. Moreover, significant pioneering R&D on middleware patterns, frameworks, and standards for distributed systems has been conducted in the DARPA *Quorum* [Quorum99] and *PCES* [PCES02] programs, which played a leading role in:

1. Demonstrating the viability of integrating host infrastructure middleware, distribution middleware and common middleware services for DoD real-time embedded systems by providing foundation elements for managing key QoS attributes, such as real time behavior,

dependability and system survivability, from a network-centric middleware perspective
2. Transitioning a number of new middleware perspectives and capabilities into DoD acquisition programs [Sha98, AegisOA, Holzer00] and commercially supported products and
3. Establishing the technical viability of collections of systems that can dynamically adapt within real-time constraints [Loy01] their collective behavior to varying operating conditions, in service of delivering the appropriate application level response under these different conditions.

3 FUTURE RESEARCH CHALLENGES AND STRATEGIES

In certain ways, each of the middleware successes mentioned in *Section 1 Introduction* can also be considered a partial failure, especially when viewed from a more complete perspective. In addition, other notable failures come from Air Traffic control, late opening of the Denver Airport, lack of integration of military systems causing misdirected targeting, and countless number of smaller, less visible systems which are cancelled, or are fielded but just do not work properly. More generally, connectivity among computers and between computers and physical devices, as well as connectivity options, is proliferating unabated, which leads to society's demand for network-centric systems of increasing scale and demanding precision to take advantage of the increased connectivity to better organize collective and group interactions/behaviors. Since these systems are growing (and will keep growing) their complexity is increasing, which motivates the need to keep application programming relatively independent of the complex issues of distribution and scale (in the form of advanced software engineering practices and middleware solutions). In addition, systems of national scale, such as the US air traffic control system or power grid, will of necessity be incremental and developed by many different organizations contributing to a common solution on an as yet undefined common high-level platform and engineering development paradigm.

Despite all the advances in the past decades, there are no mature engineering principles, solutions, or established conventions to enable large-scale, network-centric systems to be repeatably, predictably, and cost effectively created, developed, validated, operated, and enhanced. As a result, we are witnessing a complexity threshold that is stunting our ability to create large-scale, network-centric systems successfully. Some of the inherent properties that contribute to this complexity threshold include:

- Discrete platforms must be scaled to provide seamless end-to-end solutions
- Components are heterogeneous yet they need to be integrated seamlessly
- Most failures are only partial in that they effect subsets of the distributed components
- Operating environments and configurations are dynamically changing
- Large-scale systems must operate continuously, even during upgrades
- End-to-end properties must be satisfied in time and resource constrained environments
- Maintaining system-wide QoS concerns is expected

As described earlier, 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 that serve as the linkage between application requirements and resource management strategies. Having a clear understanding of the QoS information is important so that it becomes possible to:

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

This augmentation is beginning to occur, but largely on a component-by-component basis, not end-to-end. 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 new 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 conditions are constantly changing and not all requirements can be met all of the time, yet still ensuring predictable and controllable end-to-end behavior.

Ironically, there is little or no scientific underpinning for QoS-enabled resource management, despite the demand for it in most distributed systems [Narain01]. Designers of today's complex distributed systems develop concrete plans for creating global, end-to-end functionality. These plans contain high-level abstractions and doctrine associated with resource management algorithms, relationships between these, and operations upon these. There are few techniques and tools that enable *users (e.g.,*

commanders, administrators, and operators), *developers* (*e.g.,* systems engineers and application designers), and/or *applications,* to express such plans systematically, and to have these plans integrated and enforced automatically for managing resources at multiple levels in network-centric embedded systems.

Although there are no well accepted standards in these areas, work is progressing toward better understanding of these issues. To achieve these goals, middleware technologies and tools need to be based upon some type of layered architecture that is imbued with QoS adaptive middleware services. Figure 2 illustrates one such approach that is based on the Quality Objects (QuO) [ZBS97] project.

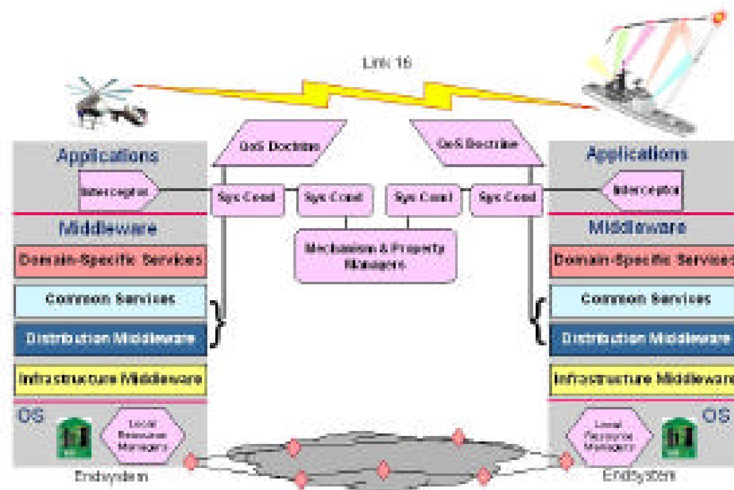

Figure 2. Decoupling Functional and QoS Attribute Paths

The QuO project and empirical demonstrations based on QuO middleware [Loy01, WMDS] are an example of how one might organize such a layered architecture designed to manage and package adaptive QoS capabilities [Sch02A] as common middleware services. The QuO architecture decouples DOC middleware and applications along the following two dimensions:

- *Functional paths,* which are flows of information between client and remote server applications. In distributed systems, middleware ensures that this information is exchanged efficiently, predictably, scalably, dependably, and securely between remote peers. The information itself

is largely application-specific and determined by the functionality being provided (hence the term "functional path").

- *QoS attribute paths*, which are responsible for determining how well the functional interactions behave end-to-end with respect to key distributed system QoS properties, such as

  - How and when resources are committed to client/server interactions at multiple levels of distributed systems
  - The proper application and system behavior if available resources are less than the expected resources and
  - The failure detection and recovery strategies necessary to meet end-to-end dependability requirements.

In the architecture shown in Figure 2, the QuO middleware is responsible for collecting, organizing, and disseminating QoS-related meta-information that is needed to

1. Monitor and manage how well the functional interactions occur at multiple levels of distributed systems and
2. Enable the adaptive and reflective decision-making needed to support QoS attribute properties robustly in the face of rapidly changing mission requirements and environmental conditions.

In next-generation distributed systems, separating systemic QoS attribute properties from the functional application properties will enable the QoS properties and resources to change independently*, e.g.,* over different distributed system configurations for the same application, and despite local failures, transient overloads, and dynamic functional or QoS reconfigurations. 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 distributed 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, perhaps hierarchically, so they can adapt to a wider variety of situations.

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 important attributes needed to support future systems, especially mision critical and embedded distributed 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. As the uses and environments for 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.

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. We must create and deploy middleware-oriented solutions and engineering principles as part of the commonly available new, network-centric software infrastructure that is needed to develop many different types of large-scale systems successfully. The payoff will be reusable DOC middleware that significantly simplifies and reduces the inherent risks in building applications for complex systems of systems environments.

The remainder of this section presents an analysis of the challenges and opportunities for next-generation middleware and outlines the promising research strategies that can help to overcome the challenges and realize the opportunities.

3.1 Specific R&D Challenges

An essential part of what is needed to alleviate the inherent complexities outlined in the discussions 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 return on investment will yield reusable middleware that significantly simplifies the development and evolution of complex network-centric systems. The following are specific R&D challenges associated with achieving this payoff:

3.1.1 Providing end-to-end QoS support, not just component-level QoS

This area represents the next great wave of evolution for advanced middleware. There is now widespread recognition that effective development of large-scale network-centric 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 *predictable*, *flexible*, and *integrated* resource management strategies, both within and between the pieces, that are understandable to developers, visible to users, and certifiable to system owners. 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, 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.

The need for autonomous and time-critical behavior necessitates more flexible system infrastructure components that can adapt robustly to dynamic end-to-end changes in application requirements and environmental conditions. 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.

Two basic premises underlying the push towards end-to-end QoS support mediated by middleware are that different levels of service are possible and desirable under different conditions and costs and 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.

3.1.2 Adaptive and reflective solutions that handle 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 needed to carry out such reconfiguration. To date interoperability concerns have focused on data interoperability and invocation interoperability across components. Little work has focused on mechanisms for controlling the overall behavior of the end-to-end integrated systems. "Control interoperability" is needed to complement data and invocation interoperability if we are to achieve something more than a collection of independently operating components. There are requirements for interoperable control capabilities to appear in the individual resources first, after which approaches can be developed to aggregate these into acceptable global behavior through middleware based multi-platform aggregate resource management services.

To manage the broader range of QoS demands for next-generation network-centric applications, middleware must become more adaptive and reflective [ARMS01]. *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 manipulatible for middleware and application programs to inspect and modify at run-time. Thus, reflective middleware supports more advanced adaptive behavior and more dynamic strategies keyed to current circumstances, *i.e.,* necessary adaptations can be performed autonomously based on conditions within the system, in the system's environment, or in system QoS policies defined by end-users.

3.1.3 Combining model-integrated computing with DOC middleware

It has been increasingly recognized that source code is a poor way to document distributed system designs. Starting from informal design documentation techniques, such as flow-charts, model-integrated computing (MIC) is evolving DOC middleware towards more formal, semantically rich high-level design languages, and toward systematically capturing core aspects of designs via patterns, pattern languages, and architectural styles [MIC97]. MIC technologies are expanding their focus beyond application functionality to specify application quality of service (QoS) requirements, such as real-time deadlines and dependability constraints. These model-based tools provide application and middleware developers and integrators with higher levels of abstraction and productivity than traditional imperative programming languages provide. The following are some of the key R&D challenges associated with combining MIC and DOC middleware:

– Determine how to overcome problems with earlier-generation CASE environments that required the modeling tools to generate all the code. Instead, the goal should be to compose large portions of distributed applications from reusable, prevalidated DOC middleware components.
– Enhancing MIC tools to work in distributed environments where run-time procedures and rules change at rapid pace, e.g., by synthesizing,

assembling, and validating newer extended components that conform to new rules that arise after a distributed system has been fielded.

- Devising domain-specific MIC languages that make DOC middleware more flexible and robust by automating the configuration of many QoS-critical aspects, such as concurrency, distribution, transactions, security, and dependability. This MIC-synthesized code may be needed to help bridge interoperability and portability problems between different middleware for which standard solutions do not yet exist.
- Train MIC tools to model the interfaces among various components in terms of standard middleware, rather than language-specific features or proprietary APIs.

3.1.4 Toward more universal use of standard 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 network-centric systems.

3.1.5 Leveraging and extending the installed base

In addition to the R&D challenges outlined above there are also 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 Fundamental Research Concepts

The following four concepts are central to addressing the R&D challenges described above:

3.2.1 Contracts and adaptive meta-programming

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, high-level middleware abstraction to improve the visibility of adaptive changes in the mandated behavior.

3.2.2 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.

3.2.3 Prioritization and physical world constrained load invariant performance

Some systems are highly correlated with physical constraints and have little flexibility in some of their requirements for computing assets,

including QoS. Deviation from requirements beyond a narrowly defined error tolerance can sometimes result in catastrophic failure of the system. The challenge is in meeting these *invariants* under varying load conditions. This often means guaranteeing access to some resources, while other resources may need to be diverted to insure proper operation. Generally collections of such components will need to be resource managed from a system (aggregate) perspective in addition to a component (individual) perspective.

3.2.4 Higher level design approaches, abstractions, and software development tools

Better techniques and more automated tools are needed to organize, integrate, and manage the software engineering paradigm and process used to construct the individual elements, the individual systems, and the systems of systems, without resorting to reimplementation for composition. Promising results in applying and embedding model-based software development practices [MIC97], as well as decomposition by aspects or views, suggest that applying similar design time approaches to QoS engineering may complement and make easier the runtime adaptation needed to control and validate these complex systems.

3.3 Promising Research Strategies

Although it is possible to satisfy contracts, achieve graceful degradation, and use modeling tools to globally manage some resources to a limited degree in a limited range of systems today, much R&D work remains. The research strategies needed to deliver these goals can be divided into the seven areas described below:

3.3.1 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 network-centric 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) 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.

3.3.2 Run-time Requirements

From a system lifecycle perspective, decisions for managing QoS are made at design time, at configuration/deployment time, and/or at run-time. Of these, the run-time 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 run-time 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 run-time environment centers on a permanent and highly tunable measurement and resource status service as a common middleware service, oriented to various granularities for different time epochs and with abstractions and aggregations appropriate to its use for run-time 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 run-time-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.

3.3.3 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.

3.3.4 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.

3.3.5 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.

3.3.6 System Engineering Methodologies and 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 discipline and 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 approaches and tools to augment advanced middleware solutions. These include:

- *View-oriented or aspect-oriented programming techniques,* to support the isolation (for specialization and focus) and the composition (to mesh the isolates into a whole) of different projections or views of the properties the system must have. The ability to isolate, and

subsequently integrate, the implementation of different, interacting features will be needed to support adapting to changing requirements.

- *Design time tools and model-integrated computing technologies*, 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 performance 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.

3.3.7 Reliability, Trust, Validation, and Certifiability

The dynamically changing behaviors we envision for next-generation large-scale, network-centric 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 systems with stringent QoS requirements.

4 CONCLUDING REMARKS

In this age of IT ubiquity, economic upheaval, deregulation, and stiff global competition it has become essential to decrease the cycle-time, level of effort, and complexity associated with developing high-quality, flexible, and interoperable large-scale, network-centric systems. Increasingly, these types of systems are developed using reusable software (middleware)

component services, rather than being implemented entirely from scratch for each use. Middleware was invented in an attempt to help simplify the software development of large-scale, network-centric 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, distributed object computing (DOC) middleware has emerged as a set of software protocol and 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 researchers and practioners involves

1. Moving toward a multi-layered architecture (i.e., applications, middleware, network and operating system infrastructure), which is oriented around application composition from reusable components, and
2. Moving 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 has 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 complex distributed application requirements and the simpler infrastructure provided by bundling existing network systems, operating systems, and programming languages. Network-centric systems today are constructed as a series of layers of intertwined technical capabilities and innovations. The main emphasis at the lower middleware layers is in providing standardized core computing and communication resources and services that drive network-centric computing: overlays for the individual computers, the

networks, and the operating systems that control the individual host and the message level communication.

At the upper layers, various types of middleware are starting to bridge the previously formidable gap between the lower-level resources and services and the abstractions that are needed to program, organize, and control systems composed of coordinated, rather than isolated, components. Key new capabilities in the upper layers include common and domain-specific middleware services that

- Enforce real-time behavior across computational nodes
- Manage redundancy across elements to support dependable computing and
- Provide coordinated and varying security services on a system wide basis, commensurate with the threat

There are significant limitations with regards to building these more complex systems today. For example, applications have increasingly more stringent QoS 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. Adaptive and reflective middleware systems [ARMS01] are a key emerging paradigm that will help to simplify the development, optimization, validation, and integration for distributed systems.

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 paper, we can do little more than summarize and lend perspective to the many activities, past and present, that contribute to making DOC middleware technology an area of exciting current development, along with considerable opportunity and unsolved challenging problems. We have provided many 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

R&D directions of middleware for large-scale, network-centric systems. Downstream, the goals of these R&D 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 and advanced network-centric infrastructures.

5 ACKNOWLEDGEMENTS

We would like to thank Don Hinton, Joe Loyall, Jeff Parsons, Andrew Sutton, Franklin Webber, and members of the Large-scale, Network-centric Systems working group at the Software Design and Producitivity workshop at Vanderbilt University, December 13-14, 2001 for comments that helped to improve this paper. 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

[AegisOA] Guidance Document for Aegis Open Architecture Baseline Specification Development, Version 2.0 (Draft), 5 July 2001.

[ARMS01] Schmidt D., Schantz R., Masters M., Sharp D., Cross J., and DiPalma L., “Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems, Crosstalk, November 2001.

[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.

[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.

[Holzer00] Holzer R., "U.S. Navy Looking for More Adaptable Aegis Radar," *Defense News*, 18 September 2000.

[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.

[MIC97] Janos Sztipanovits and Gabor Karsai, "Model-Integrated Computing," IEEE Computer, Volume 30, Number 4, April 1997.

[Narain01] Narain S., Vaidyanathan R., Moyer S., Stephens W., Parameswaran K., and Shareef A., "Middle-ware For Building Adaptive Systems via Configuration," *ACM Optimization of Middleware and Distributed Systems (OM 2001) Workshop*, Snowbird, Utah, June, 2001.

[NAS94] New Attack Submarine Open System Imple-mentation, Specification and Guidance, August 1994.

[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, "CORBAServices: 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.

[PCES02] The Programmable Composition of Embedded Software (PCES) Project, DARPA Information Exploitation Office. http://www.darpa.mil/ito/research/pces/index.html

[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

[Quorum99] DARPA, *The Quorum Program*, http://www.darpa.mil/ito/research/quorum/index.html, 1999.

[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).

[Sch02A] Schantz, R., Loyall, J., Atighetchi, M., Pal, P., "Packaging Quality of Service Control Behaviors for Reuse", ISORC 2002, *The 5th IEEE International Symposium on Object-oriented Real-time*

distributed Computing, April 29 - May 1, 2002, Washington, DC. …

[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.

[Sch02] Schmidt D., Huston S., *C++ Network Programming: Resolving Complexity with ACE and Patterns*, Addison-Wesley, Reading, MA, 2002.

[Sch03] Schmidt D., Huston S., *C++ Network Programming: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, Reading, MA, 2003.

[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.