EDITORIAL: TRENDS IN DISTRIBUTED OBJECT COMPUTING

Limitations with Current Practice. During the past decade advances in VLSI technology and fiber-optics have increased computer processing power by 3-4 orders of magnitude and network link speeds by 6-7 orders of magnitude. Assuming that these trends continue, by the end of this decade there will be billions of interactive and embedded computing and communication devices throughout the world running at clock speeds approaching $\sim$100 Gigahertz, LAN link speeds running at $\sim$100 Gigabits/second, and wireless link speeds running at $\sim$100 Megabits/second. These powerful computers and networks will be available largely at commodity prices, built mostly with robust commercial off-the-shelf (COTS) components, and will inter-operate over an increasingly convergent and pervasive Internet infrastructure.

To maximize the benefit from these advances in hardware technology, the quality and productivity of technologies for developing distributed middleware and application software must also increase. Historically, hardware has tended to become smaller, faster, and more reliable. It has also become cheaper and more predictable to develop and innovate, as evidenced by Moore's Law. In contrast, however, distributed software has often grown larger, slower, and more error-prone. It has also become very expensive and time-consuming to develop, validate, maintain, and enhance.

Although hardware improvements have alleviated the need for some low-level software optimizations, the lifecycle cost [2] and effort required to develop software–particularly mission-critical distributed and embedded real-time applications–continues to rise. The disparity between the rapid rate of hardware advances versus the slower software progress stems from a number of factors, including:

Inherent and accidental complexities.. There are vexing problems with distributed software that result from inherent and accidental complexities. Inherent complexities arise from fundamental domain challenges such as detecting and recovering from partial failures and distributed deadlocks, minimizing the impact of communication latency, determining an optimal partitioning of service components and workload onto computers throughout a network, and guaranteeing end-to-end quality of service (QoS) requirements. As networked systems have grown in scale and functionality they must now cope with a much broader and harder set of these complexities.

Accidental complexities arise from limitations with software tools and development techniques, such as non-portable programming APIs, poor distributed debuggers, and the widespread use of algorithmic–rather than object–oriented design, which results in non-extensible and non-reusable systems. Ironically, many accidental complexities stem from deliberate choices made by developers who favor low-level languages and tools that scale up poorly when applied to complex distributed software.

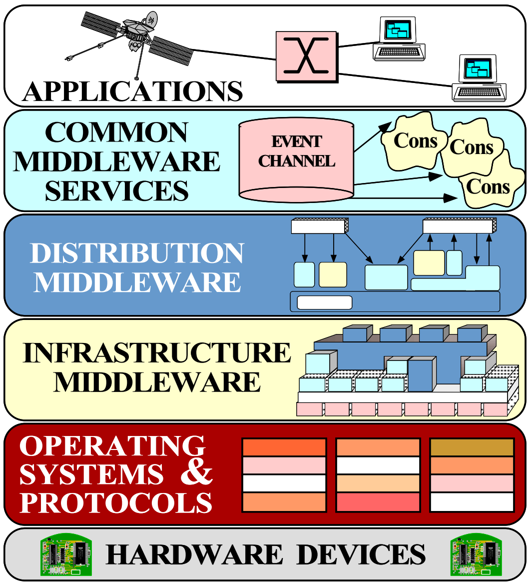
Continuous re-invention and re-discovery of core concepts and techniques.. The software industry has a long history of recreating incompatible solutions to problems that are already solved. For example, there are dozens of non-standard general-purpose and real-time operating systems that manage the same hardware resources. Likewise, there are dozens of message-oriented and method-oriented middleware frameworks that provide slightly different APIs that implement essentially the same features and services. If effort had instead been focused on enhancing and optimizing a small number of solutions, developers of distributed software would be reaping the benefits available to hardware developers, who innovate rapidly by reusing and applying common CAD tools and standard instruction sets, buses and network protocols.

Solution Approach: Distributed Object Computing. Obviously, no single silver bullet [4] can slay all the demons plaguing distributed software. Over the past decade, however, it has become clear that *distributed object computing* (DOC) can help to alleviate many inherent and accidental software complexities. DOC represents the confluence of two major areas of software technology:

- *Distributed computing systems* – Techniques for developing distributed systems focus on integrating multiple computers to act as a scalable computational resource.
- *Object-oriented (OO) design and programming* – Techniques for developing OO systems focus on reducing complexity by creating reusable frameworks and components that reify successful design patterns and software architectures.

Thus, DOC is the discipline that uses OO techniques to distribute reusable services and applications efficiently, flexibly, and robustly over multiple, often heterogeneous, computing and networking elements.

At the heart of contemporary distributed object computing is *DOC middleware*. DOC middleware is object-oriented software that resides between applications and the underlying operating systems, protocol stacks, and hardware to enable or simplify how these components are connected and interoperate, as shown in the following figure: In general, DOC



middleware can be decomposed into the following three layers:

Infrastructure middleware.. This layer encapsulates core OS communication and concurrency services to eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications using low-level network programming mechanisms, such as sockets. Widely-used examples of infrastructure middleware include the Java Virtual Machine (JVM) [8] and the ADAPTIVE Communication Environment (ACE) [12].

Distribution middleware.. This layer builds upon the lower-level infrastructure middleware to automate common network programming tasks, such as parameter marshaling/demarshaling, socket and request demultiplexing, and fault detection/recovery. Common examples of distribution middleware include the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) [9], Microsoft's Distributed COM (DCOM) [3], and JavaSoft's Remote Method Invocation (RMI) [13] .

Common middleware services. This layer contains domain-independent reusable service components [1]. Common middleware services include persistence, security, transactions, fault tolerance, and concurrency. Today, common services are the least mature layer of the middleware. In the future, however, this layer has the most potential to increase quality and decrease the cycle-time and effort required to develop distributed applications by supporting the integration of reusable service components implemented by different suppliers.

In general, these layers of DOC middleware provide the following benefits: (1) they shield software developers from low-level, tedious, and error-prone details, such as socket-level programming [12], (2) they provides a consistent set of higher-level abstractions [14] for developing distributed systems, (3) they amortize software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns [5] in reusable frameworks, rather than building them entirely from scratch for each use-case.

Emerging Trends in Distributed Object Computing R&D. The following trends are shaping the evolution of software development techniques for applications and middleware based on distributed object computing technologies:

Applying patterns to capture the "best practices" of distributed object computing.. Many patterns associated with middleware and applications for concurrent [7] and networked objects [11] have been documented during the past decade. A key next step is to document the patterns for designing [5] and optimizing [10] distributed objects, extending earlier work to focus on topics such as remote service location and partitioning, naming and directory services, load balancing, dependability and security. An increasing number of distributed object computing systems, for example, must provide high levels of dependability to client programs and end-users. With the adoption of the CORBA Fault Tolerance specification and ORBs that implement this specification, developers will have more opportunities to capture their experience in the form of patterns for fault-tolerant distributed object computing.

Real-time and embedded systems.. An increasing number of computing systems are embedded, including automotive control systems and car-based applications, control software for factory automation equipment, avionics mission computing and hand-held computing devices. Many of these systems are subject to stringent computing resource limitations, particularly memory footprint and time-constraints. Developing high-quality real-time and embedded systems is hard and remains somewhat of a "black art." As the efficiency, scalability, and predictability of DOC middleware continues to improve, and is increasingly capable of being subsetted to reduce footprint, we expect it will be applied heavily in these domains.

Mobile systems.. Wireless networks are becoming pervasive and embedded computing devices are become smaller, lighter and more capable. Thus, mobile systems will soon support many consumer communication and computing needs. Application areas for mobile systems include ubiquitous computing, mobile agents, personal assistants, position-dependent information provision, remote medical diagnostics and teleradiology and home and office automation. In addition, Internet services, ranging from Web browsing to on-line banking, will be accessed from mobile systems. Mobile systems present many challenges, such as managing low and variable bandwidth and power, adapting to frequent disruptions in connectivity and service quality, diverging protocols and maintaining cache consistency across disconnected network nodes. DOC middleware is essential to provide a flexible and adaptive framework for developing and deploying mobile systems.

Quality of service for common-off-the-shelf (COTS)-based distributed systems.. Distributed systems, such as streaming video, Internet telephony and large-scale interactive simulation systems, have increasingly stringent quality of service (QoS) requirements. Key QoS requirements include network bandwidth and latency, CPU speed, memory access time and power levels. To reduce development cycle-time and cost, such distributed systems are increasingly being developed using multiple layers of COTS hardware, operating systems and middleware components. Historically, however, it has been hard to configure COTS-based systems that can simultaneously satisfy multiple QoS properties, such as security, timeliness and fault tolerance. As developers and integrators continue to master the complexities of providing end-to-end QoS guarantees, it is essential that successful patterns and techniques be reified in DOC middleware to help others configure, monitor and control COTS-based distributed systems that possess a range of interdependent QoS properties [14].

Reflective middleware.. This term describes a collection of technologies designed to manage and control system resources in autonomous distributed application and systems. Reflective middleware techniques enable dynamic changes in application behavior by adapting core software and hardware protocols, policies and mechanisms with or without the knowledge of applications or end-users [6]. As with distributed system QoS, DOC middleware will play a key role in supporting the effective application of reflective middleware-based applications.

Concluding Remarks. Advances in distributed object computing (DOC) technology have occurred at a time when deregulation and global competition are motivating an increase in software productivity and quality. Distributed computing is perceived as a way to meet QoS requirements for dependability and scalability, and to control costs via open systems. Likewise, OO design and programming are widely touted as a means to reduce software cost and improving software quality through reuse, extensibility, and modularity. As a result, there has been a surge of interest in DOC technology in the trade press and in many organizations.

Unfortunately, the level of high quality R&D focus concerning DOC technologies has not kept pace with the level of interest. Since DOC is a combination of two fields, academic journals and conferences concerned with either field have only recently embraced the merger, which has yielded relatively few forums for technical discussion of the combined disciples. Consequently, DOC technology has been "sold" far more than it has been studied systematically. This special issue *Parallel and Distributed Computing Practices* is intended to rectify this imbalance by examining the technical benefits and the challenges provided by DOC technology.

Douglas C. Schmidt
University of California, Irvine

REFERENCES

[1] BEA SYSTEMS, *et al.*, *CORBA Component Model Joint Revised Submission*, Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.
[2] B. W. BOEHM, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, 21 (1988), pp. 61–72.
[3] D. BOX, *Essential COM*, Addison-Wesley, Reading, MA, 1997.
[4] F. P. BROOKS, *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, 20 (1987), pp. 10–19.
[5] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
[6] F. KON, M. ROMAN, P. LIU, J. MAO, T. YAMANE, L. MAGALHAES, AND R. CAMPBELL, *Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB*, in Proceedings of the Middleware 2000 Conference, ACM/IFIP, Apr. 2000.
[7] D. LEA, *Concurrent Java: Design Principles and Patterns, Second Edition*, Addison-Wesley, Reading, MA, 1999.
[8] T. LINDHOLM AND F. YELLIN, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
[9] OBJECT MANAGEMENT GROUP, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.
[10] I. PYARALI, C. O'RYAN, D. C. SCHMIDT, N. WANG, V. KACHROO, AND A. GOKHALE, *Using Principle Patterns to Optimize Real-time ORBs*, Concurrency Magazine, 8 (2000).

[11] D. C. SCHMIDT, M. STAL, H. ROHNERT, AND F. BUSCHMANN, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*, Wiley & Sons, New York, NY, 2000.
[12] D. C. SCHMIDT AND T. SUDA, *An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems*, IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems), 2 (1994), pp. 280–293.
[13] A. WOLLRATH, R. RIGGS, AND J. WALDO, *A Distributed Object Model for the Java System*, USENIX Computing Systems, 9 (1996).
[14] J. A. ZINKY, D. E. BAKKEN, AND R. SCHANTZ, *Architectural Support for Quality of Service for CORBA Objects*, Theory and Practice of Object Systems, 3 (1997).