Guest Editorial

Douglas C. Schmidt
Washington University

Over the past decade, improvements in hardware and networking technology have yielded dramatic increases in network and computer power. For instance, advances in VLSI technology and fiber optics have increased computer processing power by 2–3 orders of magnitude (from 1 MIP VAXes to Alphas with hundreds of MIPs). Likewise, network channel speeds increased by 5-6 orders of magnitude (from 300 baud modems to Gigabit ATM networks).

Despite these advances in performance, however, the effort and cost required to develop, port, and extend software for distributed systems remains remarkably high. Experience over the past decade indicates that distributed systems are hard to implement correctly, efficiently, and robustly. In addition, many existing examples of distributed systems (such as the WWW and the Internet) don't yet support the quality of service required for emerging multimedia services and electronic commerce.

Much of the complexity of building distributed systems stems from the need for both highly flexible *and* highly efficient software. Flexibility is needed to adapt quickly to new application requirements. For instance, medical systems, global trading systems, and digital library applications require increasingly sophisticated support for security, transactions, and synchronized multimedia streams. Efficiency is needed to support the quality of service demands of performance-sensitive applications that possess stringent throughput and delay requirements. For example, applications like digital movie studios, avionics, and teleconferencing require increasingly higher bandwidth, lower latency, and less jitter.

One promising approach for alleviating the limitations of conventional methods is *distributed object computing*. Distributed object computing represents the confluence of two major areas of software technology: distributed computing and object-oriented design and programming. Techniques for developing distributed systems focus on integrating multiple computers to act as a single, scalable computational resource. Likewise, techniques for developing object-oriented systems focus on reducing software complexity by capturing successful design patterns and creating reusable frameworks and components. Thus, distributed object computing is the field dealing with object-oriented systems that can be distributed efficiently and flexibly over multiple computing elements.

This issue of *Computing Systems* contains five papers that represent current leading-edge research on, and experience with developing, distributed object

computing systems. The papers were culled from the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS), which was held in Toronto, Canada on June 17–21, 1996. Like the half-dozen USENIX C++ Conferences from which it evolved, COOTS is dedicated to showcasing advanced R&D work on object-oriented technologies and software systems. The 1996 COOTS technical program covered a range of core object-oriented technologies including frameworks and components, design patterns, C++, Java, and CORBA.

The papers appearing in this issue cover a range of distributed object computing topics. The first two papers focus on the use of Java to develop frameworks for invoking methods on remote objects. The lead article is "A Distributed Object Model for the Java System" by Wollrath, Riggs, and Waldo. This paper describes the Java Remote Method Invocation (RMI) system, which adds distribution to the Java object model. Java RMI is similar to other distributed object computing models, like CORBA and DCOM, in that it supports remote object invocations. Unlike CORBA and DCOM, however, Java RMI is specific to the Java programming language. This allows Java RMI to take advantage of Java features such as garbage collection, downloadable code, and security managers.

The second paper focuses on a specific aspect of Java RMI. "Pickling State in the Java System" by Riggs, Waldo, Wollrath, and Bharat describes the process used in Java RMI to serialize objects for transferring across process boundaries or storage in databases. When a Java object is written to an output stream, the stream first serializes the object, and then traverses the object's references to other objects. The whole graph of objects is thus "pickled" to the stream and can be subsequently "unpickled" to preserve the inter-object references.

Framework support for method invocation is also important for programming parallel object-oriented systems in languages like C++. The third paper on "Smart Messages: An Object-Oriented Communication Mechanism for Parallel Systems" by Arjomandi, O'Farrell, and Wilson describes the Smart Message mechanism in their ABC++ class library. They use smart messages to create and invoke operations on active objects in shared- and distributed-memory parallel computers. Smart messages leverage C++ features like polymorphism and templates to simplify use and ensure type-safe parameter marshalling. In addition, ABC++ combines smart messages and futures to allow asynchronous communications between active objects in a type-safe and portable manner.

Another area of intense R&D in distributed object computing is the Common Object Request Broker Architecture (CORBA). Like Java RMI, CORBA "ORBs" are designed to enhance distributed applications by automating common networking tasks such as object registration, object location, parameter marshalling, framing and error handling, object activation, demultiplexing,

and upcall dispatching. With the advent of high-speed networks and complex distributed applications, it is increasingly important to implement these tasks efficiently and scalably. The remaining papers in this issue focus on these two topics.

Pyarali, Harrison, and Schmidt's paper on the "Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging" describes a framework for efficiently transferring Binary Large OBjects (Blobs) in a distributed medical imaging system. The framework integrates higher-level distributed object computing middleware (such as CORBA) with C++ wrappers for lower-level communication mechanisms (such as sockets over TCP/IP). This integration allows medical imaging applications to operate efficiently on very large Blobs (e.g., images containing dozens of Megabytes), independently of Blob location and Blob type.

Finally, Kordale, Ahamad, and Devarakonda's paper on "Object Caching in a CORBA Compliant System" explores another aspect of CORBA performance—scalability via object caching. Their paper describes the design and implementation of a framework for caching distributed CORBA objects. Applications can use either strong consistency (where changes to shared objects are made visible to all cached copies immediately and in the same order as the changes were made) or causal consistency (where newly cached objects are guaranteed to reflect causally preceding events).

As the papers in this issue of *Computing Systems* attest, distributed object computing is now being applied to develop flexible and efficient distributed systems and applications. Although a great deal of hype has appeared in the commercial software industry and trade press, it's surprisingly hard to find solid technical material on the real strengths and weaknesses of distributed object computing. As a result, distributed object computing is a technology that has been "sold" more than it has been examined empirically. Therefore, those interested in the technology have had few opportunities to evaluate the promise and the challenges that distributed object computing provides. This issue of *Computing Systems* is intended to address this lack. I encourage you to get involved with others working on these topics and to contribute your insights and experience at future USENIX COOTS conferences.

In closing, I'd like to thank the USENIX staff, the COOTS '96 program committee, and the authors and tutorial speakers for making the conference such a success. I'm particularly grateful to Doug Lea, Steve Vinoski, Jim Waldo, and Tim Harrison for helping to select and improve the papers in this issue of *Computing Systems*. In addition, I'd like to extend my appreciation to Peter Salus for expediting the production of this final issue of *Computing Systems*, and for his many years of exemplary service to the USENIX community.