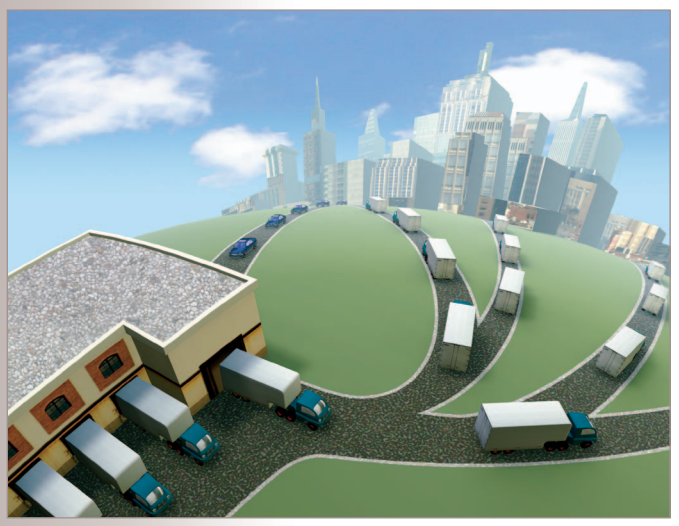

Asynchronous Middleware and Services

Doug Lea
SUNY Oswego

Steve Vinoski
IONA Technologies

Werner Vogels
Amazon.com

The theme features in this issue of *IC* explore how and why asynchronous middleware is playing an increasingly important role in distributed and Web-based systems. Several research and engineering challenges remain before this technology can fully make good on its promises. But before describing them, let's step back a bit and discuss some of the basic ideas behind asynchronous middleware.

A-Synchronicity

Exactly what does the word *asynchronous* mean? The term has come to indicate, among other things, "without time," "no central clock," "nonblocking," and "loosely coupled." Wikipedia defines the term simply as "the state of not being synchronized." Given this definition, we should begin any discussion of asynchronous technologies by first discussing synchronous systems, in two different senses.

The first sense of synchrony involves systems that require access to a clock to coordinate actions. Several networking technologies require communicating parties to be synchronized based on a clock, for example. Although Asynchronous Transfer Mode (ATM) networking brought some of the scalability advantages of packet-switched networks to the lowest level of the networking stack, it didn't remove the need for synchronization between the different components, and thus required the development of additional signaling protocols. As another example, all the way up the stack, many distributed systems algorithms assume access to a clock to help structure the interactions between different parties through synchronization. Distributed systems theoreticians, however, have designed algorithms for asynchronous distributed systems in which time doesn't exist. Under these difficult conditions, they've designed algorithms that work under all circumstances and place minimal requirements on execution environments.

The second, and probably more common, sense of synchrony is applied to program flow. A procedure or system call is considered synchronous if it blocks the caller until the response is ready, and asynchronous if the caller isn't blocked and can separately retrieve the response later.

 Published by the IEEE Computer Society 1089-7801/06/$20.00

In the developer community, the choice between using synchronous and asynchronous programming primitives can often sound like a religious war, with continual debate about which strategy allows for building better-performing systems. We must realize, however, that all computer systems are, in essence, asynchronous and that the notion of a synchronous operation is merely a convenient programming style implemented using asynchronous primitives and operating-system-supported threading. The days in which operating systems didn't support fine-grained concurrency control are long behind us, and we can now build massively concurrent systems when using enterprise-class operating systems. Such systems have fundamental support for asynchronous operations, as well as for synchronous calls in combination with threading.

Programming with synchronous operations has always had a perceived advantage in terms of simplicity. Whereas asynchronous systems often had to deal with complex state management, their synchronous counterparts could rely on simpler program flow and control mechanisms for constructing applications. This simplicity came with a cost, however — the investigation of performance and reliability problems was often overwhelmed by the complexity of interactions among these mechanisms. Asynchronous systems, on the other hand, were frequently constructed as large state machines using an event-programming style for triggering state transitions. Combining this approach with sufficient concurrency to efficiently exploit system resources wasn't always possible. Yet, approaches such as Matt Welsh's staged event-driven architecture (SEDA), which combines the best of event-driven programming, concurrency management, and ease of programming, are now opening up new ways of building systems.

Architectural Styles

The boundaries between asynchronous and synchronous programming styles are rather artificial. What is it that makes middleware synchronous or asynchronous? It's not whether the Remote Procedure Call (RPC) blocks for its return value; that's purely a programming abstraction. The best way to distinguish the two is to look at the preferred way of building reliable systems: using transactions.

Transactions are one of the most powerful paradigms in computer programming. Well understood by many programmers, transactions are simple — they either commit or abort, and if they commit, they provide rock-hard guarantees on the actions inside the transaction. They're so simple to use that if we could, we'd build all our systems with them. Unfortunately, transactions come at a cost that's hard to ignore: the more resources involved in the transaction, and the longer it takes to complete, the higher the chance that it will block other transactions from getting work done. This makes transactions unsuitable for building large-scale operations, especially if they include many distributed resources.

Synchronous middleware aims to provide many of transactions' all-or-none guarantees for certain classes of applications without some of the overhead. Some applications don't require transactional guarantees, however, or can't be served by synchronous middleware because of scale and distribution limitations. Asynchronous middleware addresses such applications, in which asynchrony means that we don't try to achieve transactional-style synchronization between the participants.

The choice between using synchronous and asynchronous programming primitives can often sound like a religious war.

The challenge for middleware based on asynchronous principles is to provide alternatives for the powerful combination of atomicity, consistency, isolation, and durability that transactional systems provide. To build their systems, application architects need primitives that are as powerful as transactions, given that large distributed applications' requirements are just as complex as those we can construct with transactions.

Most asynchronous middleware uses the explicit notion of a message as its fundamental building block, which often leads to applications that are structured as series of message exchanges. These messages carry application state, updates to state, Web service requests, event notifications, and so on. Two main categories of middleware use these explicit asynchronous messaging systems:

- *Message queues*. Although the term "message-oriented middleware" is currently more popular, this technology has its roots in the

message-queuing techniques developed for mainframe access, which itself was asynchronous in nature. These days, message-oriented systems don't necessarily advertise themselves as queuing systems, but they do provide similar guarantees. Hosts can send and deliver messages with various reliability guarantees, including (transactional) persistence, routing via several parties, security properties, and so on. This makes messaging a very versatile basic primitive with which to build systems.

- *Publish–subscribe*. Pub–sub adds two important concepts to basic messaging: conceptual naming and infrastructure-independent routing. How we route and deliver messages no longer depends on the physical infrastructure but on concepts such as named channels or topics, behind which one or more recipients might be hiding. Systems might conduct additional message filtering on the basis of message content. In pub–sub systems, message delivery depends fully on the actions of the receivers, which frequently are unknown to the senders.

Middleware aficionados realize that, in the end, it's not about the technology.

Event-oriented technologies represent a third form of asynchronous middleware geared toward more lightweight sensor-controller-actuator patterns used in areas from factory-floor systems to enterprise application integration to mobile applications. In essence, event-based systems are built out of messaging and publish–subscribe components, and are an application of the two areas described earlier.

Messaging technologies, however, form only a starting point. They let architects build applications out of simple but powerful techniques, but this is hardly sufficient for building real, complex applications. The true power comes from using these techniques in higher-level structuring engines such as business-process execution, long-running business activities with compensating actions, or persistent conversations. The basic technologies have matured, and research has started addressing the more complex interactions that are necessary to make large-scale distributed application architectures successful.

Middleware aficionados realize that, in the end, it's not about the technology but about the applications you need to build with it. Real-world distributed applications are complex in functionality, often incorporating many different technologies that must work together, and having strict requirements in terms of performance, reliability, security, and efficiency. Given that no single technology can solve all the requirements, such technologies must be able to coexist peacefully — for instance, enabling architects to integrate interfaces to a new payment system into their applications without too much pain.

In this Issue

This issue's articles span the range of recent work in asynchronous middleware systems. The first two focus on the description, representation, and analysis of asynchronous systems. Asynchronous messages are wonderfully flexible and composable, but at the potential expense of unpredictability. In "Analyzing Conversations of Web Services," Tevfik Bultan, Xiang Fu, and Jianwen Su present a graphical model for representing *conversations* spanning multiple asynchronous messages among participants. Not only do these models help make protocols more humanly understandable, they also set the stage for powerful analytic tools that can help answer questions about how systems will operate. In "Asynchronous Messaging between Web Services Using SSDL," Savas Parastatidis, Jim Webber, Simon Woodman, Dean Kuo, and Paul Greenfield describe the SOAP Services Description Language, an XML-based approach to describing asynchronous protocols that enables SOAP-based systems to represent and rely on modular contracts that capture their interaction responsibilities.

The next two articles delve into pub–sub systems. In "Publish–Subscribe for High-Performance Computing," Greg Eisenhauer, Fabián Bustamante, and Karsten Schwan show how basic pub–sub designs can be tuned to provide competitive alternatives to other approaches to high-performance cluster and grid computing. And in "Publish–Subscribe Grows Up: Support for Management, Visibility Control, and Heterogeneity," Ludger Fiege, Mariano Cilia, Gero Mühl, and Alejandro Buchmann present a new approach to scalability and manageability concerns, based on scoping constructs that extend those found in other aspects of computing.

Finally, in "Asynchronous Mediation for Integrating Business and Operational Processes," Philippe Lalanda, Luc Bellissard, and Roland Balter describe the design and implementation of a framework that allows programmers of asynchronous systems to use analogs of interception techniques often employed in synchronous systems to add capabilities to existing services.

It's clear that synchronous systems, especially those based on RPC abstractions coupled with popular programming languages such as Java and C++, have provided the basis for many important advances made in distributed systems development and middleware over the past 15 to 20 years. It's just as clear that as our applications scale up to the Internet, World Wide Web, and beyond, the known limitations in synchronous systems, especially in coupling and scalability, will increasingly become liabilities that outweigh synchronous abstractions' benefits. We might very well be witnessing a significant, fundamental shift in how distributed systems and middleware-based applications are designed and built, with asynchronous approaches becoming a truly basic and fundamental assumption at the heart of such applications.

Doug Lea is a professor of computer science at the State University of New York, Oswego. He has a BA, an MA, and a PhD from the University of New Hampshire. Lea has written several widely used software utility packages in C, C++, and Java. Contact him at dl@cs.oswego.edu.

Steve Vinoski is chief engineer for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski has helped develop middleware standards for the Object Management Group (OMG) and the World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.

Werner Vogels is the director of systems research at Amazon.com. Previously, he was a research scientist at Cornell University, studying scalability and reliability of mission-critical enterprise systems. Vogels has a PhD in computer science from the Vrije Universiteit in Amsterdam. Contact him at werner@vogels.net.