From the Editor in Chief...

Middleware's Shrinking Middle

Doug Lea • *State University of New York, Oswego*

Bob Filman is taking a break from writing his column, and asked me to play guest columnist.

During the early days (late 1980s and early 1990s) of what's come to be called object-oriented middleware, researchers and developers shared the vision that distributed-object systems should look something like the present-day Corba, Java RMI, COM, and .NET. These frameworks consist of tools for defining service objects that clients can use across a network via "remote" references. Clients access services within distributed applications using a variant of method calls that communicate data by sending and receiving "marshaled" copies of nonremote objects as message arguments. Basic framework support includes registries that track remote objects' locations, proxies that intercept outgoing calls and transform them to use custom network remote procedure call (RPC) protocols, and listeners that receive incoming calls, dispatch them to their target service objects, and relay results.

It's a pretty good story, but it hasn't worked out according to plan. Experience has shown that the gap is too big between middleware frameworks' two main usage modes:

- *external* communication among hosts to support services over the open Internet, and
- *internal* communication among processes on the same machine or within local clusters running under common administrative domains.

These different usage modes often coexist in the same programs. Many systems employ externally inaccessible distributed objects to support integrated applications featuring externally accessible objects that are ready to serve unknown requesters with particular services. However, programs rarely employ objects that mix both internal and external modes.

Most middleware frameworks try to cope with these differences by specially "optimizing" internal communication (streamlining marshaling for local invocations, for example) and "pessimizing" external communication (such as interposing container objects to perform more elaborate security checks). Over time, however, it's become clear that these usage modes' underlying programming models are so dissimilar that it does more harm than good to treat them as variants along a continuum. This has led to searches for alternatives for supporting both external and internal modes.

External Communication

Alternatives for external communication have received the most attention over the past five years or so, mainly under the heading of service-oriented architectures (SOAs). The old-guard distributed-object system camp tends to think of SOA as a gratuitous reinvention of everything that's been done before (five generations of overhauls, if you include early RPC and Distributed Computing Environment [DCE] frameworks). SOA can even appear to be a step backward in that implementations tend to be noticeably slower than mature distributed-object frameworks. However, such dismissals ignore the inescapable pressures that led to its development: firewalls and related security concerns argue for adopting common, basic Web protocols (such as HTTP) rather than custom RPC protocols (such as IIOP). Similarly, XML's rise in various other contexts makes it the only reasonable candidate for representing data used in distributed messaging rather than relying on custom language- or framework-specific data marshaling. Furthermore, the statelessness of Web-based communication argues for abstracting functionality into services performed by anonymous objects assuming appropriate roles, rather than identifiable stateful objects. Finally, the SOA approach appears far more amenable to the scripted orchestration and choreography that's increasingly required to coordinate loosely coupled services, and increasingly desired

 Published by the IEEE Computer Society 1089-7801/06/$20.00 © 2006 IEEE

for Asynchronous JavaScript and XML (AJAX), mash-ups, and the like.

Internal Communication

Researchers and developers have also explored alternatives on the internal front, but these haven't converged as rapidly as SOA. This will likely change as clusters (and clusters of multicores, in particular) become the most common substrate for application servers and other large applications. Perhaps the simplest-sounding approach is distributed shared memory (DSM), in which programs executing across different machines in a LAN or different processes on a single machine "transparently" share objects in the same way that different threads in a process do. The underlying mechanics are usually based on caching protocols that track reads and writes of pages of memory containing the shared objects. This tends to be more efficient than programmer-directed marshaling. Although successful in several experimental research systems, this approach hasn't widely caught on, in part because most DSM systems ignore a useful software engineering aspect of RPC-like systems: for the sake of fault containment, manageability, and program modularity, most objects should be directly accessible only from their containing processes.

A different approach to programming clusters has roots in the high-performance computing (HPC) world. Frameworks such as the message-passing interface (MPI) supply simple but fast communication primitives to explicitly transmit data across processes. This results in problems opposite to those in DSM — including the inability to use shared names for shared objects — and requirements that programmers manually arrive at efficient solutions for load-balancing, fault-tolerance, and moving data where it's needed. In non-HPC systems, various custom clustering frameworks tend to take either a shared-memory or message-passing approach and then partially address shortcomings. Differences among such coexisting yet nearly redundant frameworks contribute to large systems' complexity and management difficulties.

The leading contender for converging internal middleware frameworks in the same sense that SOA did for external frameworks is the Partitioned Global Address Space (PGAS) approach, which unifies aspects of both shared memory and message passing. As with DSM, all processes in a cluster can reference all objects. Here, however, only the containing process can access a given object's fields or invoke its methods. Other processes must explicitly ask owners to perform such operations. The owners can, in turn, disallow such operations, depending on programmer-defined access policies. For ease of use, languages and frameworks might automatically translate certain accesses as asynchronous or synchronous requests to the owning processes. Among language-level efforts, the most prominent example of this approach is IBM's in-development language, X10, which is geared mainly toward HPC applications. Some variant of this line of attack seems likely to prevail in upcoming languages and frameworks.

The rise of SOA and PGAS doesn't quite spell the end of Corba, RMI, and other classic distributed-object systems, but it does increasingly restrict their ranges of use. These technologies will surely still play major roles in enterprise-integration systems and heterogeneous intranet applications and niches such as real-time Corba, as well as the countless existing deployments that will remain with us for years. They might even re-emerge as the approach of choice in future grid computing frameworks that must sometimes straddle administrative domains. But the idea that one approach to object-oriented middleware could serve all needs seems to be one whose time has come and gone — which is to say that it never really came at all.

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. He is Associate Editor in Chief for *IEEE Internet Computing*. Contact him at dl@cs.oswego.edu.