The Trouble with Distributed Objects

Daniel F. Savarese

Last Updated: 2012-05-23

Note

This column was published originally in the August 2003 issue of *Java Pro* under the same title. It was written when service-oriented architecture (SOA) hype was getting started and published when the hype was in full swing. As a result, the column generated criticism from people who were buying into the vendor-generated hype. Programmers who had worked with distributed programming models extensively "got it." Brief discussions in the straight_talking_java[2] and the service-orientated-architecture[3] Yahoo groups show the disparity of reaction. The critics must have skipped over or ignored key sentences in the column that invalidate their criticisms. The original column was a required reading assignment[4] for the 2005 Distributed Systems course (CS404) at Purdue University Calumet.

My most general contention was that the industry was focusing on the wrong problems and the new wave of standards was doing nothing to simplify the construction of distributed systems. I'm comfortable saying that time has proven that SOA standards touted at the time didn't deliver on their promises. Also, the large-scale distributed computing infrastructures that emerged at companies such as Amazon, Google, and Facebook have spurred the development of distributed software programming systems that address some of the challenges posed in this column.

Changes. I've resisted the temptation to edit the article to clarify points and make it more relevant to today. As a result, I've made only minor text edits.

Why Objects?

The benefits of object-oriented programming do not translate from shared-memory programming to distributed programming. It's time we face up to this problem and look for ways to resolve it. Web services may facilitate integration and interoperability, but they don't do much to let you specialize component behavior to meet application-specific requirements. If you work with distributed object or service component frameworks on a daily basis, you may feel that it is much too hard to build systems that do exactly what you want done and do it exactly how you want it done. I have felt this way for years. Let me explain why.

Object-oriented programming delivers data encapsulation, inheritance, and polymorphism, which promote program understanding and code reuse through code modularization, data type specialization, and generic programming. The byproduct is easier-to-maintain software.

Data encapsulation hides information from the programmer, exposing only the operations that can be performed on data. This design principle encourages the modular organization of code, where related data and operations are grouped in a single source-code file. The unit of modularity in Java is the class.

Inheritance allows programmers to customize classes to suit their needs without writing redundant code. Those desired aspects of a class are retained while new behaviors are introduced by adding new methods or overriding existing ones. Java implements data type specialization through single-class inheritance.

Polymorphism allows objects of different types to be operated on without regard to their type. Code reuse is promoted because a single method can be implemented for use with multiple data types. Java supports polymorphism through interface definition and implementation. A method that operates on an interface will work with any object that implements that interface. Polymorphism in Java does not extend to primitive types. You can't imple-

[1] https://www.savarese.org/
[2] http://tech.groups.yahoo.com/group/straight_talking_java/messages/31855?threaded=1&m=e&var=1&tidx=1
[3] http://tech.groups.yahoo.com/group/service-orientated-architecture/messages/573?threaded=1&m=e&var=1&tidx=1
[4] http://cs.purduecal.edu/~rlkraft/cs404-2005/class.html

ment a single method that will sort an array of ints as well as an array of floats without resorting to dynamic type identification and casting or reflection.[5]

Object-oriented programming aims to ease the implementation, understanding, and maintenance of programs through language-based program organization and definition techniques. How well do these techniques translate to distributed objects?

At first glance, data encapsulation appears to work rather well in a distributed environment. Operations are grouped by the data or resources they operate on. In addition to relieving the programmer from having to know the structure of data, by grouping operations with data, you avoid having to move the data around the network. In general, distributed resources are shared, causing operations to be modularized into service components, such as Web services. So far so good.

Well, not exactly. In a distributed environment, data encapsulation conflicts with the need to specialize data type behavior. When you use a third-party class library, you can tailor behavior by deriving new classes with inheritance and overriding methods or by applying aggregation and wrapping a class with an adapter. Inheritance is not an option with distributed objects. Sure, CORBA, RMI, and the like allow the object, component, or service developer to use inheritance. But once a component is deployed, they do not allow an application developer using the remotely situated component to specialize its behavior with inheritance. If you don't have access to both the source code and the deployment host, you're out of luck. Aggregation doesn't work well either, because each delegated method call crosses a network boundary. You're denied direct access to state variables, forcing you to use expensive accessor methods.

Polymorphism is achievable in some sense, but isn't as useful as one would expect. Remote objects and services may present compatible interfaces to the world, but without inheritance, this feature does not achieve the same effect as interfaces, abstract classes, and virtual methods. At best, you get interchangeable components. Most distributed object systems rely on interface definition languages (IDL) and automatically generated stubs to access remote objects. Therefore, even if two objects have matching methods with identical signatures, the ability of your code to use them interchangeably is at the mercy of the stub generator.

In the case of Java RMI, remote objects are invariably forced to extend `java.rmi.UnicastRemoteObject` or `java.rmi.activation.Activatable`. Therefore, any polymorphism depends on all distributed parties agreeing on the same Java interface and for client code stubs derived from the interface implementations to be made available to all parties. This tight coupling makes RMI more of a client/server implementation system than a truly distributed computing implementation system.

Services. Not Objects.

If the benefits of object-oriented programming don't evidence themselves in distributed computing, then what's the right approach? You're going to hear a lot of talk about service-oriented architectures (SOA) this year, if you haven't already. Some Web services development vendors have come around to understanding that Web services don't work well as objects. So they're promoting the design of coarse-grained, loosely coupled services interconnected by asynchronous communication. You're also going to hear about event-driven architectures (EDA), which are the translation of event-based programming to a distributed context. The idea is that SOA is better suited to implementing real-time business processes, and EDA is better suited for long-running asynchronous business processes. Unfortunately, there's nothing new here.

If services are so great, why do they look so much like objects? If Web services are most effective when not treated as objects, why did WSDL turn into the umpteenth coming of IDL? For that matter, why have distributed objects always looked more like services than objects? It's really just a semantic game. Even EDA is a bit of a farce because event-based programming is isomorphic to message-based programming, which is equivalent to remote procedure calls. EDA only gets interesting when events also provide the code to process the event—in other words, self-servicing messages.[6]

The crux of the problem with distributed programming is that we're using the equivalent of a distributed assembly language to build distributed software. Compilers have no knowledge of distribution. We specify interfaces statically with interface definition languages and can sometimes discover and invoke those interfaces dynamically, but with a lot of difficulty. Services are deployed statically and cannot be adapted by applications. In 1988,

[5]This ceased to be true—at least in part—after the addition of autoboxing. The extra work doesn't really go away; it's done for you by the compiler. Also, generic programming via autoboxing in Java yields poor performance compared to equivalent template-based C++ code.

[6]I was doing experimental research at the time with what I called self-servicing messages. The self-indulgent reference was meant to spark the reader's imagination.

NeXT Computer proclaimed the next step in computing was the object and later offered the world Portable Distributed Objects (PDO). After 15 years and the reinvention of countless distributed object frameworks, it's clear there are more steps to be taken.

How do we translate the benefits of object-oriented programming to distributed systems? I don't profess to offer a grand solution, but I'm pretty sure we have to move past the idea of distributing objects. Services are useful, but let's call them services and not objects. A *service* is a package of functionality that can be shared concurrently by multiple applications. *Objects* have additional properties that are just not evidenced in a distributed context.

Beyond the Status Quo

The ultimate benefit we're looking for is to make distributed software development easier. An obstacle to that goal is that distributed services don't allow applications to adapt service behavior to meet application-specific requirements. The object-oriented approach of inheritance, coupled with reuse-enabling polymorphism, doesn't work.

Either we have to throw services out the window or we have to look for ways to enable services to be customized by applications. Services appear to be a convenient building block for distributed systems, so let's not throw them out just yet. Even after developing ways to customize services in application-specific ways, we'll still have an assembly language of mechanisms. To finish the job, we'll have to add distribution abstractions to programming languages and design compilers that generate code using the new mechanisms. Still, the generated code must be dynamically reconfigurable. You should not have to recompile an application to redistribute its elements across a set of hosts. Orchestration server vendors will tell you that business process modeling languages are the answer. Business process modeling languages, however, are not suitable for general-purpose programming.

Distributed programming presents several problems that today are left up to programmers to resolve for themselves. The question of optimizing the frequency of communication, the size of messages, and the distribution of computation is ever present. So is the question of how to tie together different elements of a distributed application and allow new elements to be inserted and existing ones removed. The research community has developed a number of module interconnection languages for this purpose. Service orchestration languages provide some of the same functions. Yet they do not address the problem of mapping an interconnection of services to a physical configuration of hosts to meet a specific set of criteria. Services are almost always taken to be statically placed. Perhaps the most pressing question is the one I have talked about the most so far: How do we allow new application-specific behavior to be applied dynamically to distributed components?

Until all of these questions are resolved in a unified manner, distributed programming will remain difficult—or, at best, cumbersome. Let's not remain satisfied with the status quo.