Developing any nontrivial middleware-based distributed system is hard, regardless of the middleware technology. Although technology’s continuous march should make systems easier to develop, somehow the details always remain challenging. New technologies only seem to make simple things simpler; they don’t help with the more complex things. Horizontal concerns, such as security, transactions, fault tolerance, load balancing, and enterprise management, never seem to get easier, regardless of what middleware development platform you choose.

The complexity in deploying a production-quality middleware system dwarfs its development difficulty. Along with the application, a lot of moving parts come into play in a production deployment: network and computing hardware, operating systems, and all the software the application depends on, including databases, management systems, security systems, runtime libraries, and virtual machines, each of which must be proven production-worthy. Production testing involves many hours of verification and burn in, and the problems that invariably crop up during these test periods typically are hard to reproduce and complex to solve.

Of course, a system’s care and feeding does not end at deployment. If applications were static, maintenance efforts primarily could focus on tracking down and fixing intermittent bugs that appeared only after days or weeks of continuous operation. Unfortunately, middleware applications rarely are static. Not only do changing business requirements impact the applications themselves, but for a variety of reasons, the various platforms, operating systems, and middleware underneath the applications also change. Dealing with versioning and change management in a deployed middleware application can be complicated and costly.

Version Control

Versioning a stand-alone monolithic application usually is straightforward; you ship a new one, and the biggest problem you might have to worry about is backward compatibility with data read or written by the previous version. Unfortunately, it’s not so simple with middleware applications, which typically involve libraries and distribution. These libraries are dynamic link libraries (DLLs), shared libraries, and Java archive (JAR) files, which are physically separate from the application itself. Distribution means that middleware applications typically consist of numerous smaller applications or components that work together over a network.

A good example of versioning coming into play for distributed middleware systems is the interface between any two applications or components. For a client or sender to interact over a network with a server or receiver, the former must know the message syntax that the latter expects. The network message that the sending application creates typically indicates the target operation or service and includes data the receiver expects. If the receiving application were to change its interface so its operations or services changed, sending applications that based their messages on previous interface versions could very well send messages or requests that no longer conformed to the expected syntax.

Arguments abound about whether distributed systems should have explicit interface definitions, with some claiming that explicit definitions result in tight coupling, where dependencies between two or more application components are so great that no component can be maintained or modified without also requiring changes to the others. It’s true that wherever tight coupling exists in a deployed application, versioning issues invariably crop up, but explicit interface definitions by themselves are not the primary cause of tight coupling. Such arguments are misguided. Interface versioning issues apply whether the middleware explicitly employs an interface definition language (IDL), as in distributed-object systems like Corba and Microsoft COM, or whether “interfaces” really are represented as exchanged documents, as in many messag-
Distributed-Object Versioning

Let’s consider modifying an already-deployed Corba interface. If you change one interface operation’s name, all applications using that interface must be recompiled with the new interface definition. Otherwise, clients based on the original interface will send the old operation name in their requests, and the server will reject them. Similarly, if you change an operation parameter type definition, then those clients will marshal data for a parameter that doesn’t match the data expected by the revised server. The server also will likely reject this request.

Corba assumes that sender and receiver have the same understanding of their marshaled request data; thus, Corba requests contain no type information. This means that some data type changes could find the server accepting malformed requests and attempting operations on the incorrect data they contain. Changing an operation’s signature also is problematic because data marshaled by a sender wouldn’t match what the receiver is expecting.

These problems imply that if you want to modify a Corba server without affecting clients, you should never modify its operations’ names or signatures or its operation parameter types. So, what changes can you make to an interface without adversely affecting existing client applications?

Adding an operation generally is acceptable because Corba operations are identified by name in marshaled requests, unlike other systems in which numbers identify operations (as offsets from the beginning of the interface definition). The IDL compiler guarantees operation name uniqueness because it won’t let you overload interface operation names. Thus, clients aware of the new operation can send requests for it, but not knowing about it generally won’t break existing clients. Adding an operation can be harmful, however, if the operation introduces state manipulations on which other existing operations have been modified to rely. Changes to implementations of existing operations always must be backward compatible so existing clients can rely on semantics equivalent to those provided by the previous versions.

Other additions are problematic, though. Adding an exception to the list of exceptions that an existing operation can raise is a no-no: it could cause an existing client to receive an unexpected exception. However, removing an exception from the list is okay because existing clients that know about that particular exception never will receive it.

Some practitioners dislike the idea of changing interfaces altogether. For example, if your interface is used locally within an application, such as across the boundary of a shared library or DLL, then adding operations to an interface could break existing code, depending on which programming language is used. In C++, for instance, adding an operation to an object in a shared library or DLL could break an existing application using that library because the addition could change the layout of the object’s virtual table. Java binary compatibility is not as tricky or difficult as in C++, mainly because the language specification defines precisely what it means. Even so, changing published interfaces used locally within a single application and as distributed interfaces between applications is inherently difficult.

Because of local–remote transparency problems that can arise from changing distributable interfaces, some practitioners suggest that you should avoid ever modifying an interface once it’s been deployed. Instead, you should add operations and types by deriving a new interface from the existing one. You then could declare all the new types and operations in the new derived interface. Changing a Corba object’s interface to one that’s more derived will not break existing clients because, by definition, the object still supports the original interface.

Using interface inheritance for versioning can work, but only in limited cases. In this case, inheritance uses — or, perhaps more accurately, abuses — a type classification mechanism as a versioning mechanism, and it can get confusing once multiple versions are required. Microsoft COM, which uses virtual tables as a fundamental function-dispatching mechanism, recommends using inheritance for versioning because doing so essentially extends the virtual table with the new operations supplied by the derived class. However, unlike its COM predecessor, Microsoft .NET does not recommend using inheritance in this fashion. Instead, .NET includes explicit versioning support. Specifically, .NET assemblies — collections of modules and resources that make up a single unit of deployment — include versioning information in their identifiers, and the .NET runtime ensures that only the correct assembly versions load for each application.

The .NET versioning mechanisms have not yet been widely proven through years of development, but given that they’re based on lessons learned from COM, they likely will work quite well for versioning real-world applications. Note, though, that while these mechanisms ensure version compatibility within a single application, they do not address the distributed application interface versioning problem. And unlike .NET, Corba supplies no versioning support whatsoever. In general, it’s important when developing Corba or .NET distributed systems to know the rules about what changes maintain backward compatibility. Depending on your application, it might make sense to combine Corba with other technologies such as XML to help with versioning, or use an entirely different distributed-object system.

Messaging Versioning

Messaging systems typically do not have, or need, a distributed-object sys-
dleware, so I’ll focus on a common pop-
mats available in message-oriented mid-
the variations of message types and for-
versioning issues apply only to messages
object systems. Because the messaging
systems are less-tightly coupled than distributed-
son that messaging systems generally
interface variation is the principal rea-
with variable messages. The lack of
systems tend to have fixed interfaces
face and message levels, messaging
tion and versioning occurs at the inter-
tributed-object systems in which varia-
arily break existing applications.
include the producer's name wouldn't
separate elements for artist, album title,
description of a music CD might include
necessarily break all applications using
existing definition without automati-
means that you can add them to an
application using them must be
album producer is subsequently added
and parse them using such general
more closely into the Java language,
thus trading away flexibility for pro-
language-specific and, thus, do not take
native type system or libraries. Rather
having to read XML messages and
XJAX approach tries to fit
more closely into the Java language,
trading away flexibility for pro-
grame versioning and stronger
typing (and presumably fewer pro-
programming errors as a result). Unfortu-
nately, this type of mapping eliminates
XML's flexibility.
In XML, the convention is to
ignore unknown elements, but with
programming languages, the norm is
to abort if the application doesn’t
understand something. If the music CD
XML definition previously described is
mapped into a Java class that gets
compiled into one or more applica-
tions, and the new element for the
album producer is subsequently added
to the XML definition, then those Java
classes must be regenerated, and each
application using them must be
recompiled, retested, and redeployed.
There is no practical way around this.
If the applications instead use the
more general SAX or DOM approach-
es, then the code to read and manipu-
late the music CD XML definitions is
much longer, more awkward, and
potentially slower than that required to
deal with the JAXB classes. However,
the approach also is more flexible and
— if coded properly to ignore unknown
optional elements — can easily result in
an application being able to silently
handle the album-producer element or
any other optional element or attribute
added to the revised XML definition.

**Versioning Approaches**

This column barely scratches the sur-
faced of the versioning problem. How-
ever, the major issues I touched on
give a good indication of the state of the
middleware versioning problem,
which can be summed up as follows:
basically, you’re on your own. Most
middleware does little to nothing to
help applications with versioning
problems. Even worse, sometimes
middleware vendors directly con-
tribute to the problem by failing to
properly version their own platforms.
As a result, application versioning
tends to revolve around conventions
and best practices learned along the
way. I wish I had better news, but how
you deal with versioning depends
heavily on each application.

**References**

1. B. Joy et al., The Java Language Specifica-
tion, 2nd ed., Addison-Wesley, 2000; http://
java.sun.com/docs/books/jls/.
2. D. Watkins, M. Hammond, and B. Abrams,
Programming in the .NET Environment,
3. D.C. Schmidt and S. Vinoski, “Corba and
XML, Part 1: Versioning,” C/C++ Users J.,
vol. 19, no. 5, 2001; www.cuj.com/documents/
ss-7995/cujcep1905vinoski/.
4. M. Henning, “A New Approach to Object-
Oriented Middleware,” IEEE Internet Com-

**Steve Vinoski** is chief engineer of product inno-
vation for IONA Technologies. He’s been
involved in middleware for 15 years. Vinos-
ki is the coauthor of Advanced Corba Pro-
gramming with C++ (Addison Wesley Long-
man, 1999), and he has helped develop
middleware standards for the OMG and W3C.