

Services-based enterprise integration patterns made easy, Part 2: More on the evolution of basic concepts

Skill Level: Intermediate

[Dr. Waseem A. Roshen, PhD \(waroshen@us.ibm.com\)](mailto:waroshen@us.ibm.com)

IT Architect

IBM

06 Mar 2008

This installment, Part 2 of the [series](#), picks up where you left off in Part 1. Now that you've learned about the two earliest integration patterns—data sharing (socket programming) and remote procedure call (RPC)—you continue developing the basic concepts. Check out two more developed patterns: distributed objects and asynchronous messaging. Explore the concepts of language independence, declaration of service interfaces, rudimentary ideas of publication and discovery of services, and basics of the enterprise service bus (ESB).

Introduction

[Part 1](#) and Part 2 of the series trace the evolution of enterprise integration patterns, introducing several basic concepts and features involved in Service-Oriented Architecture (SOA)-based integration patterns. Part 1 covers two early patterns—data sharing (socket programming and RPC—which segues into an exploration of the service provider and service consumer, platform independence, and connectivity.

To improve on RPC functionality, let's take a look at two methods:

- **Distributed objects, also known as the Object Request Broker (ORB):** This approach focuses on code reuse and language independence.

- **Asynchronous messaging:** This approach addresses the problem of tight coupling between applications.

Let's take a look at the distributed objects approach first, because it's more closely aligned with RPC. Today, most application servers are based on ORB technology.

Distributed objects: the Object Request Broker

There are three main types of implementation of distributed objects technology. One of them is language independent and platform independent, and is known as Common Object Request Broker Architecture (CORBA). The other technologies are either language dependent or platform and language dependent. The Java Remote Method Invocation (RMI) is an example of a language-dependent technology, while the Microsoft Distributed Object Component Model (DCOM) and the IBM® System Object Model (SOM) are examples of platform-dependent technologies.

Let's look at CORBA in some detail, because it's the most general (language- and platform-independent) technology, and because products based on this technology from different vendors can work together. For example, IBM WebSphere® Application Server, which is based on ORB, can communicate with many other vendors' application servers.

In addition to introducing the benefits of object orientation, such as inheritance, polymorphism, and encapsulation, CORBA introduced a number of new features. Probably the most important was the concept of *ORB*, which extracted the code for marshaling input and output arguments and the code for communication from the client and server applications into a separate software component. In addition, ORB provides a facility to get a reference to a remote object so that methods can be invoked on that remote object.

This separation let the same code be reused by many applications and allowed a certain amount of decoupling between the applications by moving away from point-to-point integration. This move away from point-to-point integration may be considered the first step in the evolution of the concept of ESB. This is illustrated in Figure 1, which shows that multiple applications on the same machine can use the same ORB to communicate with each other and with applications on different machines.

Figure 1. Multiple applications communicating through ORBs, illustrating code reuse and indirect communication through ORBs (a first step in the direction of ESB)

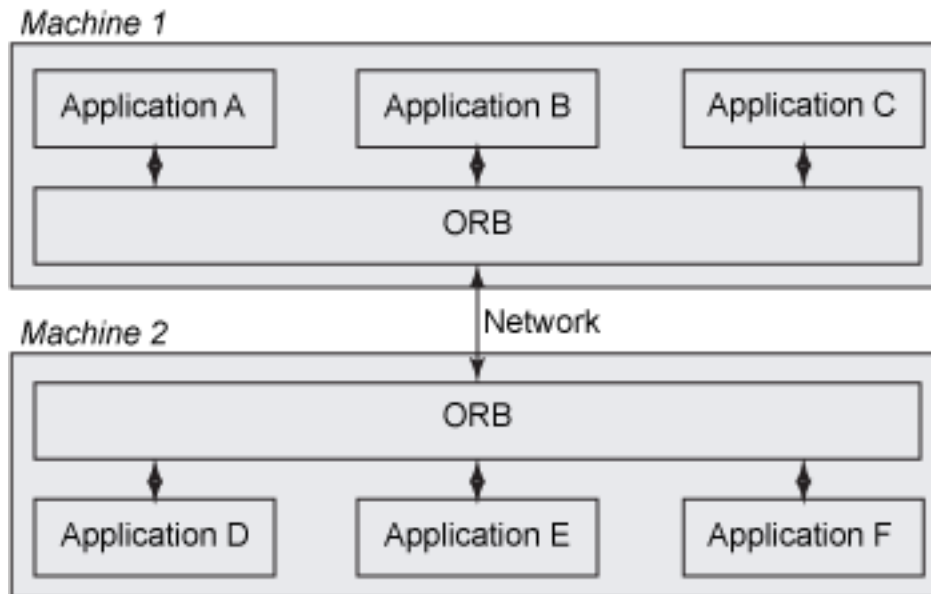
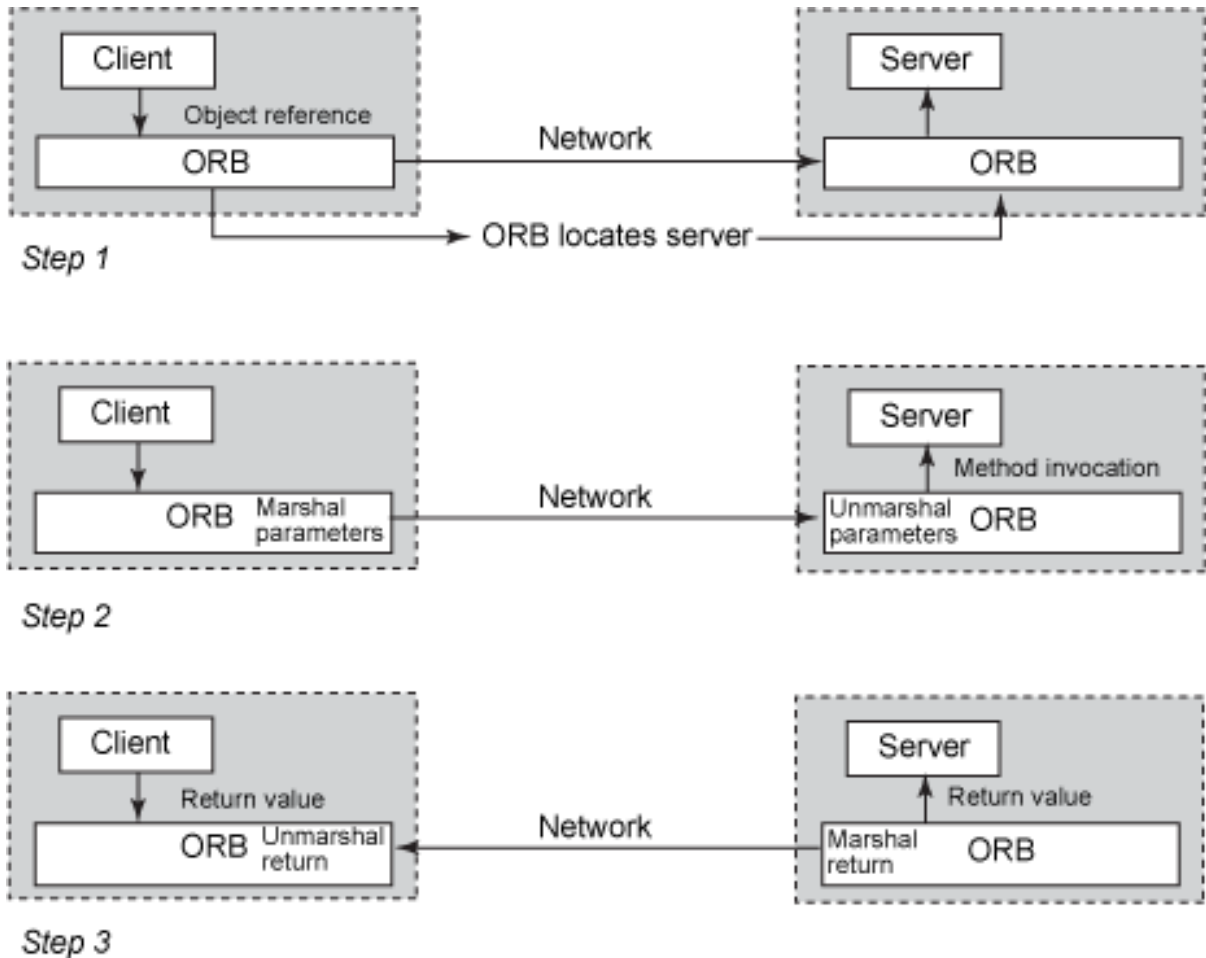


Figure 2 shows the basic working of an ORB. When an application wants to use the services of another component, it first gets an object reference for the object providing the service. After the object reference is obtained, the client application can call methods on that object as if that object was a local one.

Figure 2. Method invocation on a remote object using an ORB, including getting the remote object reference



CORBA also introduced the concept of language independence of the interface definition. This was done through the introduction of *Interface Definition Language (IDL)*, which is analogous to header files in C++ programming. It only defines the interface, but doesn't contain the implementation. IDL is responsible for ensuring that data is properly exchanged between dissimilar languages and is, thus, responsible for the language independence of CORBA. This allows a client to be implemented in one language (such as C++), while implementing the server in another language (such as Java). Listing 1 shows an example of IDL.

Listing 1. An example IDL interface definition, which defines a single interface with a single remote operation for calculating the square of a number

```

module Test {
  interface square
  attribute double arg1;
  double getSquare (in
double arg1);
};
};

```

Another important concept that was initiated in CORBA was that of the naming service, which allowed for the registration of CORBA objects and the fact that they can be located by name. This concept contained the seed for the registry concepts in SOA.

In summary, CORBA introduced a number of new features and allowed for the reuse of communication code and the marshaling and unmarshaling of code. The concept of the registration and location of remote objects, language-independent interface definition, and the move away from point-to-point integration were the significant new features introduced. Therefore, for many integration projects, an ORB-based solution might be the appropriate choice. However, there are some disadvantages to using ORB-based integration, so it might not be the perfect choice in some situations. Some of these considerations include:

- An ORB-based solution isn't scalable to high volumes, so it's not suitable when you expect high volumes of transactions. This lack of scalability is due to the synchronous nature of the interaction, which blocks the client application from proceeding further with its work until it receives the response from the server.
- The interaction between the client object and the server object is too fine grained and results in many trips across the network. Thus, the network bandwidth isn't used efficiently, which further limits the scalability of the solution.
- ORB-based communication isn't reliable, and there's no guarantee that the messages and return values will be delivered to the intended targets. Thus, the client application might experience a hangup in its operation under certain circumstances, such as a break in the network connection.
- Although CORBA, in principle, is language independent, most of the commercial products that employ ORBs are language specific, such as Java or Java 2 Platform, Enterprise Edition (J2EE). This is because the CORBA open standards proved to be too difficult to implement in their most general form. This limits the integration capabilities of ORBs to the applications that are written in those specific languages.

Asynchronous messaging

To deal with these problems, a parallel development has occurred, which is based on asynchronous messaging and contains the seeds for the development of another type of ESB. This type of ESB provides a more scalable solution than the ESB type based on ORB. In *asynchronous messaging*, the client or client object sends a message to the target application, but doesn't wait for the response to continue its work. This leads to a certain amount of decoupling between the applications involved. Thus asynchronous messaging may be employed as the integration basis

if high transaction volumes are expected.

In messaging, the applications don't communicate with each other directly and don't have a dedicated communication link established between them. Instead, they communicate indirectly through queues, as shown in Figure 3. Application A sends a message to the queue. Application B retrieves the message from the queue after it has been delivered by application A. However, the communication can still be point to point if there's a dedicated queue for each receiving application. In asynchronous messaging there's an option called *publish and subscribe* in which multiple applications can receive the same message. However, many times that's not enough, because one application needs more complicated message routing. For example, a message might need to be routed based on the content or the size of the message. In such cases, in addition to the messaging software, the middleware must also include a *message router*, often called a *message broker*. A central message broker can receive messages from different applications, determine the correct destination for each message type, and route the message to the appropriate destination application. This lets applications communicate with each other without knowing the location of the receiving applications. This is shown in Figure 4, which also indicates that messaging along with the message broker can form the backbone of an ESB.

Figure 3. Messaging using queues

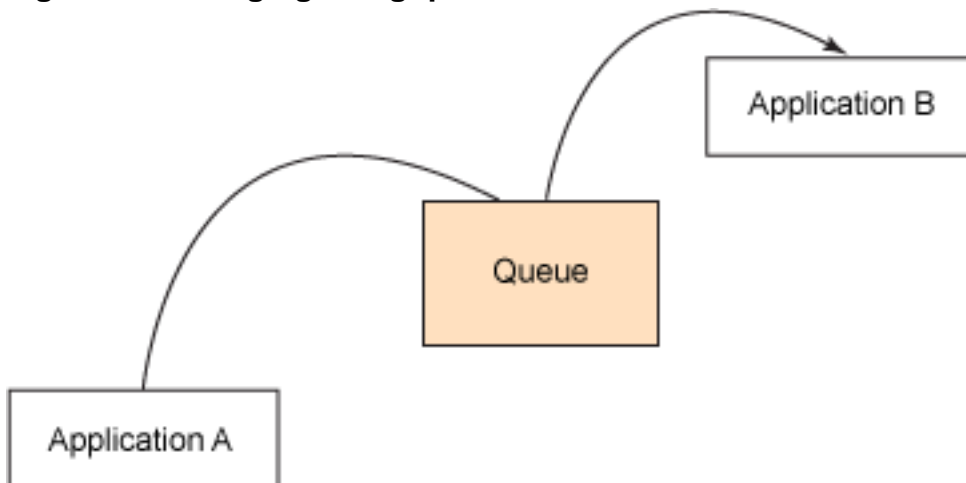
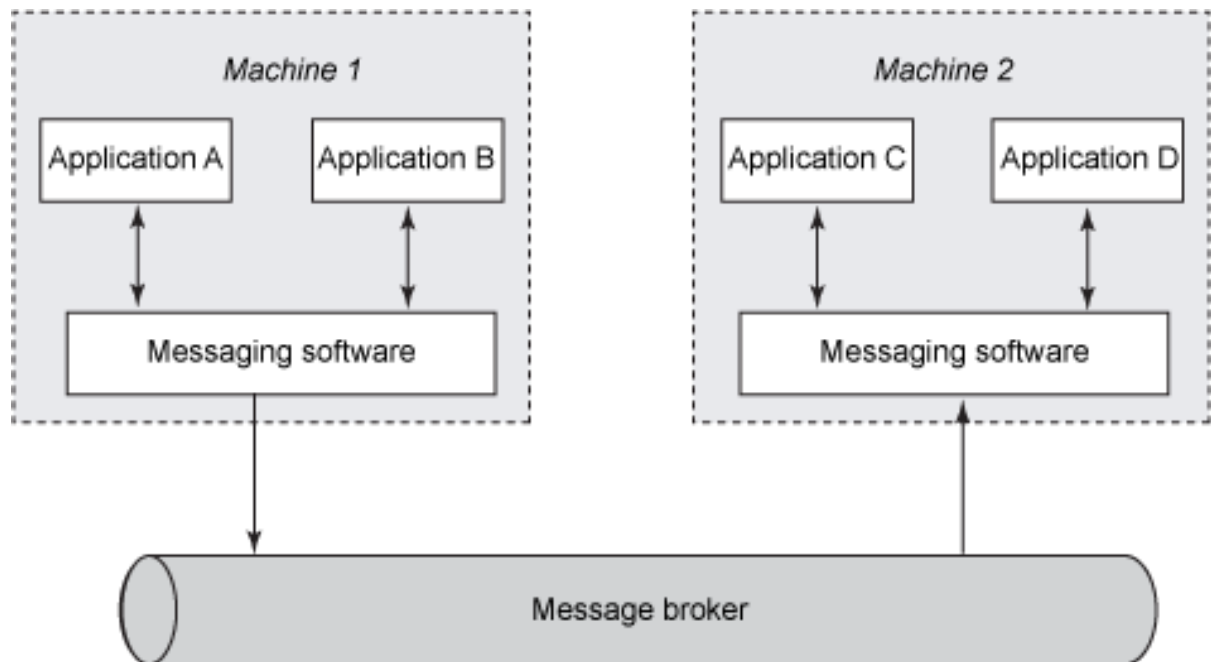


Figure 4. Multiple applications using the messaging software and a central message broker (router) component to communicate with each other and with applications on other machines connected by a network

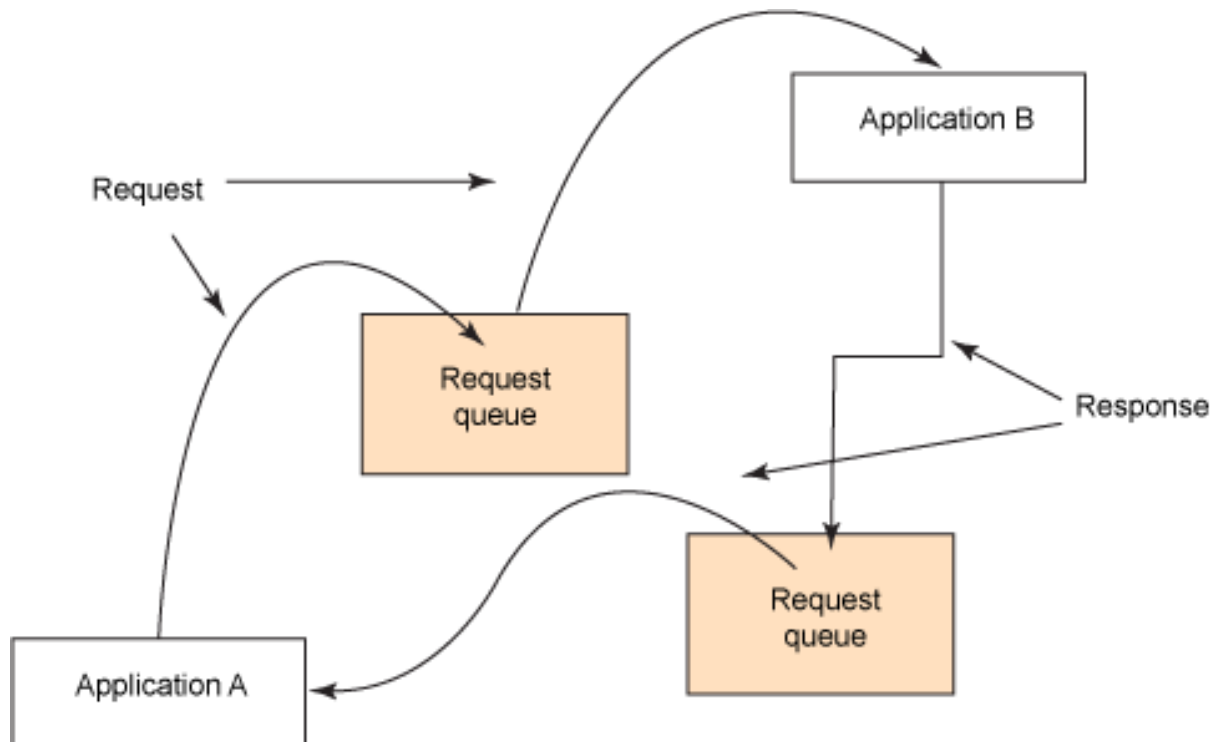


Another advantage of synchronous messaging is that it's possible to guarantee delivery of the message. You do this by persisting the message on both sides of the network connecting the two applications. This ensures that the message is delivered even in the event of a temporary network breakdown or if the receiving application isn't running at the same time as the sending application. Such a guarantee is not possible with RPC or using ORBs.

Yet another advantage of using messaging middleware, such as IBM WebSphere MQ, is that a large set of data can be exchanged and transported across the network, resulting in coarse-grained data transfer. This leads to a more efficient use of the network bandwidth.

Although asynchronous messaging is, in principle, a one-way communication, you can make it invoke some functionality in the receiving application. An example of such invocation of functionality in the receiving application is message-driven beans (MDBs). MDBs and similar software pieces don't have return values. And it's possible to use asynchronous messaging to simulate synchronous messaging using two queues. Figure 5 shows this where one queue, the request queue, is used to deliver the request, while the return values are obtained through another queue. The request queue is the output queue for the requesting application (application A); at the same time it serves as the input queue for the receiving application (application B). Similarly, the response queue is used as an output queue for application B and as an input queue for the return value for application A.

Figure 5. Simulated synchronous messaging using messaging software



Among the options described so far, asynchronous messaging may be the most powerful way for the applications to share data and functionality when large transaction volumes are involved. However, it's not suitable for all situations; you must make proper tradeoffs to arrive at a solution for your situation. There are some disadvantages for asynchronous messaging:

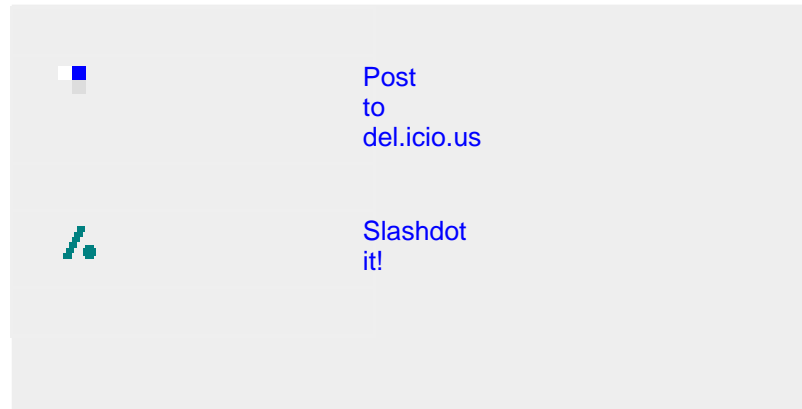
- Generally, asynchronous messaging software is costly, with the dollar amount of ESB based on asynchronous messaging middleware, sometimes more than an order of magnitude higher than the cost of an ORB-based middleware ESB.
- There's a learning curve associated with an asynchronous messaging environment.
- There's a certain amount of overhead and bookkeeping involved in simulating a synchronous interaction between two applications.

Conclusion

Share this...



Digg
this
story



[Part 1](#) and [Part 2](#) of this [series](#) described the basic concepts essential for understanding a services-based integration pattern. These concepts include loose coupling, code reuse and layering, language and platform independence, language independent interface, the idea of discovering a remote object at run time, invoking methods remotely, and asynchronous messaging for scalability. The next two installments, [Part 3](#) and [Part 4](#), explore how these concepts are used and further developed to become service-oriented integration patterns. For example, the idea of definition and discovery develops into the concept of SOA registry, while the concepts of ORB and asynchronous messaging form the core of the ESB pattern.

Resources

Learn

- Learn more about SOA architecture in the book [Enterprise SOA: Service-oriented Architecture Best Practices](#).
- Find out more about ORB and CORBA in [Distributed Architectures with CORBA](#) and on the [CORBA Web site](#).
- Check out [Enterprise JMS Programming](#).
- Read [Enterprise Integration Patterns](#), an excellent resource for integration patterns.
- The [SOA and Web services zone](#) on IBM developerWorks hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.
- Play in the [IBM SOA Sandbox!](#) Increase your SOA skills through practical, hands-on experience with the IBM SOA entry points.
- The [IBM SOA Web site](#) offers an overview of SOA and how IBM can help you get there.
- Stay current with [developerWorks technical events and webcasts](#).
- Browse for books on these and other technical topics at the [Safari bookstore](#).
- Check out a quick [Web services on demand demo](#).
- Get an [RSS feed for this series](#). (Find out more about [RSS](#).)

Get products and technologies

- Innovate your next development project with [IBM trial software](#), available for download or on DVD.

Discuss

- [Participate in the discussion forum for this content](#).
- Get involved in the developerWorks community by participating in [developerWorks blogs](#), including the following SOA and Web services-related blogs:
 - [Service Oriented Architecture -- Off the Record](#) with Sandy Carter
 - [Best Practices in Service-Oriented Architecture](#) with Ali Arsanjani
 - [WebSphere SOA and J2EE in Practice](#) with Bobby Woolf
 - [Building SOA applications with patterns](#) with Dr. Eoin Lane

- [Client Insights, Concerns and Perspectives on SOA](#) with Kerrie Holley
- [Service-Oriented Architecture and Business-Level Tooling](#) with Simon Johnston
- [SOA, ESB and Beyond](#) with Sanjay Bose

About the author

Dr. Waseem A. Roshen, PhD

Dr. Waseem Roshen is an IT architect in the Enterprise Architecture and Technology Center of Excellence of IBM Global Business Services in Columbus, Ohio. He works on enterprise architecture and integration. He's also a Sun Certified J2EE Architect, has published 60 articles, and has worked on 24 patents.

Trademarks

IBM, the IBM logo, and WebSphere are registered trademarks of IBM in the United States, other countries or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.