Understanding how EJB calls operate in WebSphere Application Server V6.1

John Pape
Dr. Mahesh Rathi

July 30, 2008

Many applications use Enterprise JavaBeans™ (EJB) deployed in an IBM® WebSphere ® Application Server EJB container. Communication to these EJB components requires the use of concepts like Internet Inter-ORB Protocol (IIOP) communication, JNDI look-ups, and workload management. This interaction can be complex and is often a black box for many WebSphere users. This article provides a basic understanding of how EJB communication works in the context of WebSphere Application Server. Communication into and out of the ORB is addressed, along with the definitions of basic call patterns and components involved in making calls to EJB components deployed in the EJB container. A basic understanding of J2EE™ principles and general administrative knowledge of WebSphere Application Server is assumed.

Introduction

For many WebSphere users, EJB communications are a black box. That is, we understand that calling method 'X' will return the desired result from the EJB business logic. Under normal operating conditions, this knowledge may be sufficient as it is not usually necessary to understand the inner workings of the ORB. However, when performance problems or load balancing issues arise, in-depth knowledge of the ORB can greatly enhance the the problem determination effort and reduce time to resolution. As with any WebSphere component, understanding the mechanism of operation for the ORB and its related communication protocols can yield further expertise when troubleshooting it.

The ORB

The ORB is the central part of CORBA, or Common Object Request Broker Architecture, which aims to make remote objects usable like local objects in object-oriented, distributed applications. In addition, ORB lets these objects communicate with each other regardless of the specific platforms, techniques, and languages that are used to implement those objects.

CORBA is an open, vendor-neutral specification for distributed computing created by the Object Management Group (OMG), a consortium of over 600 software vendors and other companies.
promoting open, standard interfaces. CORBA and its Interface Definition Language (IDL) provides a common format for representing an object that can be distributed to applications written in Java™, C, C++, and other languages and running on different operating systems and machines.

The ORB (Object Request Broker) plays an integral part in the application server runtime. The ORB is implemented as part of the Java Virtual Machine (JVM). IBM WebSphere Application Server includes additional code (generally referred to as the extension ORB) that manages the ORB instance, and also provides the plug-in framework for object services (such as security). This IBM ORB implementation is not the same as the Sun™ ORB; the code base is different altogether. It is the foundation on which the EJB container depends to broker communication for Java Remote Method Invocation (RMI) interactions. RMI is Java's traditional method of performing remote communications and uses a non-standardized protocol, named Java Remote Method Protocol (JRMP), to communicate between Java objects. At the most basic level, RMI is just Java's object-oriented version of Remote Procedure Calls (RPCs). WebSphere Application Server uses the ORB to facilitate client/server RMI communication, as well as to communicate between components.

The ORB manages inbound and outbound requests for remote Java objects, like EJB components. ORB provides a framework for clients to locate EJBs on the server and invoke operations on those as if they were local. (This concept is referred to as local-remote transparency.) The relationship of the ORB with the EJB container can be described with Figure 1.

**Figure 1. ORB and the EJB container**

![Image of ORB and EJB container](image)

**Inter-ORB communication**

Communication between ORBs is accomplished via the IIOP, which is a concrete implementation of the General Inter-Orb Protocol (GIOP) abstract reference specification. The Interoperable Object Reference (IOR) is simply a string representation of the CORBA or RMI-IIOP object reference in a format that ORBs can understand. You can think of this as a normal Java object reference used in everyday coding. IIOP maps GIOP to the TCP/IP communication stack.

The GIOP protocol specifies several different types of messages, such as request messages, reply messages, locate request messages, locate reply messages, and fragment messages. The request and reply message types are self-explanatory, as they deal with the request and response.
interaction between client and server applications. The locate request and fragment message types might not be as familiar to you unless you have needed to read through GIOP tracing to solve a problem:

- The locate request message type is used to verify that the ORB understands and houses the requested remote object. If the ORB does not know the object requested in the locate request message, it will attempt to provide details (for example, an OBJECT_FORWARD which points to a server that does house the object) on where the client can find the requested object.
- The fragment message is a continuation of the previous message delivered to the ORB.

Processing large GIOP messages can cause problems for the ORB. GIOP messages can be fragmented into manageable pieces for easier and faster processing. Custom properties can be set in WebSphere Application Server that specify the exact size in which to fragment a message (or whether or not to fragment messages at all). By default, WebSphere Application Server will fragment GIOP messages at 1024 bytes.

In the most basic of interaction scenarios, the client and the server ORBs communicate directly with each other to satisfy application requests for remote resources. However, in more complex interactions, such as clustered servers with workload management features, processes other than the client and server can become involved (for the purpose of this article) in the NodeAgent process.

Each application server contains an ORB that listens for incoming requests. The ORB has three or more ports that are associated with it: the ORB listener port, the bootstrap port, and the security ports. The ORB listener port is the actual port that the ORB listens on for incoming IIOP requests if ORB security is not enabled. The bootstrap port is always an active, well-advertised port, and always returns an indirect IOR containing the secured and unsecured listener ports. A well-constructed client should always bootstrap to the server in which the remote resource resides (or to a multiple host provider URL that points to the cluster if the resource resides in a cluster). There are three other ports associated with the ORB and security: the CSIv2 Client Authentication SSL port, the CSIv2 SSL port, and the SAS SSL port. (Details of these ports is beyond the scope of this article and will not be covered here.) ORB listener ports are, by default, dynamically determined from the platform’s ephemeral port range at run time. However, these ports can be set to a specific static port for various reasons, such as firewall rules, business policies, and so on.

When an ORB client requests a reference to a remote object, the remote ORB will reply with an Interoperable Object Reference (IOR), which is simply a string representation of the CORBA or RMI-IIOP object reference in a format that ORBs can understand, and at the very basic level contains host and port information for the server on which the object can be found. The IOR can be thought of as a typical Java object reference used in everyday coding.

**How is the NodeAgent involved?**

First, the NodeAgent process is only involved in an ORB transaction in the WebSphere Application Server Network Deployment product; it is not present in the base application server offering. The NodeAgent process houses a service called the location service daemon, which enables
application servers to register their services and remote objects by providing the daemon with
direct IORs pointing to the said application server. Later, when a locate request comes into the
location service daemon, it uses the registration information to provide a direct IOR back to a
client that points to the registered application server’s ORB listener ports. This registration/IOR
information could be thought of as a sort of routing table that is used by the daemon to locate EJBs
and other resources found on the various application servers managed by a particular NodeAgent.

Before a managed application server can start properly, a NodeAgent must be available locally
on the node. It is possible to configure the application servers such that the dependency on the
NodeAgent process (actually the location service daemon service) is gone. However, this setup
involves carrying out static routing for IIOP requests, and thus WebSphere Application Server’s
workload management services would be nullified. This static routing setup should be avoided
unless truly required. (Consult IBM Support before making this assessment on your own.)

In the event that the NodeAgent process dies or must be taken down or restarted during the
operation of an application server on the same node with EJBs deployed on it, exceptions might
be thrown in the server logs once the NodeAgent is unavailable. These messages are due to the
fact that, once the NodeAgent is removed from the process, the IIOP communication is altered in a
way that prevents both ORBs (client and server) from communicating correctly. However, as long
as clients already have direct IORs to their desired servers, have already created their EJBHome
object, and are making remote calls to that EJB, the NodeAgent is unlikely to be involved in the
IIOP communication after that point.

Ideally, incoming locate requests to the location service daemon for EJB-related objects will not
come in until all the application servers on a particular node have completed their registration
with the location service daemon on the NodeAgent. When locate requests come in for objects
on servers that have not yet registered or are temporarily down (and hence no longer registered),
a CORBA.OBJECT_NOT_EXIST exception will be returned to the client. In this case, the status
of the application server needs to be checked to ensure they are all up and running and have
registered with the location service daemon.

**Understanding the ORB call pattern**

In order for high level application functionality (like EJB components) to work, inter-ORB
communication must occur. There are two basic scenarios that we can observe:

- A standalone application server (WebSphere Application Server, not Network Deployment)
- A clustered application server in a WebSphere Application Server Network Deployment
  environment.

The next sections describe these scenarios.

**WebSphere Application Server base instance**

In this scenario, the EJB component and its client are contained in the separate application
servers. The client code to make the call might look like this:
Listing 1

```java
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL,"corbaloc::isador:2810");
Context ctx = new InitialContext(env);
TestEJBHome home = (TestEJBHome)
PortableRemoteObject.narrow(ctx.lookup("ejb/ejbs/TestEJBHome"),
TestEJBHome.class);
TestEJB bean = home.create();
```

The code in bold text are the lines that result in remote ORB calls. When put into motion, you can follow the events in Figure 2, where the events are listed, in sequence, from top to bottom.

**Figure 2. ORB event calls in base instance**

![Diagram of ORB event calls in base instance](image)

Figure 2 is divided into three parts, each part corresponding to a portion of the Java code in Listing 1:

- The top section contains the events that result from the InitialContext creation on the client.
- The middle section contains the events that result from the InitialContext.lookup() method.
- The bottom section contains the events that result from the creation of the EJB instance.

From this diagram, you can see that the process of obtaining an initial context from the server involves sending a locate request message to the ORB (client and server ORB are the same here) on the ORB’s bootstrap port. The resulting reply message contains a direct IOR (with host and TCP/IP and SSL port information) that points the client to the server containing the requested object. Next, the client issues another locate request message, but this time to the ORB’s listener port. After this, the client requests some information from the ORB to establish an initial context.
After the InitialContext value has been obtained, the client code then does a lookup in the JNDI namespace to determine how to get a reference to the desired remote object (WLMTestEJB). To accomplish this, the client issues yet another locate request aimed at the ORB’s listener port. The ORB replies back to the client via a locate reply message stating that the object is indeed contained locally. In response to the locate reply, the client then requests that the ORB resolve the object and return a reference to the client. Under the covers, this entails the ORB using a tie class to connect the real Java object with the request that came to the ORB. When the client has the reference, it uses a “narrow” call to marshal the generic object returned from the ORB into the desired object type.

The last thing the code does is create an instance of the remote EJB. To complete the process, the client issues a locate request to the remote ORB, to which the ORB replies with a locate reply stating that the object is present. Finally, the client then invokes the create() method on the remote object. When the process is complete in the ORB, the ORB marshals the response back to the client. All data that goes into or leaves the ORB (like method parameters and method return values) are serialized and deserialized using a format called common data representation (CDR). As such, all values passed into the ORB must implement the serializable interface.

The locate request messages used in this base application server scenario are simple locate requests that are processed by the ORB itself. This is different from the next scenario because the location service daemon plug-in (attached to the ORB instance) handles messages responding to the locate requests.

**WebSphere Application Server Network Deployment clustered instance**

In this scenario, the major difference is that the client ORB and server ORB will almost always be two different JVM instances, often located on completely separate hosts. The process will appear much the same, but you will notice the involvement of the location service daemon server contained in the NodeAgent. Listing 2 shows the code involved to make the call.

```java
Listing 2
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "corbaloc::boris:9811,:natasha:9812");
Context ctx = new InitialContext(env);
TestEJBHome home = (TestEJBHome)
PortableRemoteObject.narrow(ctx.lookup("ejb/ejbs/TestEJBHome"),
TestEJBHome.class);
TestEJB bean = home.create();
```

The code only changed from the previous example in only one place: the values placed in the PROVIDER_URL key of the environment Hashtable object. In this case, the developer included multiple corbaloc URLs, each of which points to a separate server’s bootstrap port. This change introduces some fault tolerance and failover to the process as the code now has two paths from which the InitialContext object can be obtained. As in the previous scenario, the code in bold text is responsible for generating the remote ORB requests. Figure 3 illustrates the sequence of events for this scenario, formatted the same as the previous example.
For the creation of the InitialContext object in the top section, there are no obvious differences between this scenario and the first one. However, with the addition of WebSphere Application Server workload management in the picture, there are slight differences.

First, during the creation of the InitialContext (top section), a list of IORs that point to each of the cell’s location service daemon clusters are prepared and returned to the client ORB during the getProperties() portion of the conversation. When the client ORB makes further calls to the server ORB, the workload management plug-in on the client side iterates over the list and issues a resolve_complete_info request. This first request is routed to the location service daemon service in the NodeAgent (shown as the first call in the middle section), where the desired cluster data is sent back to the client in the form of a direct IOR pointing to the destination server.

In the bottom section of the diagram, you can see that the locate request message issued in the first scenario is not needed and the create() method is invoked instead, immediately without having to process the additional locate request because workload management has already “located” an appropriate application server.

**Summary**

This article defined the ORB and how it provides the application server runtime with vital functionality needed to service clients making remote requests to an EJB server. The high level sequence of events that make up inter-ORB communications was also explained, along with the role that the NodeAgent plays in the part of ORB communication between clients and servers.

There is much more information available on these topics. If you wish to continue your reading, you might want to look into these subjects:

- GIOP protocol
• Problem determination for the ORB/EJB container
• RMI over IIOP
• Java Naming and Directory Interface (JNDI)
• EJB clustering
• WebSphere workload management (WLM)
• CORBA and the OMG.

Acknowledgements

The authors extend their thanks to Paul Bullis and Claudia Barrett for providing guidance and clarification on the topics discussed in this article.
Related topics

- CORBA Basics
- How the ORB works
- Wikipedia: General Inter-ORB Protocol
- Wikipedia: Java remote method invocation
- Wikipedia: Object request broker
- Wikipedia: Java Serialization
- Redpaper: WebSphere Application Server V6.1: Workload Management Problem Determination
- Webcast: Workload Management Overview and Problem Determination
- Webcast: Object Request Broker (ORB) Problem Determination and Best Practices

© Copyright IBM Corporation 2008

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)