Distributed Systems Lecture 14

Professor Louise E. Moser

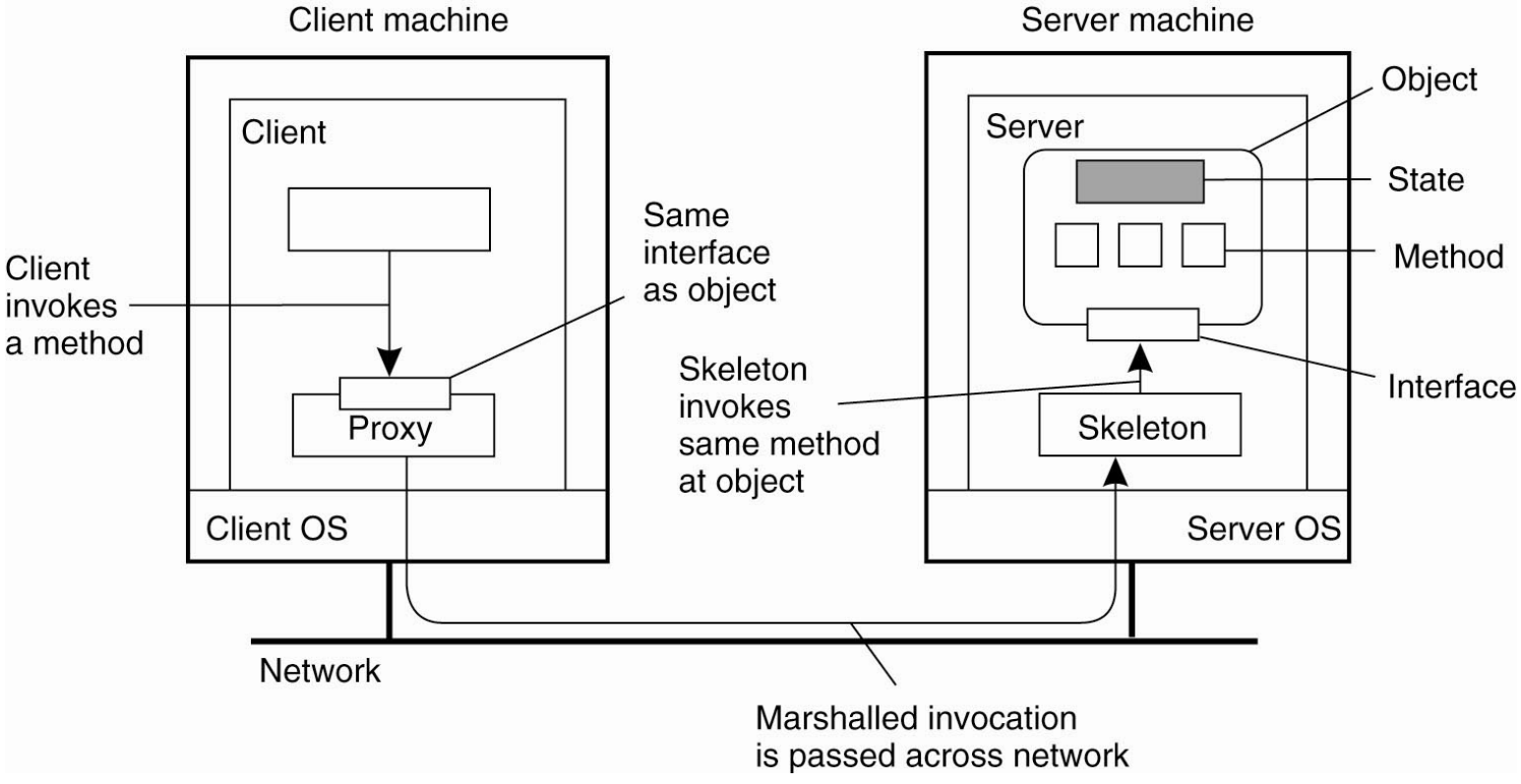
Spring 2014

Distributed Object-Based Systems

- Common Organization of a Client Object and a Remote Server Object
- Java Distributed Object Technology
  - **Java Remote Method Invocation (RMI)**
- Common Object Request Broker Architecture (CORBA) supports C, C++, Java, etc.
- Distributed Component Object Model (DCOM) supports C#
- Java Distributed Object Technology
  - **Java Enterprise Java Beans (EJBs),** based on CORBA

Distributed Object-Based Systems

- Common organization of a client object and a remote server object with a client-side **proxy** and a server-side **skeleton**



Java Remote Method Invocation

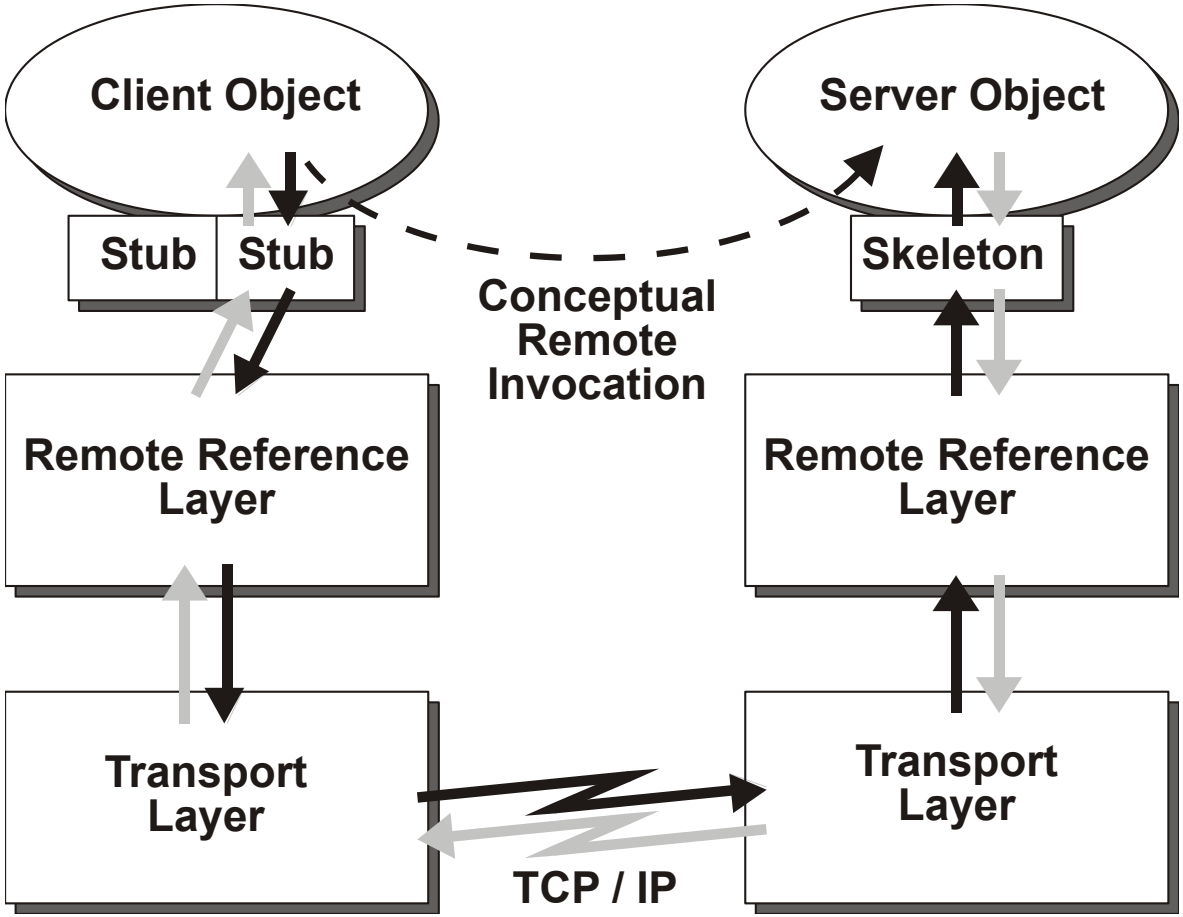
- **Java Remote Method Invocation (RMI)** allows objects to invoke methods of remote objects, located on other computers, as if they were local
- The method invocation is packaged into a TCP/IP message and sent to the other computer, where the method of the remote object is invoked
- The result comes back in another TCP/IP message
- Java RMI is important because
  - **Method invocation is easier to program than message passing**
  - **Method invocation is uniform for both local and remote objects**
  - **The Java program is determined by its logic, rather than by the allocation of objects to processors**
  - **Java RMI is like RPC for CORBA, DCOM, Web Services, etc.**

At a High Level How Does Java RMI Work?

- The client object invokes a method of a remote object, as if it were local, but there is no such local method for it to invoke
- Instead, there is a **stub** that the client object invokes
  - **The stub has the same interface as the remote object**
- When the stub is invoked, it passes the call (method name and parameters) to the **Remote Reference Layer (RRL)** middleware
- The RRL packages the call into a message, and transmits the message to the remote computer via TCP/IP
- At the remote computer, the RRL unpacks the message and passes the call to a **skeleton** that invokes the method of the server
- The results go back to the client in the reverse order, i.e.,

  server to **skeleton** to **RRL** to **TCP/IP** to **RRL** to **stub** to **client**

A Client Object Invoking a Remote Method of a Server Object Using RMI



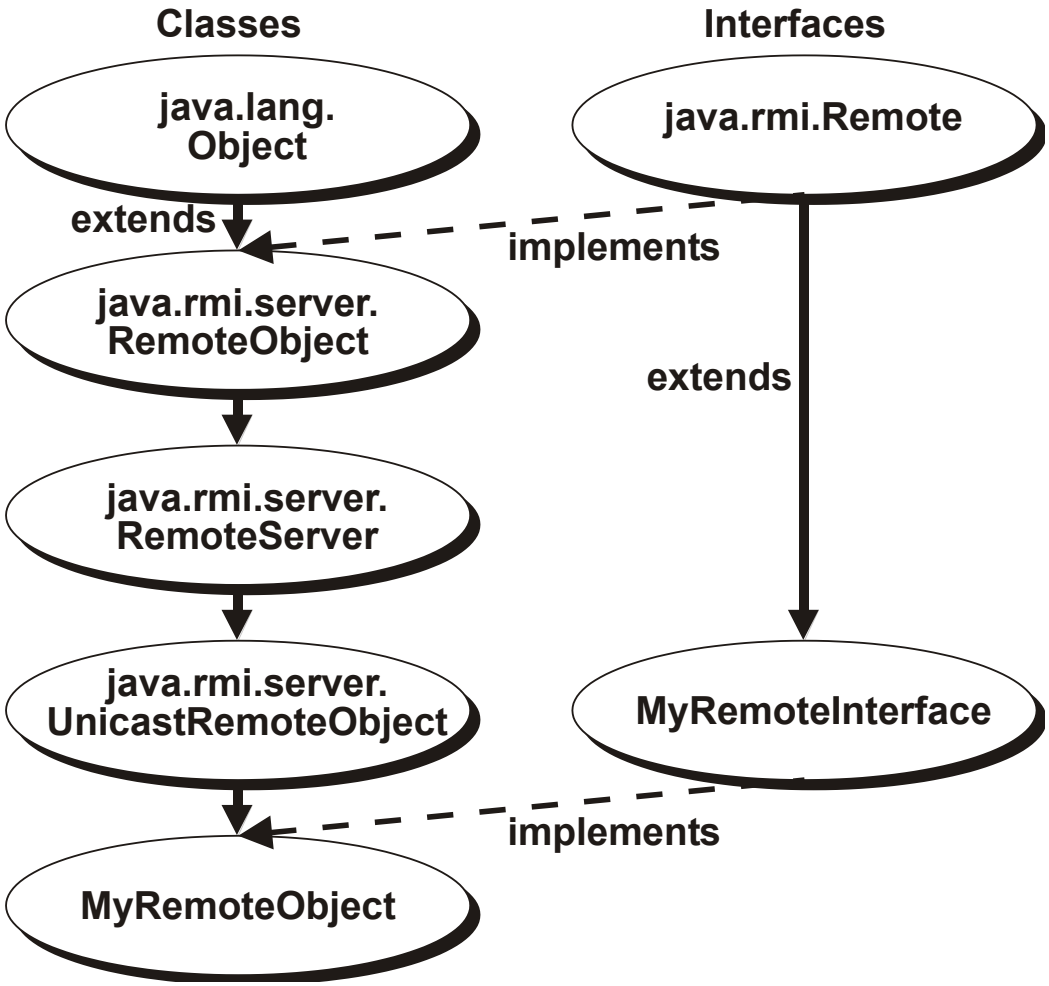
How Does Java RMI Really Work?

- There are lots of questions that must be answered before we can understand how Java RMI really works
  - **How do we make a server object invocable remotely?**
  - **How does a client object find a remote server object?**
  - **How does a client object get a stub?**
  - **How does RMI pass parameters and results?**
  - **How does RMI handle exceptions?**

How Do We Make a Server Object Invocable Remotely?

- The server object must extend the class *java.rmi.server.UnicastRemoteObject*
- The methods that are invoked remotely must have their signatures defined in an interface that extends the interface *java.rmi.Remote*
- The server object must be registered with
  - **The Java RMI Registry, or**
  - **A private registry maintained by the application**
- These registries (typically) run on the same machine as the server object

The Java RMI Class/Interface Hierarchy



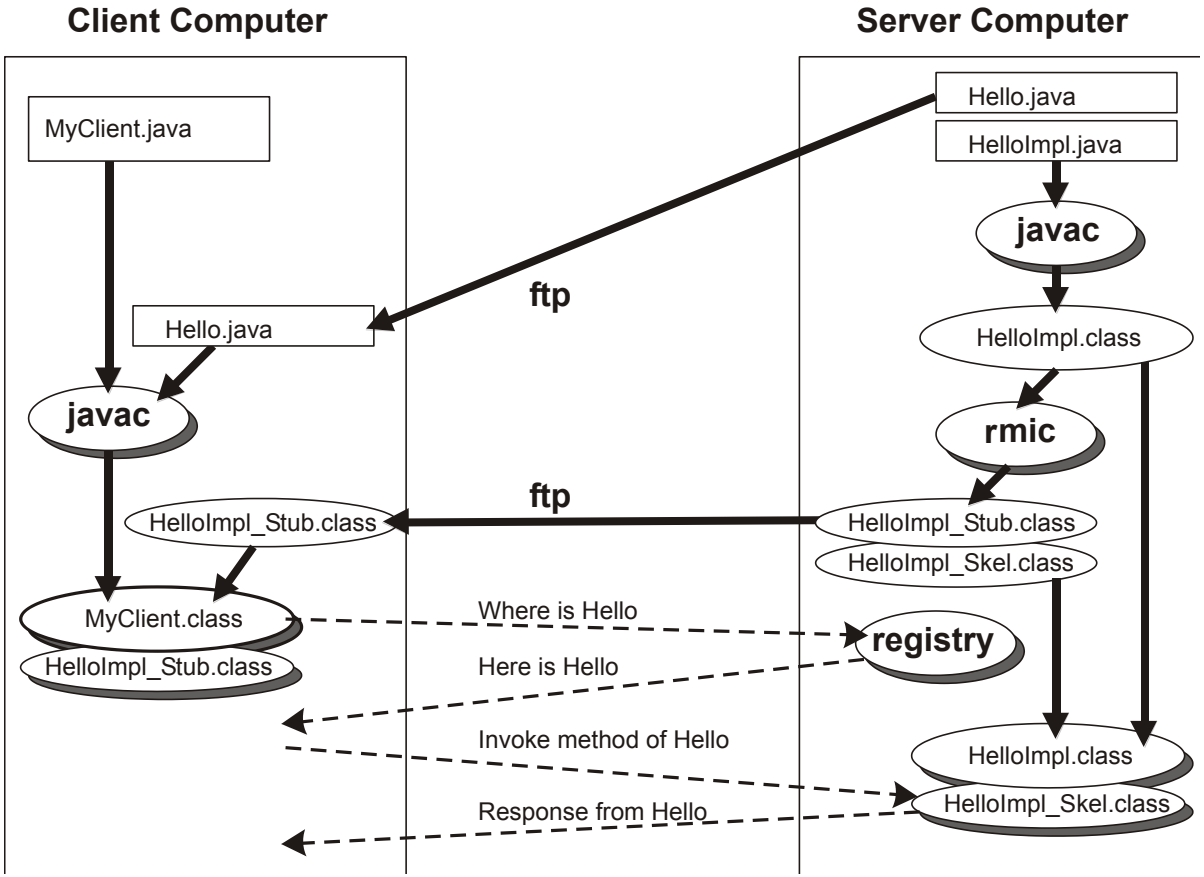
How Does a Client Object Find a Remote Server Object?

- There are two ways to do this:
  - **The client can invoke the *lookup()* method of the RMI Registry on the server machine, passing a URL as a parameter, that indicates the name of the machine and the server object, as in: *rmi://foo.bar.edu/TimeServer***
    - The RMI Registry returns a reference to a remote object (actually, a socket connection to the RRL, which connects to the skeleton)
    - This is what is done for the initial contact
  - **The client invokes a method of a remote object, which returns a reference to another remote object**
    - This is how most other references to remote objects are obtained

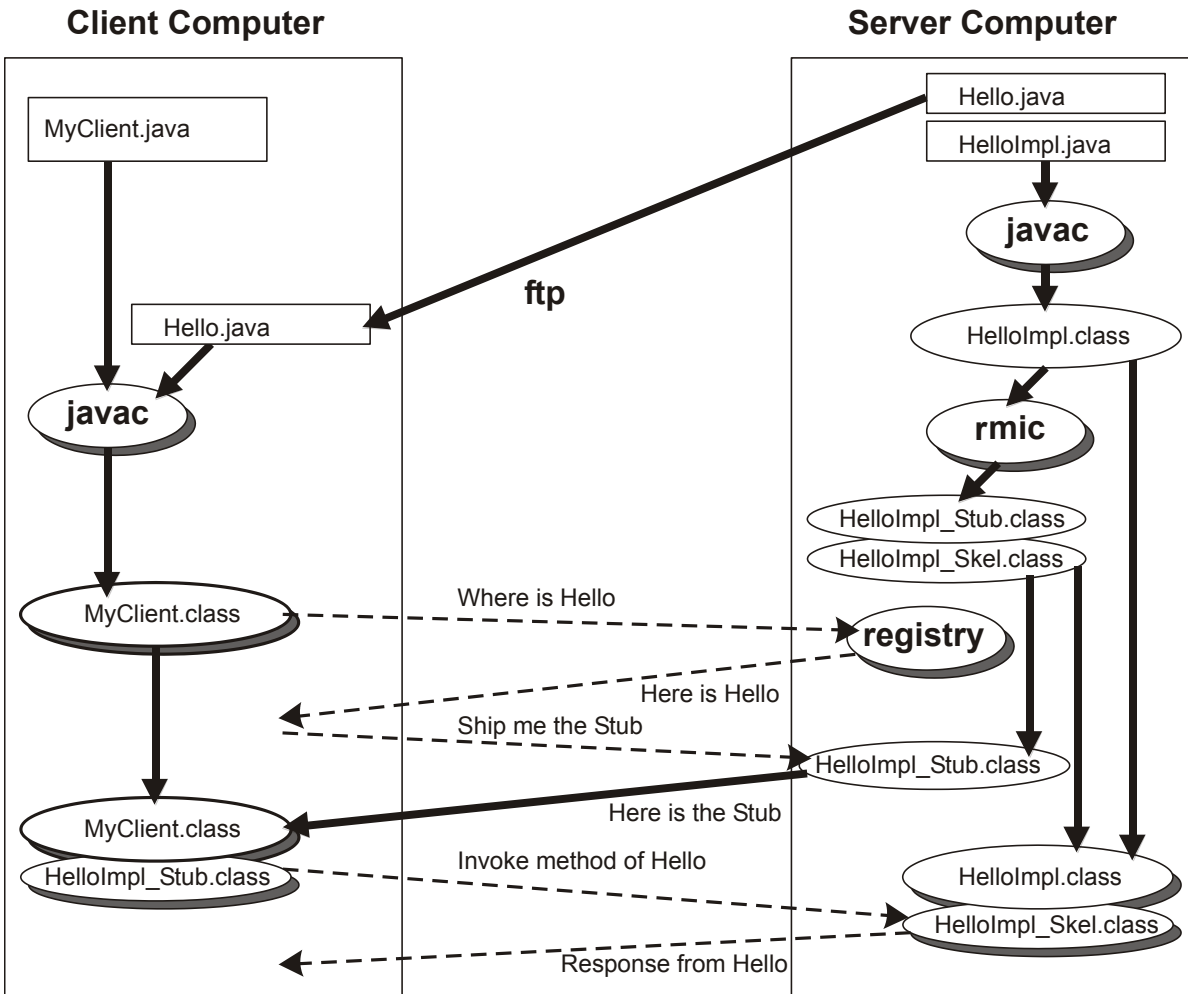
How Does a Client Object Get a Stub for the Server Object?

- The **RMI compiler**, *rmic*, processes the server source code and generates a set of *.class* files for the stub and the skeleton
- If the server source code is stored in the *XXX.java* file
  - **The stub is *XXX_Stub*, which is stored in the *XXX_Stub.class* file**
  - **The skeleton is *XXX_Skel*, which is stored in the *XXX_Skel.class* file**
- The programmer of the client object needs the interface for the server object, in order to know how to invoke the server methods
- The user on the client-side needs the interface for the server object and the *XXX_Stub.class* file, in order to run the client
  - **The *XXX_Stub.class* file can brought to the client-side before the client is compiled and run, as on Slide 12**
  - **Alternatively, the client can be compiled using only the interface for the server object, and the *XXX_Stub.class* file can be retrieved from the server machine at run time, as on Slide 13**

How Does a Client Object Get a Stub for the Server Object?



How Does a Client Object Get a Stub for the Server Object?



How Does RMI Pass Parameters and Results?

- RMI must package the parameters into a message
  - **This is easy for simple types such as *int***
- *Strin*gs and *vectors* are packaged as a length, followed by that many elements
- Some objects contain other objects
  - **Those must be sent across the network too**
- Everything in *java.lang*, *java.util, java.swing* can be sent across the network
- Some things cannot be sent across the network, such as *Threads*, *InputStreams*, *OutputStreams*, *java.sun.** because they are local to a machine

How Does RMI Handle Exceptions?

- It is possible that the network connection is down, the remote object is not loaded, or the remote object generates an exception
- All remote methods should handle exceptions and be defined with *throws java.rmi.RemoteException,* as in:

```
public String remoteMethod()
        throws java.rmi.RemoteException;
```

- All invocations should be placed within a *try/catch* block, as in:

```
try {
   remoteObject.remoteMethod();
}
catch (java.rmi.RemoteException  e) {
   System.err.println(e);
}
```

- For more precise handling of remote exceptions, there are 16 subclasses that extend *java.rmi.RemoteException*

CORBA

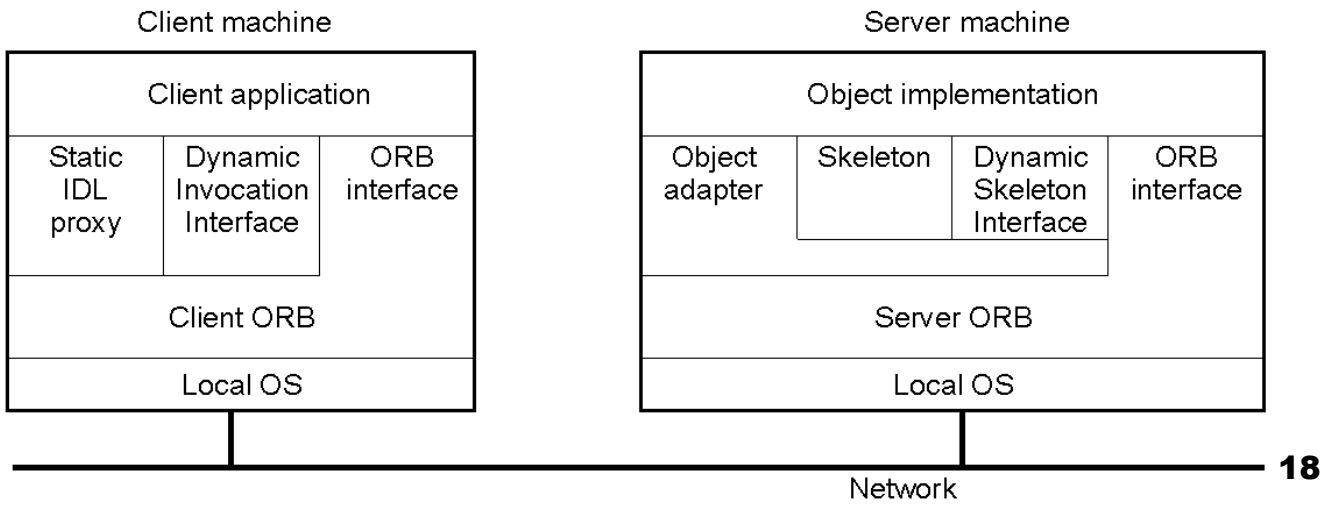
- **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA)
- Provides a distributed-object model and defines sets of ***specifications*** that define APIs for the middleware and the applications
  - **CORBA is *implemented* as middleware and as application services**
- Developed by the **Object Management Group (OMG)**, a consortium of 800+ member companies, in response to industrial demands for object-based middleware
- For enterprise application programmers, CORBA is legacy
- CORBA implementations are hidden in other middleware, e.g., Java EJB / J2EE

CORBA

- CORBA uses a typical remote object model in which an object at a server is accessible remotely through proxies
- CORBA follows an interface-based approach to objects
  - **An object may implement one or more interfaces**
  - **Interface descriptions are stored in an Interface Repository, and are looked up at runtime**
- CORBA specifications are given by interface descriptions, expressed in an **Interface Definition Language (IDL)**
- Mappings from IDL to specific programming languages (C, C++, Java, etc.) are part of the CORBA specifications

CORBA

- **Object Request Broker (ORB):** Middleware that connects clients and servers
- **Proxy / Skeleton:** Automatically generated code that does marshalling / unmarshalling of invocations / responses at the client / server
- **Dynamic Invocation / Skeleton Interface (DII / DSI):** Allows invocation requests / responses to be constructed at runtime

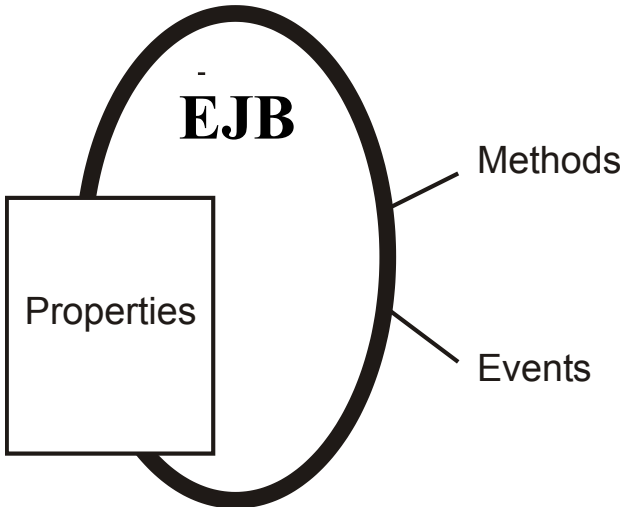


Enterprise Java Beans

- **Enterprise Java Beans (EJBs)** are a server-side technology that allows the creation and deployment of **Java components** that run within **EJB containers** in an **EJB/J2EE application server**
- EJB/J2EE has the goal of **portability**, i.e., an EJB application can run on any platform and with any vendor's EJB/J2EE application server
- Commercial application servers include IBM WebSphere, BEA WebLogic, Oracle 9I, and JBoss (open source)

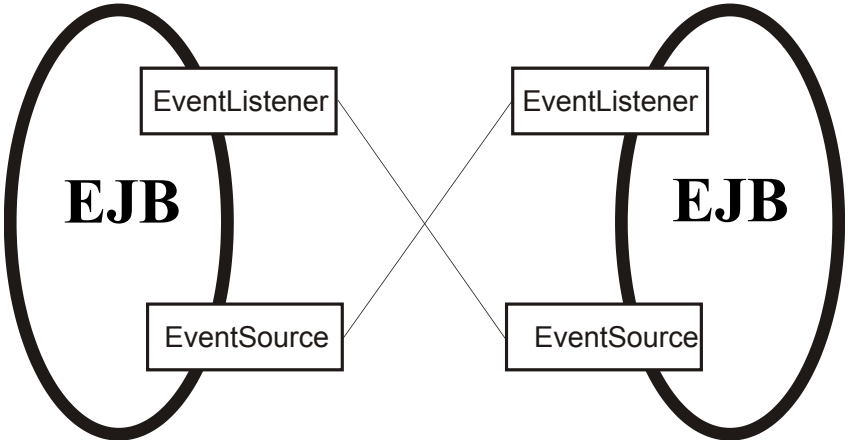
Enterprise Java Beans

- **Component** - A piece of software with a well-defined public interface and a model that defines how it communicates and interacts with other components
- **EJB** - A component that has **properties** together with **methods** and **events** by which it communicates and interacts with other EJBs



Enterprise Java Beans

- **Properties** - Provide the bean's internal representation, which can be saved between method invocations
- **Methods** - Follow a specific naming convention, i.e., *setXXX()* and *getXXX()*, that allow properties to be assigned and retrieved
- **Events** - Allow interactions with other beans, by means of *EventSource* and *EventListener* objects



EJB Container

- An **EJB container** contains EJBs and provides an interface between the EJBs and the clients
- An EJB client never accesses an EJB directly, but does so by means of **container methods** which, in turn, invoke the methods of the EJB
- An EJB container provides various generic services for the EJBs it contains, including:
  - **Lifecycle - Manages process, thread, object and Bean creation, destruction and cleanup**
  - **Security - Performs security checking for Beans**
  - **Transactions - Starts, commits and rolls back transactions on behalf of Beans**
  - **Persistence - Manages the storage and restoration of persistent state for Beans**

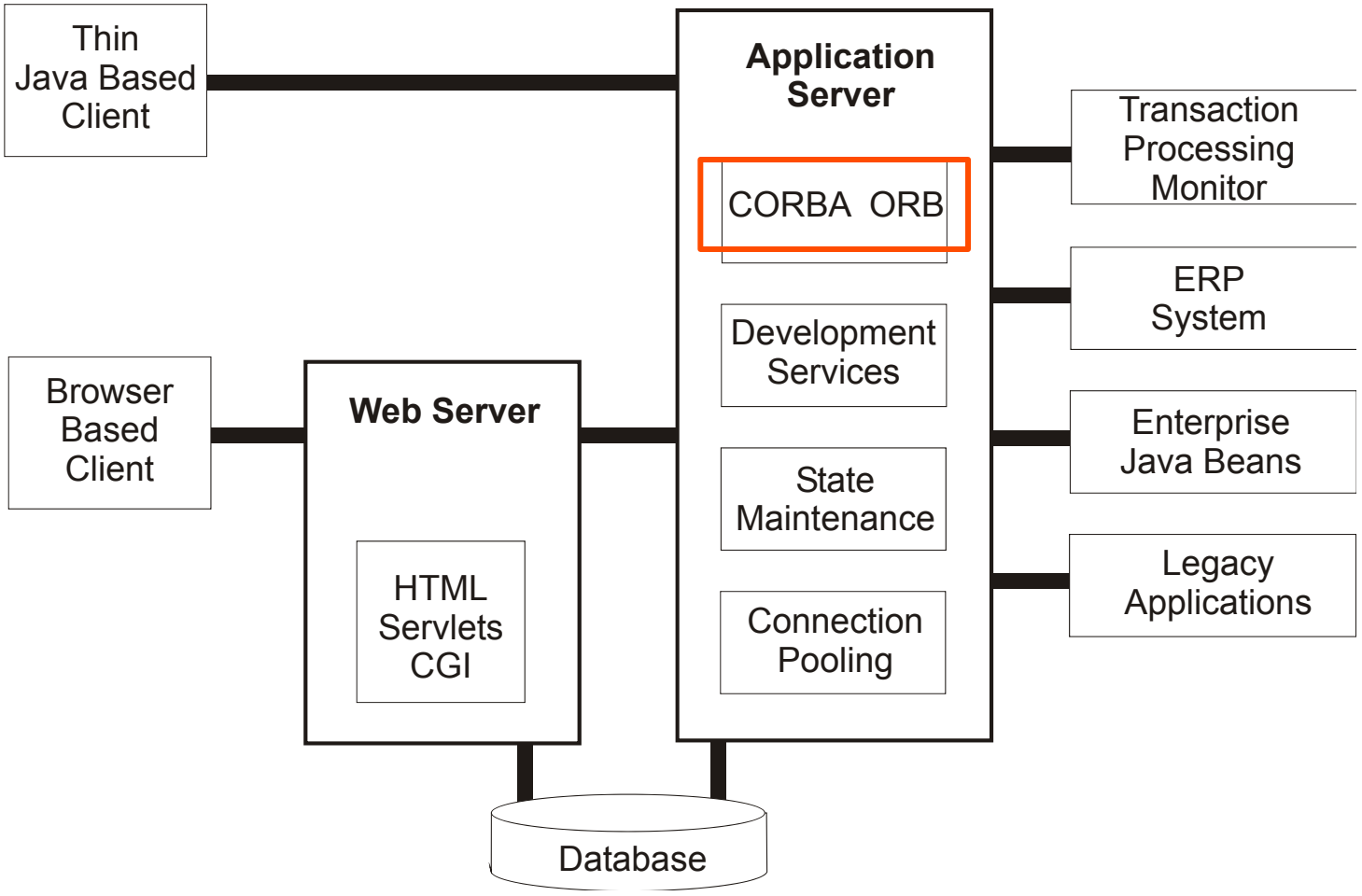
EJB/J2EE Application Server

- A typical **EJB/J2EE application server** provides
  - **Execution environment**
  - **Concurrent processing**
  - **Load balancing**
  - **Device access**
- A typical EJB/J2EE application server uses the
  - **Java Naming and Directory Interface (JNDI)**
  - **Java Transaction Service**
- An EJB client finds an EJB container using the JNDI

EJB/J2EE Application Server

- An EJB/J2EE application server typically provides
  - **High-performance Web Server or the ability to integrate with such a Web Server**
  - **Integration with a Database Management System (DBMS)**
  - **Ability to interface to a Transaction Processing (TP) Monitor (e.g., BEA's Tuxedo)**
  - **Ability to interface to an Enterprise Resource Planning (ERP) System (e.g., from SAP or QAD)**
  - **Integrated Development Environment (IDE) or the ability to integrate an IDE**

EJB/J2EE Architecture



Session Beans and Entity Beans

- **EJBs** are categorized as:
  - **Session Beans**, which are further categorized as
    - **Stateless Session Beans**, which can be pooled to serve multiple clients concurrently
    - **Stateful Session Beans**, which serve a single client (because they contain state related to that particular client)
  - **Entity Beans**, which can be shared by multiple clients and are typically associated with databases
- The states of Stateful Session Beans and Entity Beans can be persisted to, and retrieved from, disk

Session Beans and Entity Beans

- For **Entity Beans**, **persistence** is either
  - **Container Managed - Container is responsible for saving the Bean's state, or**
  - **Bean Managed - Bean is responsible for saving its own state by invoking database methods**
- An EJB has a **Deployment Descriptor**, either a ***SessionDescriptor*** or an ***EntityDescriptor***
  - **Container managed fields for Entity Beans are specified in the Deployment Descriptor**

Four-Tier EJB Application

- A typical **four-tier EJB application** that consists of
  - **Browser-Based Client**
  - **Web Server and Servlets/JSPs**
  - **EJBs in Containers provided by the EJB Application Server**
  - **Database Management System**