Chapter 6

Distributed Event Programming

6.1 Introduction

Distributed computing involves multiple computers working together to accomplish some task. The computers communicate with each other by sending messages back and forth across a network. This model of computing is common in many real world applications. For example, browsing the Internet involves a computer running a web browser and web servers feeding it pages. In addition, web browsing requires other specialized services invisible to the user, but supporting the interactions, such as the domain name server (DNS) which translates the URL the user enters to a binary form, the IP address, that the network understands.

Another practical example of distributed systems are Enterprise Resource Planning (ERP) systems. Historically, different departments within a business purchased computing systems to help them with their particular part of the business, e.g. payroll systems, inventory systems, human resources systems, manufacturing systems, etc. These diverse systems need to work with each other many business related tasks. For example, the sales system needs to work with the inventory system to make certain there are enough of each product being purchases. ERP systems were developed to fill this niche. Now, it is rare to find a company that is not running an ERP system like SAP or JDEdwards.

6.2 The Big Picture

To understand distributed systems we start by looking at the big picture. There are three types of things that participate in a computer application: data, computation and users. All three can be distributed to various degrees.

Data and Information

Data may exist in a central repository, e.g. in a database, or on a file server. Remote computers can then access them, as needed, either by making a local copy or by directly accessing the storage on a remove machine. Data may also be distributed for a variety of reasons. For example, various groups may own various parts of the data, e.g. as in the example above, the payroll department or the inventory department own their own data, and store them on a machine local to their department.

Computer scientists have been distributing data for decades. Modern operating systems let us mount remote file systems and treat them as if they are local. Languages like Java let us open remote files and read from them using the same I/O classes as processing local files.

Information

The last decade has seen significant effort to structure data into *information*. Data are just values. Information, by contrast, places the values into a context. For example, 39 is a datum.
<age>39</age> implies that we should treat 39 as an age. The *eXtensible Markup Language (XML)* and a variety of its derivatives, are currently the dominant way of structuring data into information. They use tags, as those shown here, to build hierarchical structures from the data. For example,

```
<person>
   <name>Erica Haller<\name>
   <age>39<\age>
   <profession>College Professor</profession>
<\person>
```

could be a simple XML structure describing a person.

If two separate computing systems are to access the information, they must first each understand the structure of the data. to meet this need, XML introduced *schema* which describe the structure of the data. Schema are XML documents that describe the structure of other XML documents.

Computation

Distributed computation takes place across spatially separated computing systems operating asynchronously. This is different than multi-threading discussed earlier in the text. Multi-threading gives us concurrency, but isn't considered distributed, as the threads share the application's code and heap. Distributed systems are more loosely coupled, communicating with each only via well defined protocols. For example, it is very common to have a database "back-end" running on a separate computer from the rest of the application. This allows extra computing power to be dedicated to running complex database queries and adds another level of security. The application sends a query to the database. The database executes the query and returns the result.

Distributed computation is still an important area in computer science research. It poses some ongoing interesting and challenging problems. For example, developing dynamic distributed systems, where services and devices come online and go offline as needed, is still very difficult. Some progress is being made. With protocols like Bluetooth, it is now possible for two devices to discover each other, once they are within reasonable proximity.

Users

It may seem odd at first to talk about users as being distributed, but in many cases the primary purpose of a distributed system is to coordinate distributed users. Online multiplayer games and business professionals holding online meetings are both examples. The most typical paradigm used for these applications is the client-server model, where each user interacts with client software, which, in turn, interacts with the server.

6.3 Distributed System Infrastructures

Obviously, if we are going to build distributed systems, we need hardware and software infrastructure to support the systems working together. A detailed understanding of networking hardware is beyond the scope of this text. For our purposes you may assume that the computers being used have a network card installed with a cable (or virtual cable) connected to it.

Since this is a programming text we do need to gain an understanding of a variety of types and levels of programming support supplied by various distributed computation infrastructures.

Low Level Primitives

At the lowest level, bytes of data are passed between two computers. Almost all programming languages include objects known as *sockets* to facilitate this computer to computer communication. A socket is simply a stream of data flowing between two computers. It is completely up to the developers to decide what data should be sent and received and how that data should be interpreted. On this level, the developers must deal with many subtle challenges. The program must potentially contend with different hardware and operating systems, different implementation languages, unreliable data communication, crash/recovery scenarios, security, and much more. For these reasons higher level abstractions are now commonly used.

Middleware

Middleware systems provide an infrastructure around which to build a distributed system. As the name *middleware* implies, the software sits 'between' the two communicating computers. Both systems must have the middleware installed. The application program then communicates with the middleware, which in turn takes over all low level details of communicating.

Middleware systems vary significantly based on the computing niche they are trying to serve. Here are a few examples:

The Common Object Request Broker Architecture (CORBA)
CORBA was designed to facilitate integrating *legacy systems*. Legacy systems are pre-existing systems that are still filling a useful role in an enterprise. Two legacy systems may have been written in different languages and run on different hardware with different operating systems. They may differ in how they represent numbers, e.g. big-endian vs. little-endian, or in the order parameters appear in a compiled method. The *Common Object Request Broker Architecture, CORBA*, abstracts away these nasty differences. A CORBA implementation supplies libraries that allow an application to communicate with an *Object Request Broker, an ORB*. Any system that can communicate with an ORB can communicate via the ORB with any other CORBA based system.

Web Services
The World Wide Web infrastructure is immense. There are millions of web servers running and millions upon millions of web browsers accessing them. Not only that – amazingly, it all works. A user sitting in Kenosha, WI can call up a web page on a server in Bangalore India, and it will arrive in just a few seconds. One of the things that makes the web work so well is that it is built on top of very well understood languages and protocols, e.g. *html* and *http*.

Web services take advantage of the web infrastructure to build distributed applications. Web services are programs that run on web servers. Because they are programs, they can provide more complex services than just returning simple web pages. Web services receive requests for information and return results in *XML* documents. Because HTML and XML are closely related tagged languages, the web infrastructure works well with both.

Jini
Jini is a Java based infrastructure developed to implement service oriented dynamic distributed systems. *Service* and *dynamic* are central to Jini's intended use. A Jini network is made up of services

that are used by clients. The service may be a chat service, or a toaster[1]. Dynamic means that the services availability can change. New services can join the network. Others may crash or have network failures. When a service comes online, it advertises itself. When a client wants to use an available service, it sends the service requests. Building robust, dynamic distributed systems is still one of the major ongoing challenges in distributed systems research. Jini is a step in that direction.

Advanced Infrastructures

There are dozens of other distributed system infrastructures. Some are production level and compete directly with those above. Other are research tools. Distributed systems programming is still a research field in academia, and more tools are always being developed.

6.4 Event Based and Distributed Systems

Conceptually, event based and distributed systems have much in common. They are both made up of different parts running fairly independently of each other. The parts communicate by sending discrete messages to each other.

More specifically, many of the properties discussed in Chapter 1 apply to both types of systems.

Loose Coupling

Agents in distributed systems are loosely coupled in ways very similar to the source handler coupling in event based systems.

Dynamic Binding

Event based systems register handlers at runtime. Distributed systems go through a setup phase similar to handler registration. In a distributed system, a program begins running on each system. Sometime thereafter communication links are established between them.

Nonblocking

In general, event sources do not block, but continue to run, not waiting for handlers to complete. Similarly, in many distributed systems messages may be sent to remote agents and the local process continues running, not waiting for the remote computer to complete or return a message. If a return message, e.g. a result, is expected, a common model is to listen for it using a separate thread.

Decentralized Control

We say that event based systems use decentralized control because an event can cause a handler to execute in object A and a different handler to execute in object B. Each of these can fire ten more events that run handlers in still different parts of the code. Each handler takes responsibility for carrying out some part of the overall computing task, but there is no centralized authority taking charge of completing the task.

Processing in distributed systems can be very similar. Messages are passed from system to system, each carrying out its responsibilities, but again with no particular system in charge. The message senders assume that the receiving agent knows what it is supposed to do.

[1]Toasters serve us toast, don't they?

Nondeterminism

Both types of systems contain nondeterminism. As we have seen, the order in which events are handled may not be the same as the order in which they were fired. It depends on the underlying infrastructure. The network introduces significant nondeterminism in a distributed system. There may be arbitrarily long delays between when the message is sent and when it is received. Similarly, the order the packets arrive may not be the same as the order in which they were sent. It depends on their route through the network and the network load.

6.5 Publish - Subscribe

There are several additional ways that distributed systems may be even more event driven. First, as introduced in Chapter 1, many distributed systems work on a publish-subscribe basis. That is, System A registers its interest in receiving information of Type X from System B. Any time this information becomes available, B sends it to A and all other registered subscribers. This interaction motif closely parallels what happens between sources and handlers in event based systems.

Remote Events

A few distributed system infrastructures, e.g. CORBA and Jini, support remote events. These services are the equivalent of the event infrastructure discussed in Chapter 3. Event sources, running on one system, fire events to handlers located on a different system. The event service serves as an intermediary. It receives the event from the source and queues it until the handler is ready to process it.

6.6 Challenges Developing Distributed Systems

Programming distributed systems poses numerous challenges. In many ways these challenges parallel those of developing event based systems.

Shared Resources

If you have multiple processes competing for resources, you introduce the possibility of *deadlock*, where each process is waiting for a resource another holds, so no one makes progress. These types of problems also exist in any multitasking operating system. Solutions can be implemented using semaphores and monitors, but designs need to be very carefully thought through.

Nondeterminism

In a distributed system there is no way of knowing how fast remote processes are executing or how long messages will take to be delivered. This means that messages may be sent and arrive in many different orders. Each process in the system should continue functioning correctly, regardless of the order the messages are received.

Unreliable Networks

Distributed systems depend directly on an underlying network to communicate. Networks are much more reliable than they were in the past, but failures still occur. *Packets*, messages between two nodes in our system, may be dropped. Robust distributed systems should be able to recover from dropped packets and temporary network outages.

Failure and Recovery

Computer hardware and software fails. Distributed systems should be able to recover from *transient failures*. By this, we mean that if a node fails and is brought back to life, the computation should be able to continue. Consider an RSS feed. Users subscribe to an RSS feed about some topic, say rugby news. Any time there is some new rugby news, the feed sends all registered users the story. If the RSS feed fails and recovers, users should not have to re-subscribe.

Security

Distributed systems contain more potential security risks than standalone applications. Each process in the application needs to be secured, as does the communication between them.

6.7 The CORBA Infrastructure

The authors chose CORBA as the distributed system infrastructure for their examples in this chapter. The *Object Management Group (OMG)* designed CORBA as a specification, not an implementation. Different vendors provide their own CORBA implementations. As such, CORBA is platform independent and avoids issues involving programming languages and operating systems. For example, an accounting system, written in COBOL, running on an IBM mainframe, can communicate using CORBA with a payroll system written in C++, running on a Linux machine.

CORBA has been around long enough to have many solid implementations. A CORBA implementation comes with Sun's Java SDK making it widely available. Our examples will use this Java implementation, but here are CORBA implementations for most operating systems and programming languages.

Like any major software technology, there is a lot to learn before you can claim to be proficient in it. CORBA is no exception, and the reader should not expect to be an expert CORBA programmer when they have finished reading this chapter. Introductory CORBA can be cookbooked, however. That is, the 20-30 lines of CORBA code that a process needs can be borrowed from other examples with only minimal changes. Also, CORBA doesn't require as much detailed technical knowledge to run as many of the others infrastructures. This makes it quicker for students to jump start their programming.

6.7.1 Services, Tools and Classes

Perhaps the easiest way to understand CORBA is to think about the CORBA infrastructure as consisting of three different types of entities: services, tools and classes. The *services* are predefined CORBA objects that we will run along with our application code. The *tools* are compilers and other assorted software that we will use while we are developing our programs. The *classes* are programming language classes that we will depend on within our code.

CORBA Services

The CORBA specification contains a dozen optional services that may be delivered by a CORBA provider. Here we will discuss only two:

- **Name Service**
  In any distributed system, agents need to find each other. A web browser needs to find the web server before it can download a page. An airline reservation system needs to find the database of flight information before it can make a reservation. CORBA is no exception. Our

code, a CORBA client, needs to find other CORBA objects before it can communicate with them. CORBA objects are located using their *Interoperable Object Reference (IOR).* The IOR is a binary string containing, among other things, the IP and port where the service may be found. There are multiple ways a client may obtain a service's IOR:

– We can hardcode into our client the IORs for other CORBA objects. This has major drawbacks, however. If the service moves to another system, for example, you have to change your program, too..
– We can give the location of the remote CORBA objects as runtime parameters, or as program input. This, too, would work, but if our program needs to interact with many services, the input becomes excessive.
– The most elegant solution is to use the CORBA name service. The *name service* is a CORBA object that maintains a table of names and IORs. We can use CORBA's name service to keep track of where CORBA objects are located. CORBA objects register with the name service. Other CORBA objects query the name service which returns the IOR of the desired service.

The observant reader will notice immediately that the name service, by itself, does not completely solve the problem. We still need to find the location of the name service, and we can't use the name service to find it! The problem is much simpler, however, as we only need to find the name service, which can then provide us with the location of other the other CORBA objects. An easy solution to this problem is to pass the location of the name service as runtime parameters, which is the approach we take in our examples. More on what those parameters look like in the section below.

The name service delivered with Sun's CORBA implementation is the *Object Request Broker Daemon (orbd). orbd* is a standalone application that we will start at the command line, just as we will start our own processes.

- **Event Service**
The CORBA *event service* provides a delivery mechanism for distributed events. Remote events behave much like local events. There are times when we don't need to call a method on a remote object, we just need to notify the object that something of interest has occurred. CORBA event sources fire their events to the event service, which takes responsibility to delivering them to the handlers. Unfortunately, Java's SDK does not provide a CORBA Event Service and it is not discussed further here.

CORBA Tools

In CORBA terms, the public interface accessible by other CORBA objects is known as as a *service*, and the objects that implement the service are known as servers. CORBA clients access the service by calling the server's methods[2].

CORBA defines a "meta-language", the *Interface Definition Language (IDL)* to represent the signatures of the methods a CORBA service provides. These are the methods that other CORBA objects can access remotely. IDL is necessary because the different CORBA objects may be written in different languages. IDL is the universal language for all CORBA implementations.

[2]Note, however, that there is nothing stopping a CORBA object from being both a client and server. That is, a CORBA object can both provide services to others and call on other servers. The only real requirement is that a CORBA service needs to have its public interface defined.

Compiling IDL is a two step process. First, IDL descriptions are compiled into source code of another high level language, in our case, Java. If two CORBA objects are written in different languages, however, then different IDL compilers are used to generate the appropriate source code in each language. The IDL generated source code is later compiled, along with the rest of the CORBA object, using a standard language compiler, e.g. g++ or javac.

The primary CORBA tool we will need to develop our applications is the *IDL compiler, idlj.* `idlj` comes with Sun's Java SDK.

CORBA Interfaces and Classes

The Java package hierarchy beginning `org.omg.*` contains literally hundreds of Java interfaces and classes for CORBA development. In one chapter, we can't hope to acquaint the reader with all of them, but it is possible to write CORBA based programs with a working knowledge of only a few.

- *Object Request Broker*
  At the heart of CORBA is the *Object Request Broker (ORB).* It is the ORBs that supply the communication infrastructure. The ORBs communicate with each other, and pass information on to our application objects. Every process in our application will be associated with an ORB. If our accounting system can communicate with an ORB and our human resources system can communicate with an ORB, then they can communicate with each other.

- *Portable Object Adapter*
  Portable Object Adapters (POAs) sit between the ORB and the service. They help the ORB call methods in the service and pass the results back to the ORB. Since each service will have its own collection of methods, POAs are specialized to a particular service.

Figure 6.1 shows the structure of a client/server call in CORBA. The developer writes the **Client** and the **Server**. The ORBs are supplied by CORBA. Stubs and POAs are generated by `idlj`. The stub implements the required methods, but does not carry out the operations itself. Instead, the stub uses the ORB to communicate with the server, where the work is done. The server provides concrete implementations of the methods. Results are returned along the reverse path to the client.

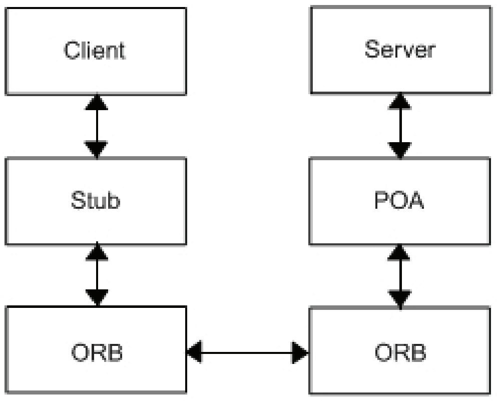

Figure 6.1: The main objects involved in invoking a remote CORBA service from a client.

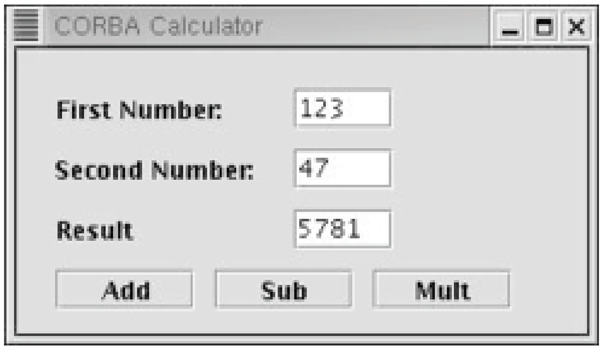

Figure 6.2: The Graphical User Interface for the calculator.

6.7.2 CORBA Programming Steps

The following steps are used to create and run CORBA based programs.

1. *Design the system*
   Designing a distributed system requires more work than a standalone application. The designer needs to decide different agents will be involved and what will be the responsibilities of each. The examples in this chapter are intentionally contrived to use a very small number of agents, but in the real world there may be hundreds of agents collaborating on some task, and the design decisions are far from trivial.

2. *Develop the IDL*
   Create and compile the idl description for each of the service agents. If you have done a good job in Step #1, this should follow easily.

3. *Implement the Agents*
   Implement the system agents, both the clients and the servers, using the interfaces and classes developed in Step #2.

4. *Start Everything Running*
   It is frequently the case in distributed systems that the various agents must be started in a particular order. With CORBA we will generally start the name service, then our servers, finally the clients.

6.8 Calculator Example

In the remainder of this chapter we present several small CORBA examples. Only illustrative code fragments are included. Complete source code for each of the examples is available from the text website.

Our first example is that of a calculator. In practice, there is really no point in having a distributed service to do simple math. However, a calculator is a well understood application so it serves as a nice introductory example.

Figure 6.2 shows the calculator's interface. The user enters two integers and the calculator performs one of three operations, addition, subtraction or multiplication, on them.

6.8.1 Defining the IDL

Our calculator service is simple. It provides three operations, addition, subtraction and multiplication. Below is our IDL file that contains the declares our service.

```
/* Calculator.idl
 * This file contains the interface definitions for a simple calculator
 *
 * Written by: Stuart Hansen
 *
 */
module calculator {
   interface Calculator {

      // A method to add two numbers
      long add (in long a, in long b);

      // A method to subtract two numbers
      long sub (in long a, in long b);

      // A method to multiply two numbers
      long mult (in long a, in long b);
   };
};
```

IDL is a meta-language. It provides reserved words and data types that can be translated into high level programming languages. The IDL constructs in this example match fairly closely with Java's, but not exactly.

`module` translates to be `package` in Java.
`interface` remains `interface`.
`long` translates to Java's `int`[3].

The `in` in front of the `long` as part of a parameter declaration declares the parameter to be pass by value. That is, `a` and `b` are copied to the server, but are not returned. Java already uses pass by value for primitive parameters, so this presents no problems. Other legal parameter descriptors are `out` and `inout`. Java does not support copy out, or copy in and out parameters, so additional support classes called *holders*, are needed to use these. This example does not use `out` or `inout` parameters, so holders aren't needed.

6.8.2 Compile the IDL file

Java's IDL compiler, `idlj`, can be found in the `bin` directory of the JRE. It translates the IDL description into Java source code. It generates the stub and the POA classes discussed above as well as a few other interfaces and classes. The `idlj` command to translate this file is:

```
idlj -fall Calculator.idl
```

The `-fall` option tells `idlj` to generate `all` the CORBA classes needed to complete both the client and the server. The files generated are:

- `CalculatorOperations.java` and `Calculator.java` are interfaces containing the Java declarations for the Calculator. The contents of `CalculatorOperations.java` are:

[3]If you want a Java `long`, use IDL's `long long`.

```
package calculator;


/**
* calculator/CalculatorOperations.java .
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from Calculator.idl
* Monday, February 2, 2009 9:18:49 AM CDT
*/

public interface CalculatorOperations
{

  // A method to add two numbers
  int add (int a, int b);

  // A method to subtract two numbers
  int sub (int a, int b);

  // A method to multiply two numbers
  int mult (int a, int b);
} // interface CalculatorOperations
```

Note how neatly each of the idl methods translated to a Java method.

`Calculator.java` contains an interface that extends `CalculatorOperations` and CORBA's `Object` class. Our calculator client will access methods from an object that implements `Calculator`, since it will implement `add()`, etc. and will be a CORBA `Object`.

- `CalculatorPOA.java` contains an abstract class that implements `CalculatorOperations`. It does not implement any of the methods from `CalculatorOperations`, however. To complete our server, we will extend `CalculatorPOA` and define the methods.

- `CalculatorHelper.java` contains some useful static methods.

- `_CalculatorStub.java` contains some CORBA methods to facilitate communications on the client end. Client code will not interact with it directly however, but will do so through the `CalculatorHelper` class.

- `CalculatorHolder.java` is a wrapper class used for `out` and `inout` parameters. Since our example contains neither, this class is not used.

6.8.3 Completing the Server

To complete coding the server, we will extend `CalculatorPOA`, implement the methods required by `CalculatorOperations` and write a constructor that does some CORBA setup.

```java
import calculator.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.NamingContextPackage.*;

// This class implements a very simple CORBA server.
// It provides add, sub, and mult methods that
// clients may invoke.
//
// This Server requires that a CORBA Name Service be running.
//
// Written by: Stuart Hansen
// Date: February 2, 2009
public class CalculatorServer extends CalculatorPOA {

    private ORB orb;  // the orb for this server
    private POA rootpoa; // the root POA for this server

    // The constructor sets up the ORB for communication
    public CalculatorServer (String [] args)
    {
        setUpServerORB(args);
    }

    // implement the add method
    public int add(int a, int b)
    {
        return a + b;
    }

    // implement the sub method
    public int sub(int a, int b)
    {
        return a - b;
    }

     // implement the mult method
    public int mult(int a, int b)
    {
        return a * b;
    }
```

Lines	Commentary
1–5	Import various CORBA packages which contain supporting classes.
17–18	Declare the `ORB` and `POA` variables that we will use to facilitate communication.
20–24	Define the constructor which calls a setup routine to initialize our ORB.
26–42	Define the service methods declared in the idl and `CalculatorOperations`.

Setting up the CORBA Communication

The final programming step to complete the server is to set up the CORBA communications. Below is the listing of the method, `setUpServerORB()`, that accomplishes this. `setUpServerORB()` may at first seem complex with obscure CORBA method calls, but we will analyze it line by line, to show how it accomplishes its tasks. For those readers not interested in the details, the same method can be used for almost any server side application, just replacing all references to "Calculator" with the name of your server.

```java

    // This method setups the Server ORB
    public void setUpServerORB(String [] args)
    {
        try{
            // create and initialize the ORB
            orb = ORB.init(args, null);

            // get reference to rootpoa & activate the POAManager
            rootpoa = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();

            // Convert our server to a CORBA object and IOR
            org.omg.CORBA.Object ref =
                rootpoa.servant_to_reference(this);
            Calculator calc = CalculatorHelper.narrow(ref);

            // Look up the Name Service
            org.omg.CORBA.Object nameServiceObj =
              orb.resolve_initial_references("NameService");
            NamingContextExt nameService =
                NamingContextExtHelper.narrow(nameServiceObj);

            // Bind (register) the Calculator Server with the Name Service
            String name = "CalculatorServer";
            NameComponent path[] = nameService.to_name( name );
            nameService.rebind(path, calc);

            System.out.println("Calculator Server Ready");

            // wait for client requests
            orb.run();

        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }

    // The main program simply creates a new Calculator Server
    public static void main(String [] args) {
      new CalculatorServer(args);
    }
}
```

Lines	Commentary
45	The parameters coming into `setUpServerORB()` come from the `main()` method and contain the location where the name service, `orbd`, is running. See the section below on starting the application.
49	Initialize the ORB. We pass along the name service location to the ORB. In our case, the second parameter is `null`, but could contain a list of application specific properties. Note that you do not construct an ORB, only call its `init` method.
52-54	Set up the POAs for our system. POAs exist in a hierarchical structure. For this example we really don't need to worry about multiple POAs, but there is one `rootPOA` for our server.
52-53	Ask the ORB to give us a reference to the `rootPOA`. Wrap this request with a call to `narrow()`. Narrowing is CORBA's equivalent of casting to a more specific type. Line 53 returns a CORBA object. `narrow()` converts it to a `rootPOA`.
54	Activate the rootPOA's manager.
56-70	Register our server with the name service.
56-59	Convert our server (`this`) to a CORBA object, using the `rootPOA`.
62-65	Request a reference to the name service from the orb and again narrow it to be a name service (`NamingContextExt`) object.
68-70	Register our server with the name service.
75	Place the ORB into listening mode, actively waiting for requests from clients. Note that the ORB executes `run()` in the current thread. Because `run()` waits for clients to send requests to the server, any code following `run()` will not be reached.
77–80	Handle exceptions. Most exceptions that occur when starting out programming CORBA will be for null references. For instance, if the name service is not running, `orb.resolve_initial_references("NameService");` will return `null`. An exception will be thrown on the next line, where we try to narrow it to be a `NamingContextExt`. The problem is not with the narrowing, but with the null reference.
83–86	The `main()` method simply creates a `CalculatorServer`.

6.8.4 The Client

Much of the code for the client is GUI programming, which while event based, is not of interest to us in this chapter. We show only the portions of the code that are central to our understanding of CORBA.

```
 1 import calculator.*;
 2 import org.omg.CORBA.*;
 3 import org.omg.CosNaming.*;
 4 import org.omg.CosNaming.NamingContextPackage.*;
 5
 6 import java.awt.*;
 7 import java.awt.event.*;
 8 import java.util.*;
 9 import javax.swing.*;
10
11 // This class implements a simple calculator client program.
12 // It relies on a CORBA server to do the actual calculations
13 //
14 // Written by: Stuart Hansen
15 //
16 public class CalculatorClient extends JFrame {
17
18     private Calculator calc;  // the remote calculator
19
20     // The GUI Components
21     private JLabel firstLabel, secondLabel, resultLabel;
22     private JTextField first, second, result;
23     private JButton addButton, subButton, multButton;
24
25     // The constructor sets up the ORB and the GUI
26     public CalculatorClient(String [] args)
27     {
28         setUpORB(args);
29         setUpGUI();
30     }
```

Lines	Commentary
18	Declare a `Calculator` object. It looks like a local object and its methods will be invoked just like a local object. The only difference will be when we instantiate it.
28	`setUpORB()` is shown below.
29	`setUpGUI()` is omitted to save space. The complete listing may be found the text's website.

```
88     // An inner class to handle clicks on the add button
89     private class AddHandler implements ActionListener
90     {
91         public void actionPerformed (ActionEvent e)
92         {
93             int firstNum = Integer.parseInt(first.getText());
94             int secondNum = Integer.parseInt(second.getText());
95             int resultNum = calc.add(firstNum, secondNum);
96             result.setText(Integer.toString(resultNum));
97         }
98     }
```

The interesting part of this handler is line 95 where we call `calc.add()`. As we will see below `calc` is a reference to a CORBA object that handles all the communication with the server, but it looks completely like a local object. All the communication with the server is completely hidden. Subtraction and multiplication handlers follow the same pattern and are omitted.

```
124     // Set up the data communication with the server
125     private void setUpORB(String [] args)
126     {
127         try {
128             // Create and initialize the ORB
129             ORB orb = ORB.init(args, null);
130
131             // Get a reference to the name service
132             org.omg.CORBA.Object nameServiceObj =
133                 orb.resolve_initial_references("NameService");
134
135             // Narrow (cast) the object reference to a name service reference
136             NamingContextExt nameService =
137                 NamingContextExtHelper.narrow(nameServiceObj);
138
139             // Get a reference to the calculator from the name service
140             org.omg.CORBA.Object calcObj =
141                     nameService.resolve_str("CalculatorServer");
142
143             // Narrow (cast) the object reference to a calculator reference
144             calc = CalculatorHelper.narrow(calcObj);
145
146         } catch (Exception e) {
147             System.out.println("ERROR : " + e) ;
148             e.printStackTrace(System.out);
149         }
150     }
151
152     // The main simply makes a new calculator client
153     public static void main(String args[]) {
154         new CalculatorClient (args);
155     }
156 }
```

Setting up the client's ORB follows a slightly different pattern than setting up the server's. The client does not register with the name service, since it is not receiving any calls from other objects. For the same reason, its ORB will not be waiting to receive requests. On the other hand, the client has to use the name service to find the calculator server.

Lines	**Commentary**
129	Initialize the ORB.
131–137	Use the ORB to find the name service and narrow it to be a `NamingContextExt`, a name service object. This code parallels that found in the server.
139-143	Use the name service to find the calculator server and narrow it to be a `Calculator`.

6.8.5 Starting the Application

Three separate processes must be started to run our application, the name service, the server and the client. They must be started in the order shown below.

```
orbd -ORBInitialPort 1060
java CalculatorServer -ORBInitialPort 1060
java CalculatorClient -ORBInitialPort 1060
```

`orbd` is the Java's CORBA name service. The `-ORBInitialPort 1060` specifies which IP port that should be used when the client and server communicate with the name service. Any unused port number will work. Some operating systems block lower numbered ports, so choosing a number above 1024 is appropriate.

In the lines shown above, all three programs are running on the same machine. The name service, server and client may also run on different machines. In this case use `-ORBInitialHost` when starting the server and client, and specify the remote machine where the name service is running. For example, if `orbd` is started on a machine named `onion.cs.uwp.edu` and the server and client are started on `bratwurst.cs.uwp.edu`, then the commands will be:

On onion:

```
orbd -ORBInitialPort 1060
```

On bratwurst:

```
java CalculatorServer -ORBInitialHost onion.cs.uwp.edu -ORBInitialPort 1060
java CalculatorClient -ORBInitialHost onion.cs.uwp.edu -ORBInitialPort 1060
```

6.9 Asynchronous Method Invocation

The previous example illustrated a client-server application. The client blocked on each call to the server, waiting for results. This made sense, as we didn't want to move on to a new calculation until we obtained results from the previous on. The same blocking model is used for most client-server interactions. A web browser waits for a web server to return a page. An ftp client waits for an ftp server to download a file it requests.

There are nonblocking models for distributed computation, as well. Messages are sent between agents, but the sender does not wait for a response. The nonblocking models are more event like. In event based programming our event sources do not block waiting for the handlers to complete. In nonblocking distributed systems, the message sender does not block waiting for a response from the message recipient.

6.9.1 CORBA and Asynchronous Messages

CORBA's IDL declares asynchronous methods as `oneway`. A `oneway` method has no return value and has no `out` or `inout` parameters. The call is made to the server and the client continues running, not waiting for results. When a `oneway` method is invoked, the server acts much as an event handler does in a standalone event based application. It receives the message. It runs the method specified (equivalent to an event handler) and does not return any value to the client. Amazingly, the only change needed in our coding style is to include the word `oneway`.

6.9.2 Heart Monitor Example

Heart patients in hospitals are frequently hooked up to monitors to keep track of heart rate, blood pressure, or any of a number of other critical values. The monitors relay signals down to the nursing station, so a small team of nurses can easily keep track of all the patients on a wing. The software running at the nurses' station is the server, receiving messages from the patients, a.k.a. the clients.

Below is a simple idl file for such a system.

```
 1 /* Heart.idl
 2  * This file contains an idl description of a heart rate monitor
 3  * at a nurses station.  It receives messages from local monitors in
 4  * each patient's room.
 5  *
 6  * Written by: Stuart Hansen
 7  *
 8  */
 9 module heartMonitor {
10    interface HeartRateMonitor {
11
12       // This method registers a local monitor with the server
13       oneway void register (in string name);
14
15       // This method unregisters a local monitor with the server
16       oneway void unregister (in string name);
17
18       // A method to receive a new heart rate value
19       oneway void newRate (in string name, in long rate);
20
21       // A method to sound an alarm when the heart rate goes too fast or
22       // too slow.  It is left to the local client to decide what the
23       // appropriate values for the alarm are.  Typical values in a real
24       // hospital setting would be 40 and 140.
25       oneway void soundAlarm (in string name, in long rate);
26    };
27 };
```

Lines	**Commentary**
13	`register()` is called when a patient is initially hooked into the system. It informs the nursing station monitor that data will be arriving for this patient.
16	`unregister()` is called when the patient is taken off the system.
19	`newRate()` sends the current heart rate to the nursing station monitor.
25	`soundAlarm()` sends a message that the patient's heart rate is too high or too low.

The only new idl type in this example is string. An IDL `string` translates directly to Java's `String`.

The text's website contains complete code listings for `HeartPatient.java` and `NursingStation.java`. The classes are left intentionally simple, in order to emphasize their CORBA aspects.

6.10 Peer to Peer Distributed Applications

Our heart monitor example introduced asynchronous messages, but was still a client-server application. The patients (the clients) sent messages to the nurses' station, the server. In peer to peer applications all objects can both originate and receive messages. One way to think of peer to peer applications is that the objects will behave as both clients and servers.

6.10.1 CORBA Peer to Peer Applications

If all objects behave completely identically a single idl file describing their methods will suffice.

Frequently, however, even though all objects can both originate and receive messages, the communication remains asymmetric, with different objects sending and receiving different messages. This type of model requires multiple CORBA interfaces, one for each role in the system.

6.10.2 Chat Example

Consider a typical chat system. Users log onto a chat server. While logged on, they can send messages to the other users via the server. They can also receive messages from the other users, again via the server. Two interfaces are needed, one for the clients and one for the server, because both types of objects are sending and receiving messages.

chat.idl

```
/* chat.idl
   Written by: Stuart Hansen

   This file contains the idl code for a CORBA chat application.
*/
module chatpkg {

   // The Message struct contains all information for passing
   // messages to the server and from the server to the clients.
   struct Message {
      string sender;
      string messageText;
   };
```

```

    // A ChatClient sends and receives messages only with the server.
    // Other client communicate with this client via the server.
    interface ChatClient {

        // Receive a message that a client is registered with the server
        oneway void newClientNotification (in string clientName);

        // Receive a message that a client has deregistered with the server
        oneway void clientUnregisteredNotification (in string clientName);

        // Receive a message from another client via the server
        oneway void sendMessageToClient (in Message message);

        // Receive a whispered message from another client via the server
        oneway void whisperMessageToClient (in Message message);
    };

    // The ChatServer has methods to register/unregister ChatClients
    // and to receive messages from Clients.
    interface ChatServer {

        // Register a new ChatClient
        //   return values
        //     0 = taken
        //     1 = ok */
        long register (in string clientName);

        // Unregister a ChatClient
        oneway void unregister (in string clientName);

        // Post a message to all registered clients
        oneway void post (in Message message);

        // Whisper a message to one other client
        oneway void whisper (in string clientName, in Message message);
    };
};
```

Lines	Commentary
10–13	Define a `struct`, which is a new idl entity in this example. A struct is a class that contains only public data. There are no methods and no private data. Structs translate to Java classes where all data elements are public. In this case we use it to define a data structure to hold messages. Messages will be objects of type `Message`. They contain both the name of the sender and the text. The Message class is created when we compile our idl file using `idlj`:
17–30	Define the chat client's interface. Note that some of the method names may be a bit confusing. Recall that the `ChatClient` methods will be implemented by the clients, but called by the server. We chose to name them from the server's perspective. For example, `oneway void sendMessageToClient (in Message message);` is a method within the client class that the server will call when it wants to send a message to the client. From the client's point a name like `receiveMessage()` may be a better name.
34–50	Define the chat server's interface. A chat client sends a message to all other clients by sending it to the server's `post()` method for distribution.

```java
package chatpkg;

/**
* chatpkg/Message.java .
* Generated by the IDL-to-Java compiler (portable), version "3.2"
* from chat.idl
* Tuesday, February 3, 2009 1:24:27 PM CST
*/

public final class Message implements org.omg.CORBA.portable.IDLEntity
{
  public String sender = null;
  public String messageText = null;

  public Message ()
  {
  } // ctor

  public Message (String _sender, String _messageText)
  {
    sender = _sender;
    messageText = _messageText;
  } // ctor
} // class Message
```

Client Connection to the ORB

The code in the chat clients to connect to their ORB and the server is a a combination of what was previously in the client and server classes. Code from a typical client is shown below.

```
        // We run the orb waiting method in a separate thread so that we can
        // continue processing in the main thread
        public void run ()
        {
            orb.run();
        }

        // use NameService to connect to time server
        private void connectToChatServer( String [] params )
        {
            try {
                // Initialize the orb
                orb = ORB.init( params, null );

                // Connect to the name service
                org.omg.CORBA.Object corbaObject =
                        orb.resolve_initial_references("NameService");
                nameService = NamingContextExtHelper.narrow( corbaObject );

                // Look up the chat server
                NameComponent nameComponent =
                                new NameComponent("ChatServer", "" );
                NameComponent path[] = { nameComponent };
                corbaObject = nameService.resolve( path );
                chatServer = ChatServerHelper.narrow( corbaObject );

                // Register this object with both the name service and
                // the chat server
                POA rootpoa = POAHelper.narrow(
                        orb.resolve_initial_references("RootPOA"));
                rootpoa.the_POAManager().activate();

                org.omg.CORBA.Object ref =
                    rootpoa.servant_to_reference(this);
                ChatClient meIOR = ChatClientHelper.narrow(ref);

                NameComponent path2[] = nameService.to_name( name );
                nameService.rebind(path2, meIOR);

                chatServer.register( name );

                // Use a separate thread to start the orb
                new Thread(this);
            } catch (Exception e) {
                System.out.println ("Unable to connect to chat server");
                System.out.println (e);
            }
        }
    }
```

Clients need to register with both the name service and the chat server.

Lines	Commentary
233–236	Used to create a new thread for the ORB, so that the main thread does not block. Called when the new thread is started on line 273.
245–255	Resolve the name service and chat server to CORBA objects.
257–270	Carry out the registrations.
273	Start the ORB running in a new thread.
233–236	Clients also need to call `orb.run()`, as the server did, so they can receive messages. `orb.run()` executes in the current thread and blocks, however. Because clients need this thread to continue their own processing, so for example, they can originate messages, a separate thread is started.

The complete listing for the chat server is found on the textbook's web site.

6.11 Conclusion

This chapter introduced you to distributed programming and some of the basic paradigms, e.g. client–server and peer–to–peer, that are frequently used when building distributed applications. The distributed infrastructure we used to develop our examples was CORBA. There are certainly many other infrastructures available. In the next several chapters we will continue our discussion of distributed event based programming, but using the World Wide Web as the infrastructure.