Lecture 6
Case Studies of High-Level IPC Facilities
(06-25,29,30-2009)

Lecture Outline

```
I.   SUN RPC networks
   1. Introduction and history
   2. The RPC programming guide
   3. rpcgen: a compiler for RPC
   4. RPC protocol specification
II.  Java RMI (Chapter 5)
   1. Introduction
   2. Tasks in writing an RMI client-server distributed
      application
   3. Example 2: the ‘‘ShapeList" RMI in Sect.5.5 of textbook.
   4. Example 3: the ‘‘Weather information server"
III. DCE RPC
   1. Overview
   2. Using DCE RPC interface
   3. Writing clients
   4. Pointers and arrays
   5. Writing servers
```

I. SUN RPC

1. Introduction and history

   (1) First commercial implementation of RPC

   (2) Illustration of RPC semantics (Page 1, RPC Prog. Guide)

2. The RPC Programming Guide

   (1) Levels of RPC – three levels

   a. The higher level: totally transparent about OS, machine, and networks. System calls for RPC are provided to users as ordinary system calls

   b. The intermediate level: a collection of system routines for defining and writing RPC application programs are provided. Among them are *registerrpc(), callrpc()*, and *svc_run*(). *registerrpc*() obtains a unique system-wide identification number, while *callrpc*() actually calls a RPC.
   Not flexible, for instance cannot choose transport protocol, time-out intervals.

c. Lower level: allows control of many lower-level details, such as transport protocol, time-out intervals.

(2) Higher Level (p.3, RPC Prog. Guide)

a. **Example**. writing a program that counts the number of users that are logged in on a remote machine. This can be done using the RPC service library routine *rnusers*.

b. The library function *rnusers*() is called within the client program.

c. The program should be compiled with -lrpcsvc flag. The library librpcsvc.a contains several other functions: *rusers*() (return information about users on a remote machine), *havedisk*() (determine if a remote machine has disk). *rstats*() (get performance data from remote kernel), *rwall*() (write to specified remote machines).

d. Some of RPC services such as *ether*(), *rquota*(), and *spray*(), cannot be used as library routines. They can be used from the *callprc*() function (to be introduced next).

(3) The Intermediate Level (p.4-9, RPC Prog. Guide)

a. The example client expressed using intermediate level calls: p.4

(a) The *callrpc*() call takes eight parameters.

· The first is remote host name.

· The 2nd, 3rd, and 4th are program number (RUSERSPROG in the eg.), program version number (RUSERSVER), and procedure number (RUSERSPROC_NUM) respectively.

· The 5th and 6th are a pair. The 5th is an XDR filter (type handler) for the input parameter, which is the 6th parameter.

· The 7th and 8th are another pair. The 7th is an XDR filter for the output parameter, which is the 8th parameter. In the example, the 8th parameter is a pointer type because it carries return value.

(b) If more than one value is needed for input, a built-in or user-defined type with corresponding XDR filters can be used.

(c) XDR (External Data Representation) provides a uniform type conversion mechanism (more to be covered next): **RPC can handle arbitrary data types/structures, regardless of different machines' byte order and structure layout conventions, by always converting them to a network standard called *External Data Representation* (XDR) before sending them over the network**.

(d) Return values of *callrpc*(): 0 if successful, non-zero otherwise. Detailed error values are in <rpc/clnt.h>.

(e) The Transport protocol used by *callrpc*() is UDP. The timeout interval is un-adjustable. To be able to adjust the timeout and number of retries, lower level primitives are needed.

b. The example server: p.5

(a) The remote procedure: the code for computing the number of users is left unfinished.

(b) The main program calls two RPC routines: *registerprc*() and *svc_run*().

(c) The *registerprc*() routine registers a C procedure as corresponding to a given RPC procedure number. It six parameters. The first three are the program number. program version number, and procedure number. The 4th is the name of the local procedure (*nuser* eg.) that implements the remote procedure (*nusers* eg.). The 5th is the XDR filter of the input parameter (xdr_void eg.), and the 6th is the XDR filter of the output parameter (xdr_u_long eg.).

(d) Only UDP transport mechanism can use *registerrpc*(). So this call can be safely used with the *callrpc*() call.

(e) UDP can only deals with arguments and results less than 8k bytes in length.

(f) The *svc_run*() routine, which is the RPC library's remote procedure dispatcher. This function will call the remote procedure in response to RPC call messages. The dispatcher also takes care of decoding the remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

c. Program number assignments. Program numbers are assigned in group of 0x20000000 according to the chart shown on p.6:

(a) 0x0 to 0x1fffffff defined by SUN: this group is administered by SUN for well-known and well-defined popular applications. Customers can apply for program numbers of their new programs by contacting SUN, sending a compilable rpcgen ".x" file describing the user's protocol for each new program.

(b) 0x20000000 to 0x3fffffff defined by user: reserved for specific user applications, primarily used for debugging new programs.

(c) 0x40000000 to 0x5fffffff defined transient: is reserved for applications that generated program numbers dynamically.

(d) 0x60000000 to 0xffffffff reserved for future use/development.

(e) Program numbers and protocol specifications for standard SUN RPC services are described in the include files in /usr/include/rpcsvc dir.

(f) A complete list of all supported RPC services: Table 3-2, p.6-7.

d. Passing Arbitrary Data Types

(a) *Serializing and deserializing*: The action of converting from a particular machine representation to XDR format is called *serializing*. The reverse action

is called *deserializing*.

(b) Built-in type routines (XDR filters): 11 of them: xdr_int(), xdr_u_int(), xdr_long(), ...

(c) Notice: any xdr filter that is used with the *callrpc*() and *registerrpc*() calls can only have two parameters (they only pass two parameters to their XDR routines). Therefore although the routine *xdr_string*() exists, which takes three parameters, it cannot be used with *callprc*() and *registerrpc*(). *xdr_wrapstring*() has only two parameters, and thus is fine. In fact, *xdr_wrapstring*() calls *xdr_string*().

(d) User defined types and XDR routines: example 1: the type *simple*() structure p.7-8.

- The structure has two components: an integer and a short.
- The corresponding XDR filter is shown on page 8.
- Notice that *xdr_simple*() calls *xdr_int*() and *xdr_short*() respectively to convert the two components.

(e) Other prefabricated XDR filters (page 8): *xdr_array, xdr_vector, xdr_string, xdr_bytes, xdr_union, xdr_opaque, xdr_reference, xdr_pointer.*

(f) Sending variable length int arrays: utilize the prefabricated *xdr_array*() XDR filters.

- The type *varintarr*(): two components: the array and the length of the array (integer).
- The XDR filter *xdr_varintarr*(): it calls *xdr_array*(), which takes six parameters (XDR handle, the pointer to the 1st element, the pointer to the length, the maximum length constant, the size of each array element, and the xdr filter for individual element (xdr_int in this case)).

(g) Sending fixed length int array: utilize *xdr_vector* filter:

- No need to have a structure any more.
- The XDR filter *xdr_intarr*() calls *xdr_vector*(), which takes five parameters.

(h) Dealing with 32-bit unit problem: XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the example above involved characters instead of integers, each char would occupy 32 bits.

- The XDR filter *xdr_bytes*() is similar to *xdr_array*(), but it packs characters. This filter has four parameters, similar to the first four parameters of *xdr_array*() (An example is given in the section about XDR memory allocation).
- Dealing with null-terminated strings: the *xdr_string*() filter, which is the same as *xdr_bytes*(), except it only has the first three parameters (not have the 4th, which is the length parameter). On serializing, it calls the

C function *strlen*() to get length of a string. On deserializing, it creates a null-terminated char string.

- The final example: it calls the *xdr_simple*() filter previously defined, and built-in functions *xdr_string*() and *xdr_reference*(), which takes four parameters. The *xdr_reference*() filter can be easily replaced with the *xdr_simple*() in this example.

(4) The Lower Level

a. Problems with higher and intermediate levels:

(a) At the higher level, only a number of fixed functions are available.

(b) At the intermediate level, only UDP can be used as the transport protocol. UDP only allows 8k bytes of data.

(c) There is no support at higher or intermediate levels for allocating and freeing memory explicitly. Memory allocations and freeing are done implicitly (to be discussed next).

(d) There is no support for authentication on either the client or server side that allows verifications of credentials.

b. The server side of the *nusers*() program written with lower level calls: page 10.

(a) The *svcudp_create* call returns a UDP transport handle, which is used for receiving and replying to RPC messages (the *registerrpc* call in the intermediate level uses *svcudp_create* to get a UDP handle). The *svctcp_create* returns a TCP handle.

(b) The *svcupd_create* call takes one parameter, which should be socket descriptor. If you specify your own socket descriptor, it can be bound or unbound. If it is bound to a port by the user, the port numbers of *svcudp_create* and the corresponding call *clntupd_create* at client side must match.

(c) The special constant *RPC_ANYSOCK* asks the RPC library to create and open a socket on which to receive and reply RPC messages. The routine *svcudp_create*() and *clntudp_create*() will cause the library routines to *bind*() their socket if it is not bound yet.

(d) The call *pmap_unset*() will clean up the port mapper's tables for any entries left for the *nusers* program.

(e) The call *svc_register* registers the service's port number with the local portmapper service.

- This call takes five parameters: the transport handle, the program number, the version number, the program name, and the protocol type.
- A client can discover the server's port number by consulting the portmapper on the server's machine. This is done by giving a zero port number in *clntudp_create*() and *clnttcp_create*().

- Notes: The registration is done at the program level, not at the procedure level; Also, there is no XDR routines involved during registration process.

(f) The user routine *nuser*() must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by *nuser*() and *registerrpc*() automatically:

- The procedure *NULLPROC* returns with no results. This can be used as a simple test for detecting if the remote program is running;
- There is a check for in valid procedure numbers. The routine *svcerr_noproc*() will be called in response to invalid procedure number error.

(g) The remote procedure *nuser*() serializes the results and returns them to RPC caller via the routine *svc_sendreply*(), which takes three parameters. The first is the transport handle, the 2nd is the XDR routine, and the 3rd is a pointer to the data to be returned.

c. Handling input parameters in the server: the example procedure *RUSERPROC_BOOL* (page 12).

(a) This procedure passes a parameter, whose value is the number of users on a remote machine guessed by the caller. The caller responds TRUE or FALSE depending upon if the actual number of users is equal to the guessed value.

(b) The procedure will be another branch in the body of the *nuser*() program.

(c) The input parameter *nuserquery* is handled by the routine *svc_getargs*(), which takes three parameters: a transport handle, an XDR filter, and a pointer to the input value.

(d) The routine *svc_sendreply*() is still used to send a boolean value back to the caller.

d. Memory allocation with XDR

(a) XDR routines not only do input and output, they also do memory allocation.

- That is why the second parameter of *xdr_array* (it has 6 parameters) is a pointer to an array, rather than the array itself.
- If that parameter is NULL, then *xdr_array*() will allocate space to the array and return a pointer to it. It will also put the size of the array in the third parameter.

(b) An example: the *xdr_chararr1*() routine that deals with a fixed array of bytes with length *SIZE* (page 12).

- If the space has been allocated, the *xdr_chararr1*() can be used with the syntax:

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

- If the user wants XDR to allocate memory, the routine *xdr_chararr1* has to be rewritten (renamed to *xdr_chararr2*() shown on page 13).
- Notice the difference between *xdr_chararr1*() and *xdr_chararr2*(): The input parameter *chararrp* in *xdr_chararr2*() is now a pointer to a pointer of char string, while *chararr* in *xdr_chararr1* is a pointer to a char string.
- The RPC call now initializes the input parameter to have a NULL value.
- The routine *svc_freeargs*() free (deallocate) the memory pointed by its last parameter. This routine is smart and will not attempt free memory of the pointer parameter has NULL value.

(c) Summary of memory handling:

- Each XDR routine is responsible for serializing, de-serializing, allocating (if needed), and deallocating (freeing) memory;
- Serializing: when *callrpc*() is called;
- De-serializing: when *svc_getargs*() is called;
- Deallocating: when *svc_freeargs*() is called.

e. The calling side (client) of the *nusers* example:

(a) The code: page 14.

(b) Highlights:

- The socket structure *sockaddr_in* data type;
- The *gethostbyname* function;
- Initialization of the *server_addr* structure;
- The routine *clntudp_create*: will create a client handle for the client to send and receive messages. It takes five parameters: the socket structure pointer, program number, version number, a predefined timeout interval, and a pointer to a socket descriptor (it has *RPC_ANYSOCK* in this example).
- The actual remote procedure is called via the routine *clnt_call*(), which takes seven parameters: the client handle, the remote procedure number, the input XDR filter, the input parameter, the output XDR filter, the output parameter, and a timeout structure.
- Finally, the routine *clnt_destroy* is called with the client handle as the single parameter. That call deallocates the memory associated with the client handle. It closes the socket associated with the CLIENT handle only when the RPC library previously opened it. The socket will be left open if it was opened outside this function.
- For TCP client, the routine *clnttcp_create*() can be used. This call does not take timeout parameter. Instead it has an input buffer size and an output buffer size parameters.

· For TCP the server side uses the *svctcp_create*() call, which takes three parameters: the first is the socket descriptor. The last two are the send and receive sizes respectively. "0" for these two values that the system chooses a default value.

3. *rpcgen*: a compiler for RPC

(1) Introduction: what is and why a RPC compiler?

a. rpcgen is a compiler that automatically converts procedure arguments and return-results into their network formats and vice versa.

b. The details of programming applications to use RPC can be overwhelming. The most challenging task is writing the various XDR routines to convert parameters and results to their network format and vice versa.

c. rpcgen is a compiler and is intended as a tool that do most of the dirty work.

d. Input: remote program interface definition written in a language (named RPC language, similar to C).

e. Output: client stubs, skeleton of servers, XDR filter routines for both parameters and results, head files containing common definitions,

f. The client stubs interface with the RPC library and effectively the network from their callers. Similarly the server stub effectively hide the network from the server procedures that are to be called by remote clients. The XDR filter routines are generated automatically from the interface definition.

g. The rpcgen output files can be compiled and linked same as C programs.

* A server developer writes server procedures - in any language that observe SUN calling conventions - and links them with the server skeleton produced by rpcgen to produce an executable server program.
* To call a remote program, a user writes an ordinary main program that make local procedure calls to the client stubs produced by rpcgen. Linking this program with the rpcgen's stubs creates an executable client program. (At present, the main program can only be in C).

h. rpcgen options can be used to suppress stub generation and to specify the transport (UDP/TCP) to be used by the server stub.

i. Code generated by rpcgen may be less efficient than hand-written code. But one can mix these two types of code. In addition, code generated by rpcgen can be starting point to write good code.

(2) Converting local procedures to remote procedures: a local program that prints a message on the console of the local machine.

a. The code consists of the main program and the procedure *printmessage*() on page 2 of the rpcgen Prog. Guide.

b. How to convert such a local procedure to a remote a procedure so that one can call it to print a message on the console of a remote machine?

   (a) Analyze the program: the procedure is *printmessage*(), which takes a string as input, and returns an integer as output.

   (b) Knowing this info, we can write a protocol specification in RPC language that describes the remote version of *printmessage*() (page 3 the file *msg.x*).

   (c) The file *msg.x* actually declared a remote program (a remote procedure is part of a remote program). That remote program only contains a single procedure *PRINTMESSAGE*. This procedure is declared to be in version 1 of the remote program.

   (d) No null procedure (procedure 0) is necessary because rpcgen generates it automatically. Also notice that everything is capital case – this is not required but is a good convention to follow.

   (e) The argument is type *string*, not *char* $*$, because the latter is used in C.

c. Two more things to write to complete the conversion: The remote procedure itself, and the client program that will call the remote procedure.

   (a) The remote procedure itself: page 3, *msg_proc.c*. The remote procedure *printmessage_1*() differs from its local version in three ways:

      i. It takes a pointer to a string instead of the string itself as the single parameter. This is true for all remote procedures: they always take pointers to their arguments instead of arguments themselves.

      ii. It returns a pointer to an integer instead of an integer itself. This is also true for all remote procedures.

      iii. It has a "_1" suffix appended to its name. In general all remote procedures called by rpcgen are named by the following rule: the name in the program definition (*PRINTMESSAGE* here) is converted to all lower-case letters, an underscore letter "_" is appended to it, and finally the version number is appended (here is 1).

   (b) The main client program that calls the remote procedure: page 5-6: *rprintmsg.c*.

      i. The call *clnt_create*() from the RPC library routine creates a client "handle", which will be passed to the client stub routines which calls the remote procedure.

      ii. The remote procedure *printmessage_1* is called exactly the same way as it is declared in *msg_proc.c*, except for the inserted client handle as the first parameter.

d. How to compile the server and the client programs: two main programs are compiled.

(a) % rpcgen msg.x

(b) % cc rprintmsg.c msg_clnt.c -o rprintmsg

(c) % cc msg_proc.c msg_svc.c -o msg_server

e. *rpcgen* did the following to the input file *msg.x*:

(a) Created a header file *msg.h* that contains #define's for *MESSAGEPROG, MESSAGEVERS*, and *PRINTMESSAGE* for use in other modules;

(b) Created client "stub" routines in the *msg_clnt.c* file. In this case there is only one, *printmessage_1*. Names for client stubs always has the form FOO_clnt.c for an input file FOO.x.

(c) Created server program that calls *printmessage_1*() in *msg_clnt.c*. The server program is named as *msg_svc*. Names for server output files always has the form FOO_svc.c for an input file FOO.x.

f. Running the server and client

(3) Generating XDR routines: rpcgen can also generate the XDR routines, besides the client and server RPC code.

a. An example: a remote dir listing service. Will use rpcgen to generate the server/client and the XDR routines.

b. The protocol specification file *dir.x* on page 7.

– Running rpcgen on dir.x produces four files: The header file, client stub routines, and server skeleton. In addition, the fourth, named *dir_xdr.c*, are the XDR routines that are needed for converting the data types.

c. The implementation of the *READDIR* procedure *readdir_1*(): page 8.

d. The implementation of the client side program *rls.c* procedure: page 8.

e. Note: The client program and and the server procedure can be tested together on a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with debuggers such dbx.

(4) The C-Preprocessor

a. The C-preprocessor is run all input files before they are compiled by rpcgen. So all the preprocessor directives are legal within the ".x" file

b. Four symbols may be defined, depending upon which output file is getting generated: RPC_HDR for header file output; RPC_XDR for XDR routine output; RPC_SVC for server skeleton output; and RPC_CLNT for client stub output

c. Any line beginning with a percentage sign % is ignored. (not a recommended practice).

4. RPC protocol specification

(1) Introduction

a. The SUN RPC protocol

b. Transport and semantics: the RPC protocol is independent of the underlying transport protocol, i.e. it does not care how messages are passed from one process to another process.

(2) RPC Protocol Requirements

a. Three fundamental requirements:

* Unique specification of a procedure to be called;
* Provisions for matching response messages to request messages;
* Provisions for authenticating the caller to service and vice-versa.

b. Other supported features:

* RPC protocol mismatch checking;
* Remote program protocol version mismatch checking;
* Protocol errors detection;
* Notification of reasons of authentication failures;
* Notification of any other reasons why a desired procedure was not called.

c. Programs and procedures

* A RPC call message carries three unsigned fields: remote program number, remote program version number, and the remote procedure number. These three quantities uniquely identify the procedure to be called.
* The RPC protocol version number: the currently version is 2. This protocol version number may also evolve with time.
* The reply message contains enough info to distinguish the following five error conditions:
  - RPC protocol version mismatch;
  - The remote program is not available on the remote system;
  - Remote program version number mismatch;
  - The requested procedure number not available;
  - The parameters passed cannot be understood by the remote server.

d. Authentication

* A call message has two authentication fields: the credentials and the verifier. The reply message has one authentication field: the response verifier. The RPC protocol specification defines the all three fields to be the following opaque type (p.3).

* Any *opaque_auth* structure is an *auth_flavor* enumeration followed by bytes which are opaque to the RPC protocol implementation.
* The interpretation and semantics of the data contained within the authentication field is specified by individual, independent of authentication protocol specification.
* If authentication parameters were rejected, the response message contains info stating why they were not accepted.

e. Program number assignment (already covered in RPC PROG. GUIDE)

(3) RPC Message Protocol (p.5-8).

II. Java RMI

1. Introduction

(1) Java RMI is an example implementation of the RMI concept presented in last lecture.

a. As an RMI, Java RMI extends the RPC mechanism to provide RMI for distributed objects.

b. Java supports basic TCP/IP networking API, as discussed in Lecture 5.2. Java RMI is a higher level API than Java TCP/IP API, just like SUN RPC and DCE RPC are higher level API than Berkeley socket TCP/IP API.

c. Java RMI (since JDK 1.1) was designed with the following main goals:

(a) Support seamless remote invocation on objects in different JVM.

(b) Support callbacks from servers to applets.

(c) Integrate the distributed object model into the Java programming language in a natural way while retaining most of the Java language's object semantics.

(d) Provide a transparent interface of distributed object model and make writing reliable distributed applications as simple as possible.

(e) Maintain the safe environment of the Java platform provided by security managers and class loaders.

d. Like SUN RPC which provides a compiler *rpcgen*, Java RMI provides a RMI compiler *rmic* that will generate *stub* files.

(2) Preliminaries of Java RMI

a. Overview of Java RMI interfaces and classes (Fig.5.17, p.215)

b. In Java RMI, a remote interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine. Java RMI's remote interface package is java.rmi.Remote.

(a) The special interface java.rmi.Remote is a so called *tagging interface*. It has no methods declared inside and it will *tag* any interface that extends it as *remote*, meaning that methods in that interface become remote methods.

(b) Each remote method must be defined in an *remote interface* that *extends* the *Remote* interface in java.rmi.∗.

(c) Example: the BankAccount interface.

```
import java.rmi.*

public interface BankAccount extends Remote {
   public void deposit(float amount)
```

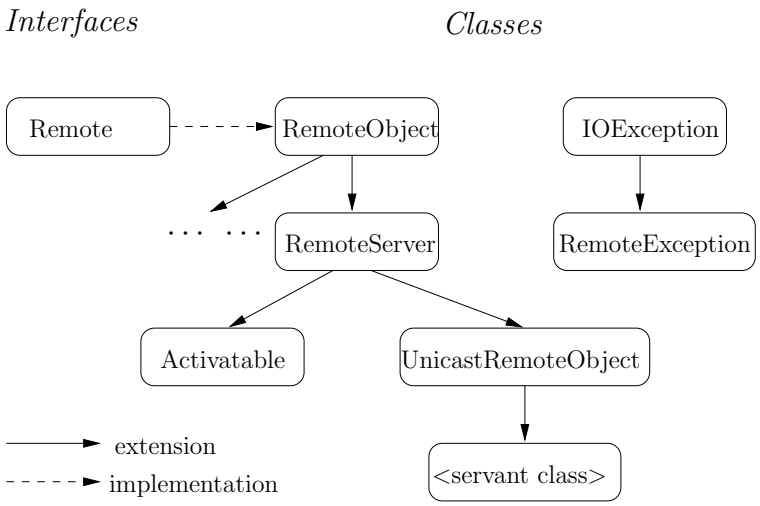

Figure 31: Overview of Java RMI interfaces and classes

```
            throws RemoteException;
    public void withdraw(float amount)
            throws OverdrawnException, RemoteException;
    public float getBalance()
            throws RemoteException;
}
```

c. A remote interface only provides definitions of a set of methods that are remotely invocable. Each remote interface must be implemented by a remote object that either *extends* the class *UnicastRemoteObject* or the class *Activable*:

   (a) The java.rmi.server.UnicastRemoteObject class defines a singleton (unicast) remote object whose references are valid only while the server process is alive.

   (b) The class java.rmi.activation.Activatable is an abstract class that defines an activatable remote object that starts executing when its remote methods are invoked and can shut itself down when necessary.

d. Each remote object may support several remote methods (servers).

   (a) Each remote method must be part of an interface that extends java.rmi.Remote

   (b) Each remote method in a remote interface must have a *throws* clause that indicates possibilities of a *RemoteException*.

   (c) The class of *RemoteException* is from the package java.IOException

e. Each RMI supporting computer runs a *RMIrgistrtry* (with executable name *rmiregistry*) daemon process at port 1099. This port number is the default (i.e. a well-known port number) for RMI service requests. Each remote server must *bind* its remote object name with its server object instance in RMIregistry.

f. Remote server object names in RMI are specified in the following format:

//Hostname[:Port]/RemoteServer

which is similar to the URL format.

(a) The *Hostname* component specifies the IP name of the computer hosting an remote server object. The *Port* component is optional.

(b) *RemoteServer* is the name of the remote server object.

(c) Both the remote server and an RMI client must specify the remote server object name in this format.

i. A remote server object declares this name using a construct like:

Naming.bind("//Hostname/RemoteServer", obj)

or

Naming.rebind("//Hostname/RemoteServer", obj)

to bind the remote object name to a remote object in rmiregistry.

ii. A remote client uses the *Naming.lookup* method to locate the remote server object:

Naming.lookup("//Hostname/RemoteServer")

2. Tasks in writing an RMI client-server distributed application (from Java RMI manual):

(1) Three tasks in writing an RMI client-server distributed application (from Java RMI manual)

a. Define an RMI interface that lists functions (methods) of the remote class

b. Write the implementation and server classes

c. Write a client program that uses the remote services

(2) Define a RMI interface

a. In creating a client-server distributed application using RMI, the first step is to define a *remote interface* that describes the *remote methods* that the client can use to access *remote objects* managed by the remote methods in the remote server through RMI.

b. This is done by declaring the interface under design as **Remote**. Example 1:

```
package examples.hello;

import java.rmi.Remote;
```

```
import java.rmi.RemoteException;

public interface Hello extends Remote {
   String sayHello() throws RemoteException;
}
```

(3) Write the implementation and server classes. A remote object implementation should contain at least three parts:

   a. Implementation of a remote interface previously defined

   b. Definition of constructor(s) for the remote object

   c. Implementation of each remote method

```
public class HelloImpl extends UnicastRemoteObject
   implements Hello {

   public HelloImpl() throws RemoteException {
      super();
   }

   public String sayHello() throws RemoteException {
      return "Hello World!";
   }

   public static void main (String args[]) throws Exception {

      System.err.println ("Initializing hello server, please wait ...");

      // create an instance of the remote object
      HelloImpl obj = new HelloImpl();

      // bind HelloImpl object to rmiregistry
      Naming.rebind("//myhost/HelloServer", obj);
   }
}
```

(4) Write a client program that uses the remote services

```
package examples.hello;
```

```
import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {

   String message = "blank";

   // "obj" is the identifier that we'll use to refer
   // to the remote object that implements the "Hello"
   // interface
   Hello obj = null;

   public void init() {
      try {
         obj = (Hello)Naming.lookup("//" +
                getCodeBase().getHost() + "/HelloServer");
         message = obj.sayHello();
      } catch (Exception e) {
         System.out.println("HelloApplet exception: " + e.getMessage());
         e.printStackTrace();
      }
   }

   public void paint(Graphics g) {
      g.drawString(message, 25, 50);
   }
}
```

(5) Compile/run the server and client: see the on-line manual for Java RMI for details and explanations as how to compile and run the program.

3. Example 2: the "ShapeList" RMI in Sect.5.5.1 of textbook

(1) The Java interface *Shape* and *ShapeList* (Fig.5.12, p.209)

```
import java.rmi.*;
import java.util.Vector;
```

```
public interface Shape extends Remote {
   int getVersion() throws RemoteException;
   GraphicalObject getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
   Shape newShape(GraphicalObject g) throws RemoteException;
   Vector allShapes()throws RemoteException;
   int getVersion() throws RemoteException;
}
```

a. Remote interfaces are defined by extending an interface called *Remote* Provided by *java.rmi* package. Each remote method listed in a remote interface must throw *RemoteException*, and it can throw other application related exceptions as well.

b. The *Shape* interface

   (a) GraphicalObject is a class that holds the state of a graphical object (its type, position, enclosing rectangle, line color, fill color) and provides operations for accessing and updating its state.

   (b) GraphicalObject must implement the *Serializable* interface as it is used in RMI.

   (c) The method *getVersion* returns an integer version number of the interface, while the method *getAllState* returns an instance of the *GraphicalObject* class.

c. The *ShapeList* interface

   (a) The only new remote method in this interface is *newShape*, which defines a new shape of a graphical object. This method requires an instance of the *GraphicalObject* as argument and returns an object with a remote interface (ie a remote object).

d. Both ordinary and remote objects can appear as arguments or results in a remote interface. In the latter case (when a remote object is used as return value), the remote object return value is always denoted by their remote interface.

(2) The server Java class *ShapeListServer* of with main method (Fig.5.14, p.212).

```
import java.rmi.*;
public class ShapeListServer {
   public static void main(String args[]){
```

```
      System.setSecurityManager(new RMISecurityManager());
      System.out.println("Main OK");
      try{
         ShapeList aShapelist = new ShapeListServant();
         Naming.rebind("ShapeList", aShapelist);
         System.out.println("ShapeList server ready");
      }catch(Exception e) {
         System.out.println("ShapeList server main " + e.getMessage());}
   }
}
```

(3) The *ShapeListServant* class that implements *ShapeList* interface (Fig.5.15, p.212).

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements
                                                    ShapeList{
   private Vector theList;
   private int version;

   public ShapeListServant()throws RemoteException{
      theList = new Vector();
      version = 0;
   }

   public Shape newShape(GraphicalObject g) throws RemoteException{
      version++;
      Shape s = new ShapeServant(g, version);
      theList.addElement(s);
      return s;
   }

   public  Vector allShapes()throws RemoteException{
      return theList;
   }

   public int getVersion() throws RemoteException{
        return version;
   }
```

```
}
```

(4) The Java client class that invokes methods in interface *ShapeList* (Fig.5.16, p.213).

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class ShapeListClient{
   public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager());
      ShapeList aShapeList = null;
      try{
         aShapeList =(ShapeList)
                   Naming.lookup("//bruno.ShapeList");
         Vector sList = aShapeList.allShapes();
      } catch(RemoteException e)
         {System.out.println("allShapes: " + e.getMessage());
      } catch(Exception e)
         {System.out.println("Client: " + e.getMessage());}
   }
}
```

4. Example 3: the "Weather information server"

(1) Program description. Copyright notice: this example is from Chapter 13 of Prentice-Hall's book *Advanced Java 2 Platform: How to Program* by Deitel, Deitel, & Santry , ISBN 0-13-089560-1, 2002.

a. The server, `WeatherServiceImpl`, upon startup, will contact the HTTP server at

http://iwin.nws.noaa.gov/iwin/us/traveler.html

to obtain the weather information.

b. A client will first contact the server obtain the weather information. It then displays the information in a window with icons for different types of weather conditions.

(2) Interface definition for interface *WeatherService* (Fig.13.1, p.793).

a. As discussed previously, the interface **WeatherService** has to extend the tagged interface **Remote**;

b. The interface declares the only remote method **getWeatherInformation**;

c. As discussed earlier, the interface does not provide the implementation of the remote method.

```
1  // WeatherService.java
2  // WeatherService interface declares a method for obtaining
3  // wether information.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.rmi.*;
8  import java.util.*;
9
10 public interface WeatherService extends Remote {
11
12    // obtain Vector of WeatherBean objects from server
13    public List getWeatherInformation() throws RemoteException;
14
15 }
```

(3) The server implementation class **WeatherServiceImpl** that implements the remote interface (Fig.13.2, p.794-796)

a. The server implementation class *extends* **UnicastRemoteObject** (from package **java.rmi.server**). The class **UnicastRemoteObject** provides the basic functions required for all remote objects:

(a) The constructors of a class that extends **UnicastRemoteObject** *exports* the remote object to make it available to receive remote invocations;

(b) *Exporting* the object enables the remote object to wait for client requests on an anonymous port number (chosen internally by the kernel of the client computer). This enables the remote object to communicate with any client through *unicast communication* (i.e. point-to-point communication);

(c) The Java RMI implementation abstracts away all the above communication details from programmers.

· The constructor (Lines 19-23) invokes the default constructor for class **UnicastRemoteObject** at line 21.

· The class **UnicastRemoteObject** has overloaded constructors that allow programmers to specifiy additional information such as an explicit port number on which to export the remote object.

(d) All **UnicastRemoteObject** constructors must throw a **RemoteException** (line 19).

(e) constructor invokes the private method **updateWeatherConditions** (line 22) to obtain the weather conditions for the current running.

b. The service only contact the HTTP server once at start up: Line 118, the invocation of the class constructor, which in turn calls the method **updateWeatherConditions** at line 22.

c. The method **updateWeatherConditions** (lines 26-91) contacts the NWS web site, reads weather information, and stores the weather information in a **List** of **WeatherBean** objects:

   (a) Lines 32-33 creates a **URL** object for the *Traveler's Forcast* web page.

   (b) Lines 36-37 invoke the **openStream** method of the **URL** class to open a connection to the specified **URL** and wrap that connection in an object of class **BufferedReader**.

   (c) The server depends on the structure of the HTML page to fetch the needed information: Lines 40-59, and the while loop at lines 70-85.

   (d) Lines 65-85 read each city's weather information and place it in **WeatherBean** objects:

      · Each **WeatherBean** object contains three pieces of information: the city name, the tempature, and the description of weather condition.

      · Line 61 creates a **List** object for storing **WeatherBean** objects.

      · Lines 76-79 construct a **WeatherBean** object for the current city.

      · Line 82 adds the new **WeatherBean** object to the **List**.

d. The method **getWeatherInformation** (lines 107-110) implements the remote method listed in the remote interface. This method is invoked in response to every RMI request.

d. The **main** method (lines 113-127):

   (a) It first creates the **WeatherServiceImpl** object (line 118). The creation of this object exports the remote object.

   (b) Line 121 creates a URL for the remote object:

   rmi://localhost/WeatherService

   which is the same as

   rmi://127.0.0.1/WeatherService

   (c) Line 124 invokes the **static** method **rebind** of class **naming** to bind the a remote object to the Java RMI registry.
   The **rebind** method ensures that if the remote object has already been registered under the given name, the new remote object will replace the previoulsy registered name.

```
1  // WeatherServiceImpl.java
2  // WeatherServiceImpl implements the WeatherService remote
3  // interface to provide a WeatherService remote object.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.io.*;
8  import java.net.URL;
9  import java.rmi.*;
10 import java.rmi.server.*;
11 import java.util.*;
12
13 public class WeatherServiceImpl extends UnicastRemoteObject
14    implements WeatherService {
15
16    private List weatherInformation;  // WeatherBean objects
17
18    // initialize server
19    public WeatherServiceImpl() throws RemoteException
20    {
21      super();
22      updateWeatherConditions();
23    }
24
25    // get weather information from NWS
26    private void updateWeatherConditions()
27    {
28       try {
29          System.err.println( "Update weather information..." );
30
31          // National Weather Service Travelers Forecast page
32          URL url = new URL(
33             "http://iwin.nws.noaa.gov/iwin/us/traveler.html" );
34
35          // set up text input stream to read Web page contents
36          BufferedReader in = new BufferedReader(
37             new InputStreamReader( url.openStream() ) );
38
39          // helps determine starting point of data on Web page
40          String separator = "TAV12";
41
42          // locate separator string in Web page
43          while ( !in.readLine().startsWith( separator ) )
```

```
44            ;    // do nothing
45
46         // strings representing headers on Travelers Forecast
47         // Web page for daytime and nighttime weather
48         String dayHeader =
49            "CITY            WEA     HI/LO   WEA     HI/LO";
50         String nightHeader =
51            "CITY            WEA     LO/HI   WEA     LO/HI";
52
53         String inputLine = "";
54
55         // locate header that begins weather information
56         do {
57            inputLine = in.readLine();
58         } while ( !inputLine.equals( dayHeader ) &&
59                   !inputLine.equals( nightHeader ) );
60
61         weatherInformation = new ArrayList(); // create List
62
63         // create WeatherBeans containing weather data and
64         // store in weatherInformation Vector
65         inputLine = in.readLine();  // get first city's info
66
67         // The portion of inputLine containing relevant data is
68         // 28 characters long. If the line length is not at
69         // least 28 characters long, done processing data.
70         while ( inputLine.length() > 28 ) {
71
72            // Create WeatherBean object for city. First 16
73            // characters are city name. Next, six characters
74            // are weather description. Next six characters
75            // are HI/LO or LO/HI temperature.
76            WeatherBean weather = new WeatherBean(
77               inputLine.substring( 0, 16 ),
78               inputLine.substring( 16, 22 ),
79               inputLine.substring( 23, 29 ) );
80
81            // add WeatherBean to List
82            weatherInformation.add( weather );
83
84            inputLine = in.readLine();  // get next city's info
85         }
86
```

```
87          in.close();  // close connection to NWS Web server
88
89          System.err.println( "Weather information updated." );
90
91       }
92
93       // process failure to connect to National Weather Service
94       catch( java.net.ConnectException connectException ) {
95          connectException.printStackTrace();
96          System.exit( 1 );
97       }
98
99       // process other exceptions
100       catch( Exception exception ) {
101          exception.printStackTrace();
102          System.exit( 1 );
103       }
104    }
105
106    // implementation for WeatherService interface method
107    public List getWeatherInformation() throws RemoteException
108    {
109       return weatherInformation;
110    }
111
112    // launch WeatherService remote object
113    public static void main( String args[] ) throws Exception
114    {
115       System.err.println( "Initializing WeatherService..." );
116
117       // create remote object
118       WeatherService service = new WeatherServiceImpl();
119
120       // specify remote object name
121       String serverObjectName = "rmi://localhost/WeatherService";
122
123       // bind WeatherService remote object in rmiregistry
124       Naming.rebind( serverObjectName, service );
125
126       System.err.println( "WeatherService running." );
127    }
128 }
```

(4) The *WeatherBean* utility class (Fig.13.3,p.799,800): stores data the class **Weath-**

erServiceImpl retrieves from the National Weather Service Web site.

a. The class stores the city, temprature, and weather description as **String**s.

b. Lines 25-45 load a property file that contains image names for displaying the weather information. The block (lines 25-45) is **static** to ensure that the image names are available as soon as the java VM loads the **WeatherBean** class into memory. Namely the image names will be available to any instance of **WeatherBean** class.

c. Lines 64-85 provide three *get* methods for each piece of information.

```
1  // WeatherBean.java
2  // WeatherBean maintains weather information for one city.
3  package com.deitel.advjhtp1.rmi.weather;
4
5  // Java core packages
6  import java.awt.*;
7  import java.io.*;
8  import java.net.*;
9  import java.util.*;
10
11 // Java extension packages
12 import javax.swing.*;
13
14 public class WeatherBean implements Serializable {
15
16    private String cityName;         // name of city
17    private String temperature;      // city's temperature
18    private String description;      // weather description
19    private ImageIcon image;         // weather image
20
21    private static Properties imageNames;
22
23    // initialize imageNames when class WeatherInfo
24    // is loaded into memory
25    static {
26       imageNames = new Properties(); // create properties table
27
28       // load weather descriptions and image names from
29       // properties file
30       try {
31
32          // obtain URL for properties file
33          URL url = WeatherBean.class.getResource(
```

```
34             "imagenames.properties" );
35
36          // load properties file contents
37          imageNames.load( new FileInputStream( url.getFile() ) );
38       }
39
40       // process exceptions from opening file
41       catch ( IOException ioException ) {
42          ioException.printStackTrace();
43       }
44
45    } // end static block
46
47    // WeatherBean constructor
48    public WeatherBean( String city, String weatherDescription,
49       String cityTemperature )
50    {
51       cityName = city;
52       temperature = cityTemperature;
53       description = weatherDescription.trim();
54
55       URL url = WeatherBean.class.getResource( "images/" +
56          imageNames.getProperty( description, "noinfo.jpg" ) );
57
58       // get weather image name or noinfo.jpg if weather
59       // description not found
60       image = new ImageIcon( url );
61    }
62
63    // get city name
64    public String getCityName()
65    {
66       return cityName;
67    }
68
69    // get temperature
70    public String getTemperature()
71    {
72       return temperature;
73    }
74
75    // get weather description
76    public String getDescription()
```

```
77    {
78       return description;
79    }
80
81    // get weather image
82    public ImageIcon getImage()
83    {
84       return image;
85    }
86 }
```

(5) The client class *WeatherServiceClient* (Fig.13.4,p.801,802):

a. This class defines the client application that retrieves weather information from the remote object **WeatherServiceImpl**.

(a) The client invokes remote method **getWeatherInformation** (line 32) of the interface **WeatherService** to obtain weather information.

(b) The class uses a **JList** with a customerized **ListCellRenderer** to display weather info (lines 73-76).

b. The main method (lines 61-77):

(a) It first checks whether there is a command-line parameter (the hostname of the remote method hosting computer): lines 67-70.

(b) It then sets display properties through three method invocations: lines 73-75.

(c) It invokes the method **setVisible** to display the weather information: line 76.

c. The constructor: lines 16-58.

(a) It first invokes the default constructor of the class **JFrame** (the **super** method at line 18).

(b) Line 24 creates a URL string for the remote object to be invoked.

(c) Line 27,28 invokes the **static** method **lookup** of the class **Naming** (which is in the **java.util— package**). This method connects to the RMI registry and returns of a **Remote** reference to the remote object identified by the URL string. As this remote object is known to be **WeatherSerice** object, line 28 type casts the returned reference to type **WeatherService**.

· Question: which RMI registry does the **lookup** method connect to? The local one, or the one at the remote method hosting computer?

· The obtained remote object reference can then be used as if it referred a local object. In fact, this remote object reference refers to a *stub* object in the client.

(d) Lines 31, 32 invoke the remote method **getWeatherInformation** on the remote reference.

- The remote object invocation will result in a copy of the **List** of **WeatherBeans** objects.
- Because parameters and returned objects of a remote method must be *marshalled* and *unmarshalled*, also termed *serialized* and *unserialized*, they must be declared as **Serializable**.

(e) Lines 35, 36 creates a **WeatherListModel** object by calling the **WeatherListModel** method to facilitate the display of the weather info in a **JList** (line 39). Line 40 sets up a **ListCellRenderer** for the **JList**. Line 41 invokes the **add** method to actually display the **JList**.

```
1  // WeatherServiceClient.java
2  // WeatherServiceClient uses the WeatherService remote object
3  // to retrieve weather information.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.rmi.*;
8  import java.util.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class WeatherServiceClient extends JFrame
14 {
15    // WeatherServiceClient constructor
16    public WeatherServiceClient( String server )
17    {
18       super( "RMI WeatherService Client" );
19
20       // connect to server and get weather information
21       try {
22
23          // name of remote server object bound to rmi registry
24          String remoteName = "rmi://" + server + "/WeatherService";
25
26          // lookup WeatherServiceImpl remote object
27          WeatherService weatherService =
28             ( WeatherService ) Naming.lookup( remoteName );
29
30          // get weather information from server
```

```
31          List weatherInformation = new ArrayList(
32             weatherService.getWeatherInformation() );
33
34          // create WeatherListModel for weather information
35          ListModel weatherListModel =
36             new WeatherListModel( weatherInformation );
37
38          // create JList, set ListCellRenderer and add to layout
39          JList weatherJList = new JList( weatherListModel );
40          weatherJList.setCellRenderer( new WeatherCellRenderer() );
41          getContentPane().add( new JScrollPane( weatherJList ) );
42
43       } // end try
44
45       // handle exception connecting to remote server
46       catch ( ConnectException connectionException ) {
47          System.err.println( "Connection to server failed. " +
48             "Server may be temporarily unavailable." );
49
50         connectionException.printStackTrace();
51       }
52
53       // handle exceptions communicating with remote object
54       catch ( Exception exception ) {
55          exception.printStackTrace();
56       }
57
58    } // end WeatherServiceClient constructor
59
60    // execute WeatherServiceClient
61    public static void main( String args[] )
62    {
63       WeatherServiceClient client = null;
64
65       // if no sever IP address or host name specified,
66       // use "localhost"; otherwise use specified host
67       if ( args.length == 0 )
68          client = new WeatherServiceClient( "localhost" );
69       else
70          client = new WeatherServiceClient( args[ 0 ] );
71
72       // configure and display application window
73       client.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

```
74       client.pack();
75       client.setResizable( false );
76       client.setVisible( true );
77    }
78 }
```

(6) The classes **WeatherListModel** (Fig.13.5,p.803,804), *WeatherCellRenderer* (Fig.13.6, p.805) and *WeatherItem* (Fig.13.7, p.805,806)

a. The class *WeatherListModel* (Fig.13.5,p.803,804)

(a) This class is a **ListModel** implementation. An object of this class contains **WeatherBean**s to be displayed in a **JList**.

(b) This is a so called *adapter* class.

- In Java the interface **List** is not compatible with the interface **JList**. A **JList** can retrieve elements only from a **ListModel** object.
- This class adapts interface **List** to make it compatible with **JList**'s interface.

(c) Specifically, this class adapts as follows:

- When the **JList** invokes **WeatherListModel** method **getSize**, **WeatherListModel** in turn invokes the method **Size** of the interface **List**.
- When the **JList** invokes **WeatherListModel** method **getElementAt**, **WeatherListModel** in turn invokes the method **get** of the interface **List**.

and so on.

(d) The class also plays the role of the model in the Swing's delegate-model architecture (c.f. Fig.3.1 for illustration).

```
1  // WeatherListModel.java
2  // WeatherListModel extends AbstractListModel to provide a
3  // ListModel for storing a List of WeatherBeans.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.util.*;
8
9  // Java extension packages
10 import javax.swing.AbstractListModel;
11
12 public class WeatherListModel extends AbstractListModel {
13
14    // List of elements in ListModel
```

```
15    private List list;
16
17    // no-argument WeatherListModel constructor
18    public WeatherListModel()
19    {
20       // create new List for WeatherBeans
21       list = new ArrayList();
22    }
23
24    // WeatherListModel constructor
25    public WeatherListModel( List itemList )
26    {
27       list = itemList;
28    }
29
30    // get size of List
31    public int getSize()
32    {
33       return list.size();
34    }
35
36    // get Object reference to element at given index
37    public Object getElementAt( int index )
38    {
39       return list.get( index );
40    }
41
42    // add element to WeatherListModel
43    public void add( Object element )
44    {
45       list.add( element );
46       fireIntervalAdded( this, list.size(), list.size() );
47    }
48
49    // remove element from WeatherListModel
50    public void remove( Object element )
51    {
52       int index = list.indexOf( element );
53
54       if ( index != -1 ) {
55          list.remove( element );
56          fireIntervalRemoved( this, index, index );
57       }
```

```
58
59    } // end method remove
60
61    // remove all elements from WeatherListModel
62    public void clear()
63    {
64       // get original size of List
65       int size = list.size();
66
67       // clear all elements from List
68       list.clear();
69
70       // notify listeners that content changed
71       fireContentsChanged( this, 0, size );
72    }
73 }
```

b. The class **WeatherCellRenderer** (Fig.13.6, p.805):

(a) The class **JList** uses the class **ListCellRenderer** to render each element in that **JList**'s **ListModel**.

(b) The class **WeatherCellRenderer** *extends* the **DefaultListCellRenderer** (i.e. it is a subclass of the latter) to render **WeatherBean**s in a **JList**.

(c) The only method **getListCellRendererComponent** creates and returns a **WeatherItem** object for the given **WeatherBean**.

```
1  // WeatherCellRenderer.java
2  // WeatherCellRenderer is a custom ListCellRenderer for
3  // WeatherBeans in a JList.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.awt.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class WeatherCellRenderer extends DefaultListCellRenderer {
13
14    // returns a WeatherItem object that displays city's weather
15    public Component getListCellRendererComponent( JList list,
16       Object value, int index, boolean isSelected, boolean focus )
17    {
18       return new WeatherItem( ( WeatherBean ) value );
```

```
19    }
20 }
```

c. The class **WeatherItem** (Fig.13.7,p.805,806):

(a) The class **WeatherItem** *extends* (i.e. it is a subclass of) **JPanel** class to display weather info in a **WeatherBean**. in a **JList**.

(b) This class is used by the **WeatherCellRenderer** (line 18 of that class) to display weather info in a **JList**.

(c) The **static** block at lines 22-29 loads the **ImageIcon** *backgroundImage* into memory when the Java virtual machine loads the **WeatherItem** class itself. This action ensures that **backgroundImage** will be available to all instances of class **WeatherItem**.

(d) The method **paintComponent** at lines 38-56 draws the **backgroundImage** (line 43), the city name (line 50), the temperature (line 51), and the **WeatherBean**'s ImageIcon (line 54).

All weather condition images are provided in a subdirectory in the class directory.

```
1  // WeatherItem.java
2  // WeatherItem displays a city's weather information in a JPanel.
3  package com.deitel.advjhtp1.rmi.weather;
4
5  // Java core packages
6  import java.awt.*;
7  import java.net.*;
8  import java.util.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class WeatherItem extends JPanel {
14
15    private WeatherBean weatherBean;  // weather information
16
17    // background ImageIcon
18    private static ImageIcon backgroundImage;
19
20    // static initializer block loads background image when class
21    // WeatherItem is loaded into memory
22    static {
23
24       // get URL for background image
25       URL url = WeatherItem.class.getResource( "images/back.jpg" );
```

```
26
27       // background image for each city's weather info
28       backgroundImage = new ImageIcon( url );
29    }
30
31    // initialize a WeatherItem
32    public WeatherItem( WeatherBean bean )
33    {
34       weatherBean = bean;
35    }
36
37    // display information for city's weather
38    public void paintComponent( Graphics g )
39    {
40       super.paintComponent( g );
41
42       // draw background
43       backgroundImage.paintIcon( this, g, 0, 0 );
44
45       // set font and drawing color,
46       // then display city name and temperature
47       Font font = new Font( "SansSerif", Font.BOLD, 12 );
48       g.setFont( font );
49       g.setColor( Color.white );
50       g.drawString( weatherBean.getCityName(), 10, 19 );
51       g.drawString( weatherBean.getTemperature(), 130, 19 );
52
53       // display weather image
54       weatherBean.getImage().paintIcon( this, g, 253, 1 );
55
56    } // end method paintComponent
57
58    // make WeatherItem's preferred size the width and height of
59    // the background image
60    public Dimension getPreferredSize()
61    {
62       return new Dimension( backgroundImage.getIconWidth(),
63          backgroundImage.getIconHeight() );
64    }
65 }
```

(7) Compile and run the program

a. Compile the program:

javac *.java

b. Compile the remote server class `WeatherServiceImpl` using the *rmic* compiler:

rmic -v1.2 WeatherServiceImpl

The option '-v1.2' instructs rmic the classes in this example will only be used by Java 2 applications. Normally *rmic* will generate both *skeleton* and *stub* classes. With '-1.2' *rmic* will only generate *stub* classes.

c. Start the *rmiregistry* process:

rmiregistry

d. Run the server:

java WeatherServiceImpl

e. Run the client:

java WeatherServiceClient

If the server and client are in the same host, run the client with the server's IP number

java WeatherServiceClient server's-IP-number

5. Java IDL

(b) Implementing the Server (XyzServer.java)

(c) Implementing the Client (XyzClient.java)

(d) If necessary, implementing the Applet Client (XyzApplet.java)

III. OSF DCE RPC

1. Overview

   (0) DCE application programming interface (p.15 & 16 of the the book *Understanding DCE*)

   (1) The client-server model (Fig.1-1)

   (2) The RPC mechanism (Fig.1-2)

   (3) Three phases required to develop a distributed application (Fig.1-3):

      a. Interface development

      b. Server development

      c. Client development

   (4) The arithmetic example

      a. An one-client/one-server RPC application. The remote server system has special hardware (e.g. processor array) for performing arithmetic operations;

      b. Outline: Clients make RPC call to the server to perform arithmetic operations; Server computes and returns results; Results are displayed on client's screen.

      c. A simple interface for the arithmetic example (Fig.1-4).

         (a) Interface definition language (IDL). E.g.: the rpcgen language and XDR files.

         (b) IDL resembles C in syntax and semantics

      d. Universal unique identifier (UUID)

         (a) UUID: a number, uniquely identifies an interface.

         (b) *uuidgen*: a utility that generates UUIDs.

```
% uuidgen -i
```

      e. The interface definition: p.7, Example 1-1.

         (a) *uuid* attribute specifies the interface UUID

         (b) The *interface* keyword, followed by the name of the interface *arithmetic*.

         (c) Section for type and constant definitions

         (d) Section for declaring procedures: The *in* and *out* parameters.

      f. Generating stub and header files using IDL compiler

         (a) A header file that contains the definitions needed by the stubs and the application code. Can be included anywhere appropriate.

(b) A client stub file, to be linked with the client portion of the application code.

(c) A server stub file, to be linked with the server portion of the application code.

(d) Invoking the IDL compiler:

```
% idl arithmetic.idl
```

g. A simple client

(a) Produce a client (Fig.1-5);

(b) A simple client code: Example 1-2, p.9

i. Header file;

ii. RPC call.

h. A minimal server: two parts: the actual remote procedure – sometimes called the *manager*; And the code that initializes the server.

(a) Files and utilities needed to produce a server (Fig.1-7, p.13, not Fig.1-6).

(b) Remote procedure implementation (Example 1-3, p.11-12).

i. header file "arithmetic.h" included

ii. The procedure definition matches its corresponding def. in interface def.

iii. The body of the remote procedure

i. A distributed application environment (Fig.1-7,p.13).

(a) *Binding*: relate the RPC call on a local machine to the remote procedure implemented on a remote machine.

(b) *Binding information*, which includes:

i. Protocol sequence: an RPC-specific name containing a combination of comm. protocols that describe the network comm. used between a client and a server. Example: *ncacn_ip_tcp* stands for a Network Computing Architecture connection-oriented protocol, over a network with the Internet Protocol and the TCP.

ii. Server host: the name or network address of the host on which the server resides

iii. Endpoint: a number representing and identifying the specific server process on the server host – typically a port number.

(c) *Name service* on DCE: a database service that provide binding information.

i. A server can store binding info that a client on another system can retrieve later;

ii. Called CDS (Cell Directory Service)

iii. *Name service independent*(NSI) routines: provides as RPC runtime library routines. They communicate with CDS to put info into the database

when the server called them.
NSI routines are a level of abstraction above the particular name service on a system.

iv. Name service mechanism, although not required, is preferred in distributed application environments.

j. Server initializing: Fig.1-9, p.15

(a) A server first registers the interface with the RPC runtime library (Step 1), which creates binding information to identify this service (Step 2).

(b) The server places binding info in appropriate databases, and places comm. and host info in the name service database (Step 3)

(c) The server places process information (called *endpoint*, i.e. port number) in a special database called *local endpoint map* on the server system (not CDS!).

(d) The serve waits and listens for incoming RPC requests (Step 5).

k. Client finding a server: Fig.1-10, p.16

(a) An RPC is made (Step 6). The client stub looks for the server system info from the CDS (Step 7).

(b) The runtime library finds the server process endpoint by contacting the local endpoint map on the server system (Step 8).

(c) Now the binding info is complete and the client completes the binding

l. Complete a RPC: Fig.1-11, p.17

(a) The client stub puts parameters and other calling info into an internal RPC format, which is transmitted over the network (Step 10 & 11).

(b) The server runtime library receives the data and transfers it to the server stub (Step 12); The server stub converts the data back to a format appropriate for use (Step 13).

(c) Remote procedure is dispatched (Step 14).

(d) Now the reverse operations are performed: Step 15 (prepare output); Step 16 (transmit output); Step 17 (receive output); Step 18 (convert the output).

m. Server initialization code of the *arithmetic* example

(a) Register the interface.

· The routine *rpc_server_register_if* is used to register the interface. The variable *arithmetic_v0_0_s_ifspec* is called an *interface handle*, which is produced by the IDL compiler and refers to the info such as UUID that the application needs. The *CHECK_STATUS* macro is defined in the *check_status*.h header file for examples in this book.

(b) Create binding info.

- The routine *rpc_server_use_all_protseqs* requests that the clients can use all available protocols. During this call, the RPC runtime library gathers info about available protocols, your host, and endpoints to create binding info. The system allocates one buffer for each endpoint to hold incoming call info. The size of the buffers is decided by the *rpc_c_protseq_max_calls_default* argument.

(c) Obtain the binding info.

- When creating binding info, the RPC runtime library stores binding info for each protocol seq. A *binding handle* is a reference in application code to the info for one possible binding. *binding vector*: a set of binding handles. The binding vector can only be obtained through the *rpc_server_inq_bindings* routine to pass the info to other DCE services with other runtime routines.

(d) Advertise the server location.

i. The routine *rpc_ns_binding_export* places all the binding info to the name service database.

ii. Entry name: is a string obtained from an environment variable (*ARITHMETIC_SERVER_ENTRY* for this example) set by the user specifically for the application.

iii. The interface handle *arithmetic_v0_0_s_ifspec* associates interface info with the entry name in the name service database.

iv. The client later uses name service routines to obtain binding info by comparing the interface info in the name service database with info about its own interface.

v. The argument *rpc_c_ns_syntax_default* tells the routine how to interpret an entry name. At present DCE has only one syntax.

(e) Register the endpoints in the local endpoint map.

i. The RPC runtime library assigns endpoints to the server as part of creating binding info.

ii. The runtime routine *rpc_ep_register* lets the endpoint map on the local host know that the process running at these endpoints is associated with this interface.

(f) Free the set of binding handle.

i. The runtime routine *rpc_server_inq_bindings* allocates memory for binding handles.

ii. The runtime routine *rpc_binding_vector_free* frees this memory when the server finishes passing binding info to other parts of DCE.

(g) Listen for remote calls.

i. The runtime routine *rpc_server_listen* let the server waits for calls to arrive. The 1st argument *rpc_c_listen_max_calls_default* gives the maximum number of calls that a server can accept at one time.

ii. Note: the concept here is the same as the `listen` function in networking programming.

n. Producing the application: compile and link the server and client separately.

(a) DCE libraries: -ldce and -lcma libraries. The libcma.a (Concert Multithreaded Architecture library) is required for DCE to handle threads internally.

(b) Compile and link the client code

(c) Compile and link the server code

o. Running the application

(a) The binding method used in *arithmetic* example: *automatic binding*, which requires that the client has to set the RPC-specific environment variable *RPC_DEFAULT_ENTRY* so that, so that the client has an entry name with which to begin looking for binding info.

(b) Other binding methods (implicit method and explicit method) exist and will be covered later (Ch.3).

(c) The example uses the simplistic approach by assigning *RPC_DEFAULT_ENTRY* the same entry name used in server initialization when exporting the binding info to the name service database.

(d) To run the distributed arithmetic application:

i. The server process must have read and write access to the name service database to export binding info to it.

ii. First set the environment variable *RPC_DEFAULT_ENRTRY* before the executing the server. This variable represents a name for the entry that this server uses when exporting the binding info to the name service database

iii. The usual convention for entry names is to concatenate the interface and host names.

iv. An environment variable is needed here because the name can vary depending on which host the server is to run.

v. If such a name is not valid, the binding info will not be placed in the name service database and the program will fail.

vi. The special prefix /.:// is required to represent the global portion of a name in the hierarchy of name service database.

moxie> setenv ARITHMETIC_SERVER_ENTRY /.://arithmetic_moxie

vii. Execute the server.

moxie> server

(e) On the client host:

moxie> setenv RPC_DEFAULT_ENTRY /.://arithmetic_moxie

and run the client:

moxie> client

(f) Using ctl-C to kill the server.

p. Complete development process of the arithmetic example: Fig.1-12, p24.

2. Using DCE RPC interface

(0) DCE RPC Interfaces

a. Interface definition is written in IDL and interface definition files end with suffix .idl

b. Compilation of an interface produces a C header file, the client stub file, the server stub file.

c. An interface definition, which describes the procedures offered by a server, is usually written by the server developer. Client developer has to read and interpret the definition.

d. All servers that support the interface must implement the remote procedures using the same procedure names, data types, and parameters. All clients must call the RP consistently

e. A *procedure definition* in an interface definition specifies the proc. name, the data type of the value it returns, and the number, order, and data types of its parameters.

(1) The Interface Definition Language (IDL)

a. IDL has syntax similar to C

b. *Attributes*: IF attributes are special keywords that offer info that helps describe an application. IF attributes are enclosed in square brackets in the IDL file.

(a) Some attributes distinguish one interface from another on a network. Eg.: the uuid attribute declares the UUID for the interface. These attributes guarantees that the client will find the server that implements the proper RP.

(b) Some attributes explicitly describe data transmitted over a network. Some aspects, which are taken for granted in normal applications, must be described explicitly for distributed applications. Eg.: *union*, in distributed applications, a separated variable indicating the exact data type in an application must be specified in IDL so that it can be transmitted with a union parameter.

(c) Some attributes make data transmission more efficient: input, output, or both.

Table A-1 to A-8 in Appendix A show all IDL attributes.

c. Structure of an interface definition
   (a) The interface header
      i. Interface header attributes
      ii. Interface name
   (b) The interface body
      i. Import statements
      ii. Constant definitions
      iii. Data type definitions
      iv. Procedure declarations

d. Interface header attributes: they specify RPC features that apply to the entire interface.
   (a) uuid attribute; major version number; minor version number.
   (b) Rules that determine whether a client can use an interface that a server supports:
      i. The UUID of client and server must match;
      ii. The major version number of the client and server must match;
      iii. The minor version number of the client must be less than or equal to the minor version number of the server. Upward compatibility.

e. Another example: the *inventory* application
   (a) A product database is store on the server system. A client makes inquiries based on a part number. (complete code in Appendix D).
   (b) The inventory interface (Example 2-1, p.28).
      i. Notice the square brackets;
      ii. The *uuid* attribute contains the UUID of the IF;
      iii. The *version* attribute is optional;
      iv. The *pointer_default* attribute is optional and used to more efficiently transmit pointer data;
      v. The keyword *interface* and a name are required to identify an IF. The IDL compiler uses this name to construct data structure names.

f. Type definitions, data attributes, and constants
   (a) Concepts of IDL data types.
      i. In C, a data type can map to different sizes on different systems. Eg.: long int

ii. IDL Rules: to ensure that the size of an IDL data type is same on all systems (so that DCE applications can exchange data), the IDL compiler generates C code data types that begin with *idl_*, and places them in the header file.

iii. Although the size of a particular IDL data type is always the same, it may map to different C data types different systems. The *idl_* data types are defined in a DCE RPC-supplied header file to map to the proper sized C data type for the system the compile took place.

(b) Basic IDL data types (Table 2-1, p.29,30).

i. Notes on basic IDL data types: Table 2-2, p.30-31.

ii. How do the IDL data types help to distribute and application? – the stubs perform the *marshalling and unmarshalling* operations. Mashalling converts data into a byte stream format and packages it for transmission using an NDR (Network Data Representation). NDR allows successful sharing between systems with different data formats. It handles differences like big- versus little- endian (byte order), ASCII chars versus EBCDID chars, and other incompatibilities.

(c) An example IDL type definition for the inventory example: Example 2-2, p.32

i. The keyword *const* defines a constant;

ii. The keyword *typedef* defines a new data type;

iii. A data type is defined by other data types, constants, and possibly some attributes.

(d) Pointers: pointers in distributed applications cannot provide the efficiency as in normal applications, as they involve extra copying and transmission. Two types of pointers are supported in IDL to balance convenience and efficiency:

i. *Full pointers*: has all the capabilities that usually associate with pointers. Cause additional stub overhead. Notated by the keyword *ptr*.

ii. *Reference pointers*: a simpler pointer that refers to existing data. Less costly. Limited in capabilities. No new memory can be allocated for the client during RPC. Notated by the keyword *ref*.

(e) Arrays: subscript starts at 0, same as in C. To be more efficient in handling arrays, IDL provides three types of arrays:

i. *Fixed array*: like a standard C array

ii. *Varying array*: has a maximum size determined at compile time. But it also has subset bounds represented by variables. Only the portion of the array you need is transmitted.

iii. *Conformant array*: The size of a conformant array is represented by a dimension variable and the actual size is determined when the array is used.

(f) Strings: strings in IDL does not have to end with the null char. To require a string to end with a null char, the *string* attribute must be used. Example 2-3, p.34:

i. The data type of a string array must be char or byte.

ii. The example defines a *conformant string*.

iii. The array attributes *size_is* and *max_is*

(g) Enumerated types: like in C. 1st element is mapped to zero. Example 2-4, p.35.

(h) Structures: defined with the same syntax as in C. Example 2-5, p.35.

(i) Discriminated unions

i. Conventional discriminated union: a discriminator variable that is separate from the union structure is used to keep track of the data type stored in the union.

ii. In IDL, a discriminated union includes a discriminator as part of the data structure itself.

Example 2-6, p.36, Defining discriminated union in IDL

iii. Keywords *typedef union* introduce the type definition;

iv. The keyword *switch* specifies the the data type and name of the discriminator variable – can be boolean, char, or int.

v. Name of the union data type (*total*.

vi. Keyword *case*. The case value is of the same type as the discriminator variable.

vii. The name of the new data type (*part_quantity*).

The IDL compiler converts each of such definitions into a structure with two components: the 1st component is the discriminator variable and the 2nd component is a C union. Example 2-7: A discriminated union generated by the IDL compiler.

Example 2-8, p.37: using discriminated union in application code:

viii. The *part_quantity* dis. union is a member of the *part_record* structure (Example 2-5).

ix. The *part.quantity* struct member is a dis. union.

x. The *part.quantity.units* struct member is the discriminator for the union.

xi. The *part.quantity.total* struct member is the union.

The union name can be omitted and IDL compiler will generated *tagged_union* for the user.

(j) Pipes: pipes can make data transmissions much more efficient than regular RPCs in the following case:

i. Large amount of data must be transmitted at once;

ii. The total amount of data is unknown until the application is running such as when processing files;

iii. The data is incrementally produced and consumed (Ch.11 has details).

g. Procedure declarations and parameter attributes: Example 2-9, p.39. Seven procedures declared.

* An IDL procedure can return a value, just like in C.

(a) Procedures defined with *void* type do not return a value.
Input and output parameters;
As in C, arrays and strings are implicitly passed by reference.

(b) Some procedures may return a data structure or a pointer to a data structure.

(c) Output pointers requires pointers to pointers when new memory is allocated.

(d) Parameters that are changed by the RP use both *in* and *out*.

(e) *Idempotent* procedures: those that can execute more than once with the same arguments and to produce identical results without any undesirable side-effects.

(2) Using the IDL Compiler

a. Utilities used and files produced during interface production: Fig.2-1, p.41.

(a) An ACF (Attribute Configuration File) file is optional. Such an file contains info that changes how IDL compiler interprets the IF definition.

(b) The files produced depend on the compiler options used.

i. Client stub file names: contain *_cstub* suffix;

ii. Server stub file names: contain *_sstub* suffix;

iii. Header file: end with .h;

iv. Object stub files: .o for both client and server. These object files are generated by C compiler called by IDL compiler.

v. Auxiliary files: *_caux* for client and *_saux* suffix for server. AUX files are generated when certain features are used. They contain special routines required for certain data types to prepare for data transmission. Features requiring AUX files include self-referencing pointers and pipes. Those routines are placed in AUX files rather than in the stub files so that the data types can be used in other IF definitions without linking the entire stub.

b. Generating client files: p.42

(a) -v option: verbose mode;

(b) -server none option: suppresses the generation of stub and auxiliary files for the server;

(c) -Iexplicit option: search the dir *explicit* for additional file for inclusion;

(d) -out explicit option: places the output file in the chosen directory *explicit*.

c. Generating server files

(a) -v option: verbose mode;

(b) -client none option: suppresses the generation of stub and auxiliary files for the client.

(3) Using an ACF to Customize Interface Usage

a. ACF files allows a user to control some aspects of RPC on the client side without affecting the server – These aspects are not in the IF definition because we do not want to force them on all clients and servers.

(a) A client or server developer can use an optional ACF file to modify the IDL compiler creates stubs, without changing the way the stubs interact across the network.

(b) Most significant effect: add additional parameters to remote procedures not declared in the IF definition: binding method selection (as 1st parameter of RP; status parameters to the end of parameter list.

(c) One does not specify ACF files when compiling an IF file. IDF compiler will automatically uses an ACF file if one is available in the search dir.

b. Selecting a binding method:

(a) The *auto_handle* attribute selects the automatic binding method. Additional advantage of this method is error recovery: a client stub sometimes can find another server transparently if the original server is unreachable.

(b) The *implicit_handle* attribute selects the implicit binding method. This allows a user to select a specific server for all the RPC calls

(c) The *explicit_handle* attribute selects the explicit binding method. This allows a user to select a specific server for each RPC.

(d) An example ACF file: Example 2-10, p.44.

i. The *implicit_handle* attribute applies to the entire IF. A global binding handle of type *handle_t* is established in the client stub to refer to binding info the client uses to find a server;

ii. The IF name *inventory* must match the IF name in the corresponding IDF file.

c. Controlling Errors and Exceptions

(a) DCE causes comm. and server errors to be raised as exceptions;

(b) The normal action in the face of an exception is that the server will exit;

(c) An ACF can allow you to provide extra code for exception handling: the attributes *comm_status* and *fault_status* can be applied to procedure parameters or return values. The presence of these attributes will allow comm. or server errors to be comm. to the client as values in the named parameters, rather than raised as an exception (Ch.3 has more coverage).

d. Excluding Unused Procedures

(a) The *code* and *nocode* ACF attributes allow users to define which procedures a client stub supports.

(b) However, all the procedures declared in the IF must be implemented by the server.

e. Controlling marshalling and unmarshalling

(a) The *out_of_line* ACF attribute causes constructed data types such as unions, pipes, and large structures to be marshalled or unmarshalled by auxiliary routines, thus reducing the stub size: the marshalling and unmarshalling code now is put into auxiliary files, rather than in the stub directly.

(b) The *in_line* ACF attribute causes data types to be marshalled or unmarshalled as fast as possible. The marshalling and unmarshalling code is put into the stub directly. This is the default action.

(c) These two attributes only affect the stub code.

3. Writing clients

(1) Binding

a. Implementing a binding method

(a) Automatic method: the client stub automatically manages binding handles after the application calls a RP. The client stub obtains binding info from a name service database and passes the binding handle to the RPC runtime library. If the connection is disrupted, new binding info can sometimes be automatically obtained and the call is tried again.

(b) Implicit method: a binding handle is held in a global area of the client stub. After the application calls a RP, the stub passes the binding handle to the RPC runtime library. One writes application code to obtain the binding info and set the global binding handle with RPC runtime routine calls.

(c) Explicit method: an individual RPC in the application passes a binding handle explicitly as its 1st parameter. One writes application code to obtain the binding info and set the binding handle with RPC runtime routine calls.

(d) Comparisons of binding methods: Fig.3-1,p.49

i. Top half: the client application code; bottom half: the client stub code generated by IDL compiler; Shadings represents the portion of the client where binding handles are managed.

ii. Different binding methods can be employed for different RPCs in a given client instance.

iii. Automatic binding is convenient for learning and test purposes. But is not useful for most production applications because one cannot use security.

iv. The auto and implicit methods apply to an entire interface. However, in these cases, one can still use the explicit method for some or all RPCs to that interface – the explicit method takes precedence over the other two methods because the binding handle is present as the 1st parameter in the procedure.

v. A client cannot use both auto and implicit methods simultaneously for RPCs to a single interface.

vi. The imp and exp methods requires that the application code obtain binding info and manage the binding handles. Binding handles need to be obtained and managed in the client app code under the following three circumstances:
i. The client uses a specific server;
ii. The client needs to set authentication and authorization info for specific binding handles;
iii. The server has more than one implementation of the same remote procedure (UUIDs are used to differentiate between difference RP implementations).

(e) One can use an ACF file to establish a binding method with attributes *auto_handle*, *implicit_handle*, or *explicit_handle*.

(f) *Context handle*: is a special RP parameter defined in an IF definition with the *context_handle* attribute. Applications use a context handle in a seq of RPCs to refer to a context (state) on a specific server. Context handles carry with them binding info and thus can act as binding handles in RPCs – when a context handle is active, it carries with it the binding info necessary to find the same server as it did before.

b. Deciding on binding methods

(a) When an IF definitions compiled, the auto binding method is the default unless:

i. The 1st parameter of a procedure declaration is a binding handle;

ii. The proc declaration has an input context handle;

· An ACF establish a different binding method.

(b) If the 1st parameter is a binding handle, that procedure must use the explicit method.

(c) For those procedures whose 1st parameters are not binding handles, one must decides whether to use the auto or implicit binding methods to all of them.

(d) The auto binding method only works well within a small network. However you have no control over which server you get, so applications that use servers scattered over a wide area may be inefficient.

(e) If most of the RPCs need to use a specific server chosen in one's application code, the implicit method is appropriate.

(f) The explicit binding method allows one to have more control over which server to use.

(g) The explicit method is necessary for clients that make multithreaded RPCs. Eg. a commodity trade application may request a commodity price with RPCs to many locations at the same time. Load is branched over the whole system in this case.

c. Automatic binding management

(a) Simplest.

i. One does not deal with the binding handle in the IF definition, ACF, or application code.

ii. The binding handle is hidden in the client stub. One only has to set up the environment variable for entry name and the rest is done entirely by the client stub and the runtime library.

iii. Rebinding is possible if a server connection is lost.

iv. Different servers implementing the same RP may be chosen at different call instances.

(b) Interface development for automatic binding: there are no special requirements in the interface for auto binding. If *auto_binding* attribute in an ACF can be used for documentation purpose.

(c) Client development for automatic binding: the client requires the user to:

i. Include the header file in the client application code;

ii. Link the client application object code with the client stub, client stub auxiliary file (if present), and DCE libraries.

iii. Set the environment variable *RPC_DEFAULT_ENTRY* to a valid name service entry so that the client stub can automatically begin a name service database search.

The RPC is just like a LPC.

(d) Server development for automatic binding

i. A server must advertise binding info to a name service entry with the *rpc_ns_binding_export* runtime routine in the server initialization code.

ii. Alternatively, an administrator can insert the info from the shell using DCE RPC control program (*rpccp*).

d. Implicit binding management

(a) It gives a user the control of binding management in the client application code without a visible binding handle parameter in the RPC.

i. Use this method for applications that need the same server for all or most RPCs of an IF.

ii. An ACF defines the binding handle and the IDL compiler generates it as a client-global variable in the client stub.

iii. The client application code sets the binding handle before any RPCs.

iv. During a RPC, the client stub uses the global binding handle to complete the call to the RPC runtime library.

(b) Interface development for implicit binding:

i. Use the *implicit_handle* attribute in an ACF to declare a global binding handle for the client: Example 3-1,p.54.

ii. When the IF definition is compiled with the ACF present, a global binding handle is defined in the client stub, which uses this handle every time the client calls a RP for this interface.

(c) Client development for implicit binding

i. Include the header file.

ii. The client must obtain the binding info and assign its handle to the global binding handle. The binding info can be from the name service database, or composed from strings of binding info. The *do_import_binding* routine is developed in later of this ch.

(d) Server development for implicit binding

i. No special requirements in server development

ii. A server must export to a name service database if the clients use a name service to find servers. Server binding info can also be exported by *rpccp* control program.

e. Explicit binding management

(a) This binding method manages each RPC separately. The 1st parameter of the RP is a binding handle.

i. Use this method if the clients need to make RPCs to more than one server. The application code has total binding info control.

ii. If a procedure declared in an IF has a binding handle as its 1st parameter, one must use the explicit method for this RP.

iii. If a procedure does not have a binding handle as its 1st parameter, one still can use an ACF add one.

(b) Interface development for explicit binding

i. An IF definition or an ACF uses the *handle_t* data type to define binding handle parameter;

ii. Application code uses the *rpc_binding_handle_t* data type to represent and manipulate binding info. These two data types are in fact equivalent. The former exists for compatibility with previous RPC versions.

iii. An example that uses an ACF file that adds a binding handle parameter: Example 3-3, p.56.
When the IDL compiler uses that ACF file, all procedure declarations in the header file have a binding handle of type *handle_t*, named *explicit_handle* added as the first parameter.
The header file generated by the IDL compiler contains a code (shown on p.56).
The *explicit_handle* attribute can also be added to a specific procedure in the ACF file to add a binding handle as the first parameter. Example: p.57

iv. Adding a binding handle in the IF definitions: Example 3-4, p.57.
This approach is more restrictive than a binding handle defined with a ACF file.
However, RPs can be used the same way either case

(c) Client development for explicit binding

i. Before making a RPC, the client must obtain binding info and set the binding handle. The methods of obtaining binding handles are almost the same as for implicit method.
For the explicit method, one uses a specific binding handle instead of assigning the binding info to the implicit global binding handle.

ii. Example 3-5, p.57.
Header file included
Binding handle variable declared
Obtaining binding handle
Notice the 1st parameter is a binding handle in the RPC

(d) Server development for explicit binding

i. If the IF definition does not have a binding handle parameter for the RP, and one wants the RP to obtain client binding handle (such as for authentication and authorization), one must use an ACF with *explicit_handle* attribute to create the binding handle.

ii. If clients use a name service to find servers, the server must export to a

name service database. Can also be done through control program *rpccp*.

(2) Steps in Finding Servers

a. Finding a protocol sequence: two possible methods:

(a) The preferred method: using a name service database to import or look up both host address and protocol sequence at the same time. Use RPC runtime routines that begin with *rpc_ns_binding_import_* or *rpc_ns_binding_lookup_*.

(b) Use a protocol sequence string obtained from your application or from a call to the *rpc_network_inq_protseq* routine. The two runtime routines *rpc_string_binding_compose* and *rpc_binding_from_string_binding* to prepare and set the binding handle.

b. Finding a server host: two different ways:

(a) Use a name service database to import or look up a host address and at the same time get a protocol sequence.

i. The RPC runtime routines that begin with *rpc_ns_binding_import_* or *rpc_ns_binding_lookup_* to set the binding handle.

ii. For automatic binding method, the client stub does this for you.

(b) Use a host name or host network address string obtained from your application. Use the RPC runtime routines *rpc_string_binding_compose* or *rpc_binding_from_string_binding* to the binding handle.

Partially bound binding handle: is one that contains a protocol sequence and server host, but not an endpoint.

(c) That is what one gets from CDS.

(d) When a partial bound binding handle is passed to the RPC runtime library, and endpoint is automatically obtained for you from the interface or the endpoint map of the server's system.

c. Finding an endpoint

(a) Several terms:

i. *Fully bound binding handle*: is one that contains info of a partial bound binding handle together with the endpoint info.

ii. *Well-known endpoint*: is a preassigned system address that a server process uses every time it runs. Usually a well-known endpoint is assigned by the authority responsible for a transport protocol.

iii. *Dynamic endpoint*: is a system address of a server process that is requested and assigned by the RPC runtime library when a server is initiated. Most applications should use dynamic endpoints to avoid the network management needed for well-known endpoints.

(b) Endpoints can be obtained through one's application code. But it is best left for the runtime library to obtain it. Four ways to obtain an endpoint:

i. If the binding info obtained during an import or lookup of the protocol seq and host in the name service database includes an endpoint, the binding handle is fully bound in one step. However, a name service database only stores well-known endpoints, not dynamic endponts.

ii. A well-known endpoint is found that was established in the interface definition in the *endpoint* attribute. The RPC runtime library or the application finds the endpoint from an interface specific structure.

iii. An endpoint is found from the endpoint map on the server system. They can be well-known or dynamic. The runtime library first looks for an endpoint from the interface spec. If it cannot find it there, one has to use the routine *rpc_ep_resolve_binding* or routines beginning with *rpc_mgmt_ep_elt_inq_* in the application to find the endpoint.

iv. One can use a string from the application that represents an endpoint, and the use the runtime routine *rpc_string_binding_compose* and *rpc_binding__from_string_binding* to set the binding handle. These endpoints can be well-known or dynamic.

d. Interpreting binding info: what goes on in the *do_import_binding* procedure shown earlier?

(a) When using implicit or explicit binding, one needs to interpret the binding info. Example: one needs to use a server on a particular host, that means that one has to extract the host from the binding handles one gets from the CDS.

(b) One may interpret the binding info of a binding handle to control which server a RPC will use or which binding handles a server will offer. binding handles offer the following binding info:

i. Object UUID

ii. Protocol seq

iii. Network address or host name

iv. Endpoint

v. Network options (are specific to a protocol seq and not relevant to the connection-oriented or datagram protocol sequences).

(c) Example 3-6, p.63: how to use RPC runtime routines to interpret binding info. These routines can be used in either clients or servers. The *do_interpret_binding* procedure is called in the *do_import_binding* procedure (next example).

i. The example interprets binding info and extract and return protocol seq

ii. The *rpc_binding_to_string_handle* routine converts binding info to its char string representation.

iii. The *rpc_string_binding_parse* routine obtains binding info items as separate allocated strings. The components include all five pieces of info in a binding handle.

iv. The *rpc_string_free* free strings allocated by other RPC runtime routines.

e. Finding a server from a name service database

(a) The usual way for a client to find binding info is from a name service database using the name service RPC runtime routines (routines beginning with *rpc_ns*). The name service database contains entries of info, some of which contain binding info about specific servers; some of them about a set of servers; some of them a search list.

(b) Importing a binding handle: Example 3-7, p.65, the *do_import_binding* routine

i. The *rpc_ns_binding_import_begin* routine establishes the beginning of a a search for binding info in a name service database. An entry name syntax of *rpc_c_ns_syntax_default* uses the syntax in the RPC-specific environment variable *RPC_DEFAULT_SYNTAX*.

ii. The starting search entry is */.:/inventory_group*, which is passed as a parameter. If a null string is passed as entry name, the search begins with the name in the environment variable *RPC_DEFAULT_ENTRY*.

iii. The IF handle *inventory_v1_0_c_ifspec* refers to the IF specification. It is generated by the IDL compiler and defined in the header file *inventory.h*.

iv. The *rpc_ns_binding_import_next* routine obtains binding info that supports the IF, if any exists. The routine only accesses the database and does not communicate with the server. The import handle, established with the call *rpc_ns_binding_import_begin*, controls the search for compatible binding handles.

v. Once binding info is obtained, the application code can perform any action it intends. In this example, the routine *do_interpret_binding*, which returns the protocol seq of a given binding handle, is called.

vi. Each call to *rpc_ns_binding_import_next* requires a corresponding call to the *rpc_binding_free* routine, which frees memory containing the binding info and sets the binding handle to null.

vii. The *rpc_ns_binding_import_done* routine signifies that a client has finished looking for a compatible server in the name service database. This routine frees the memory of the import context created by a call to *rpc_ns_binding_import_begin*. These two calls must be used in a pair.

(c) Looking up a set of binding handles

i. Runtime routines that begin with *rpc_ns_binding_lookup_* obtain a set of binding handles from the name service database.

ii. These set of binding handles can then be selected with the routine *rpc_ns_binding_select* or one's own selection criteria.

iii. Lookup routines give users a little more control than import routines because *rpc_ns_binding_import_next* returns a random binding handle from a list of compatible binding handles.

f. Finding a server from strings of binding data

(a) Without using the name service database, one has to to construct one's own binding info and binding handles. Binding info may be represented with strings. Binding handles may be composed from appropriate strings of binding info.

(b) Minimum info for an application to obtain a binding handle is:

· A protocol seq of comm. protocols

· A server network address or host name

Endpoints can be obtained by RPC runtime library. One can set a binding handle by calling RPC runtime routines with binding info.

(c) Example 3-8, p.67-69. The procedure *do_string_binding* sets binding handle from strings of binding info. This procedure us used by the *remote_file* application.

· The network address or host name of the the server system is required. Here it is passed as a char string parameter from the caller.

· The routine *rpc_network_inq_protseqs* creates a list of valid protocol sequences. Each of these protocol sequences is tried until a valid binding handle is created.

· The routine *rpc_string_binding_compose* creates a string of binding info in the argument *string_binding* from all the necessary binding info components.

· The *rpc_binding_from_string_binding* routine obtains a binding handle from the string of binding info, which can come from either the routine *rpc_string_binding_com* or *rpc_binding_to_string_binding*.

· When binding handle is finished its usage, the routine *rpc_binding_free* is called to set the binding handle to null, hence free its associated memory.

· The *rpc_string_free* routine frees strings allocated by other RPC runtime routines.

· The *rpc_protseq_vector_free* routine is called to free the list of protocol sequences. This call has to be used with the call *rpc_network_inq_protseqs* in pair.

(3) Customizing a Binding Handle

a. Several terms:

   (a) *Primitive binding handles*(PBH): those binding handles that we used so far: with five components;

   (b) *Customized binding handles*(CBH): handles obtained by adding to the PBH extra info that one's application wants to pass between client and server.

   CBHs can only be used with implicit or explicit binding method.

b. Why CBHs?

   (a) A CBH can be used when application-specific data is appropriate to use for finding a server, and the data also is needed as a procedure parameter.

   (b) Example: in the *transfer_data* application, a structure contains a host name and a remote file name. The application creates the necessary binding info from the host name, and the filename is passed with the binding info so the server knows what data to use.

c. How a customized binding handle works: Fig.3-2,p.70

   (a) A CBH is defined with the *handle* attribute applied in a type def in an IF def.

   (b) A CBH can be used the same way as a PBH. But one has to write special *bind* and *unbind* procedures. These two procedures will be called from client stubs, not the application code. These two are used by the client stub to obtain the PBH and do application cleanup when finished with the binding handle.

   (c) A CBH must be the 1st parameter in a RPC (or the global handle for the implicit method) to act as the binding handle for the call. A CBH no present as the 1st parameter of a RPC acts as a standard parameter.

d. The *transfer_data* example: Example 3-9, p.71-72

   (a) The *handle* attribute in the IF def is used to associate the CBH with a data type;

   (b) The *file_spec* data type is a structure whose members are file specifications. This is application-specific info used by the bind procedure to obtain server binding info.

   (c) The CBH is the first parameter of a procedure declaration because explicit binding method is used.

e. Implement the *bind* and *unbind* procedures used in *transfer_data*: Example 3-10, p.72.

   (a) The bind and unbind procedure names are constructed from the data type name *file_spec*, with suffix *_bind* and *_unbind* respectively.

(b) The bind procedure takes an input parameter of the customized handle data type, and returns a primitive binding handle. It calls the application-specific procedure *do_string_binding* (defined in Example 3-8) to obtain a primitive binding handle.

(c) The unbind procedure *file_spec_unbind* takes input parameters of the customized handle data type and a primitive binding handle. It calls the RPC runtime routine *rpc_binding_free* to free the binding handle.

f. A client with a CBH: Example 3-11, p.73.

(a) The application allocates a CBH in the def.

(b) The CBH variable is initialized before the RPC is made.

(c) The RPC is called with the CBH as the 1st parameter.

(4) Error Parameters or Exceptions

a. DCE RPC clients and servers require special error handling techniques to deal with errors that may occur during a RPC.

(a) Server and comm. errors are raised to the client as exceptions during a RPC. RPC exceptions are equivalent to the RPC error status codes.

(b) Types of errors include:

i. Exceptions raised on the client system (Eg. client system is out of memory);

ii. Exceptions raised to the client application by the client stub due to comm. errors (Eg. crashes on the remote system);

iii. Exceptions raised by the client stub on behalf of the server. These errors can occur in server stub, in the RP, in pipe support routines, or in the server's RPC runtime routines.

b. Using exception handlers in clients or servers: One can handle exceptions by writing exception handler code in the application to recover from an error or gracefully exit the application. DCE threads supply macros as framework to handle exceptions in one's client or server code.

c. Using remote procedure parameters to handle errors

(a) One can add error parameters to remote procedures in the ACF to conveniently handle comm. and server errors. The RPC runtime library then stores error values in these parameters rather than raising exceptions. One can use a combination of exception handlers and error parameters.

(b) For the *is_part_available* procedure: p.75, a statement in an ACF file shown on p.75 establishes an error parameter for that procedure.

i. The ACF attributes *comm_status* and *fault_status* establish a parameter in which to report comm. and server errors.

ii. These attributes can be associated with one or more parameters.

(c) In a client, an RPC to this procedure now has an additional parameter – the last parameter.

(d) The error code returned can be interpreted by the RPC runtime routine *dce_error_inq_text*. The application-specific macro *CHECK_ERROR* (Example 3-12, p.76) uses this routine.

(5) Compiling and Linking Clients

a. Files and libraries required to produce an executable client: Fig.3-3, p.78.

(a) Client auxiliary file is generated only when certain features are present:

i. Self-referencing pointers in IF def.

ii. The *out_of_line* attribute in an ACF.

(b) Compile the client modules with C compiler

(c) Link the client application, client stub, and client auxiliary object files (if any) with the DCE libraries to produce the executable client

b. Local testing: one can compile a local version of the client to test and debug RPs without using RPCs.

(a) Compile the client object files and remote procedure implementations without the stub or auxiliary files.

(b) A compilation directive -DLOCAL is used in this textbook to distinguish a test compilation.

(c) The object files produced in test compilation should be deleted to prevent it from interfering with the executable version.

4. Pointers and arrays

(1) kind of Pointers

a. Pointers: A pointer is a variable containing the address of another data structure or variable.

b. Pointer declaration in IDL: the same as in C.

c. Two types of pointers:

(a) Full pointers: a full pointer has all the capabilities usually associated with pointers. IF def. requires full pointers for the following cases:

i. When a RPC allocates new memory for the client. The client stub usually allocates the memory;

ii. When the value of the pointer is NULL, as with an optional parameter;

iii. When two pointers refer to the same address, as in a doubly-linked list.

Full pointers are specified using the *ptr* attribute in an IF def. Full pointer incurs significant stub overhead.

(b) Reference pointers: a reference pointer is used to refer to existing data. A reference pointer has performance advantage over a full pointer because stub overhead is reduced.

i. For reference pointers, the data pointed by such a pointer can change, but the value of the pointer itself cannot.

ii. Reference pointers are specified using the *ref* attribute in an IF def.

(c) Reference pointers

d. Pointers as output parameters

e. Pointers as input parameters

f. Using pointers to pointers for new output

g. Pointers as procedure return values

h. Pointers summary: Table 4-1, p.88

(2) Kind of Arrays

a. Three types of arrays

(a) *Fixed arrays*:

(b) *Varying arrays*:

(c) *Conformant arrays*:

b. Selecting a portion of an array

c. Managing the size of a conformant array

d. Conformant arrays as procedure parameters

e. Dynamic memory allocation for conformant arrays

f. Memory allocation for conformant structures

5. Writing servers

(1) Some Background on Call Processing

a. Fig.1-9 shows how a server is initialized before it accepts and processes RP calls.

b. Fig.1-10 shows how a client finds a server using automatic binding method.

c. Fig.1-11 shows the basic steps during a remote procedure call after the client finds the server.

d. How the RPC runtime library handles an incoming call: Fig.5-1, p.98:

(a) A call request for the server comes in over the network.

i. The request is placed in a request queue for the endpoint;

ii. The server can select more than one protocol seq on which to listen for calls, and each protocol seq can have more than one endpoint associated with it;

iii. Request queues temporarily store all requests, thus allowing multiple requests on an endpoint;

iv. Requests are rejected if the request queue is full.

(b) The RPC runtime library dequeues requests on at a time from all request queues and places them in a single call queue.

i. Using threads, the server can process RPCs concurrently. The number of threads can be determined at server initialization.

ii. If all threads are in use, the call remains in the call queue until one thread is available.

(c) After a call is assigned to a thread, the IF spec of the client call is compared with the IF spec of the server.

i. An IF spec is an opaque data structure containing info (including the UUID and the version number) that identifies the interface. *Opaque* here simply means that the details are hidden from you.

ii. If the server supports the client's IF, processing goes to the stub code. Otherwise, the call is rejected.

(d) When the call reaches the stub, it unmarshalls the input data. During unmarshalling, memory is allocated if necessary, data is copied from the RPC runtime library, and data conversion is performed if needed.

(2) Initializing the Server

a. Steps in the initialization code:

(a) Register the IF with the RPC runtime library

(b) Create server binding info by selecting one or more protocol seqs for the RPC runtime library to use in one's network environment.

(c) Advertise the server location so the clients have a way to find it. This usually involves storing binding info in a name service database.

(d) Manage endpoints in a local endpoint map.

(e) Listen for RPCs.

b. Example of server initialization of the inventory application: Example 5-1, p.100:

(a) The header file is always included.

(b) A *unsigned32* type variable is needed to report errors that may occur when an RPC runtime routine is called.

(c) Some RPC runtime routines used data structures called a *vector*, which in RPC applications contains a list (array) of other data structures and a count of elements in the list.

c. Registering IFs

(a) Before a client makes a RPC, it checks its IF against the one advertised in the server's binding info. The server will also compare its IF against the IF in the client's call.

(b) The application code uses an IF handle to refer to the IF spec. An IF handle is a pointer defined in the C header file generated by IDL compiler and contains the following four pieces of info:

i. The IF name;

ii. Version number (both major and minor);

iii. The letter *s* or *c*, depending upon whether the handle is for the server or client portion of the application;

iv. The word *ifspec*.

(c) Example 5-2: Registering an Interface with Runtime Library, p.101-102.

i. The *rpc_server_register_if* routine is required to register each IF that the server supports. The IF handle refers the the IF specification;

ii. The *CHECK_STATUS* macro is defined previously.

(d) Multiple IFs may be registered from a single server by calling *rpc_server_register_if* with a different IF handle.

(e) The 2nd and 3rd parameters of *rpc_server_register_if* are used in complex applications to register more than one implementation for the set of remote procedures. A NULL value for these two parameters means only one implementation.

(f) In the event of a symbol name conflict between the RP names of an IF and other symbols in the server, one can use these parameters to assign different names to the server code's remote procedures.

d. Creating server binding info

(a) Server binding info is created when one selects protocol seqs during server initialization. RPC uses protocol sequences to identify the combinations of comm. protocols that RPC supports.

i. Most servers offer all available protocol sequences to facilitate clients' connections.

ii. One may want use only one specific protocol sequence for debugging.

iii. The server address info is furnished by the system kernel itself.

An endpoint must be selected after protocol seqs are determined.

(b) Using dynamic endpoints: most servers use this type of endpoints.

i. Flexible;

· Selected by the RPC runtime routine and vary from one invocation to another.

ii. Example 5-3: a portion of the inventory server initialization:

• The *rpc_server_use_protseq* routine is called with the chosen prot. seq as the 1st parameter. This call selects one prot seq on which the server listens for RPCs. The constant *rpc_c_protseq_max_calls_default* sets the request queue size. At this time, the size is system dependent and cannot be controlled by users.

• The *rpc_server_use_all_protseqs* routine selects all available prot seqs.

• The *rpc_server_inq_bindings* routine is required to obtain the set of binding handles referring to all of this server's binding info.

(c) Using well-know endpoints

i. A well-know endpoint is selected and assigned to the same server every time it runs. They are more restrictive. Users must register them to certain Internet authority.

ii. Well-known endpoints are often used for widely-used applications such as core DCE servers like *rpcd*, which servers to endpoints.

iii. RPC routines for creating binding info with well-known endpoints: Table 5-1,p.105.

• *rpc_server_use_protseq_ep* takes as parameters a specified prot seq and a well-known endpoint to establish server binding info.

• *rpc_server_use_protseq_if* takes as parameter a specified prot seq, but the well-known endpoint is given in the IF def with *endpoint* attribute. Both clients and servers know the well-known endpoint from the IF def.

• *rpc_server_use_protseq_if* use all supported prot seqs, but the well-known endpoint is given in the IF def with *endpoint* attribute.

e. Advertising the server

(a) Advertising: making the binding info available for clients to find this server.

i. Three methods to advertise a server:
(i) Export to a name server database (most commonly used); (ii) Store binding info in an application-specific database; (iii) Print or display binding info for clients.
Which one to use is application dependent.

ii. An RPC entry name includes the IF name and remote host name. This means that the client has to know the specific server entry name before making an RPC.

iii. A name service group associates a general group name with a set of server entries, so that a client does not have to know specific host names.

iv. The convention for naming RPC group entries includes the the IF name. The server name is added as a member of the group.

v. More sophisticated naming approach is discussed in Ch.6.

(b) Export to a name server database: most commonly used. Example 5-4, p.106-107.

i. The RPC runtime routine *rpc_ns_binding_export* exports the server binding info to a name service database.

ii. The constant *rpc_c_ns_syntax_default* establishes the syntax the RPC runtime library uses to interpret an entry name.

iii. The IF handle *inventory_v1_0_s_ifspec* is needed so that IF info is associated with the binding info in the name service database.

iv. The *binding_vector* parameter contains the list of binding handles that the binding info exported.

v. The RPC runtime routine *rpc_ns_group_mbr_add* adds the server entry exported with the *rpc_ns_binding_export* call as a member of the name service group /.:/inventory_group.

(c) Three ways to manipulate the name service database:

i. Use the *rpc_ns_binding_export* routine together with other RPC runtime routines in the server initialization code (previous example);

ii. Use the RPC control program *rpccp* to export binding info;

iii. Use the *rpc_ns_mgmt_entry_create* routine together with other RPC runtime routines in a separate management application (not covered in this book).

(d) The routine *rpc_ns_binding_export* exports well-known endpoints together with other binding info to the name service database. Dynamic endpoints are not exported by this routine.

i. They are short-living in nature;

ii. May degrade the performance of the name service database if many obsolete dynamic endpoints exist;

iii. May cause client applications to fail if obsolete endpoints are used.

f. Managing server endpoints

(a) *Endpoint map*: a database on each RPC server system that associated endpoints with other server binding info. Normally the endpoint map contains all the endpoints (well-known or dynamic) on the server system.

(b) The endpoint map on a server system is managed by the *rpcd* daemon process. Applications use RPC runtime routines. The *rpccp* program can also interact with *rpcd*.

(c) With a partially bound binding handle, the client will do:

i. If there is a well-known endpoint given in the IF def, that endpoint will be used;

ii. Otherwise it will contact the server system *rpcd* daemon process to obtain an endpoint.

iii. The daemon process will compare the info in the endpoint map with the client's IF specification and binding info.

iv. When an valid endpoint is returned, the client has a fully bound binding handle to complete the connection.

(d) Example 5-5, p.109: Managing endpoints in an endpoint map: how to register its endpoints in the endpoint map.

i. The *rpc_ep_register* routine registers the server endpoints in the local endpoint map. Use the same IF handle, binding vector, and object UUID as using *rpc_ns_binding_export* routine. Annotation is encouraged.

ii. The *rpc_binding_vector_free* routine is used in pair with the *rpc_server_inq_bindings* routine (Example 5-3). This call should be made before put the server in listening mode.

(e) Some more notes:

i. The *rpc_ep_register* routine is required if dynamic endpoints are established with the *rpc_server_use_protseq* or *rpc_server_use_all_progseqs* runtime routines.

ii. If a well-known endpoint is established with the *rpc_server_use_protseq* routine, one also has to use the *rpc_ep_register* routine so that the client can find the value of the endpoint.

(f) Outdated endpoints are removed by *rpcd* daemon.

i. An unpredictable amount of time exists before an outdated endpoint is removed from the endpoint map.

ii. A server can use the routine *rpc_ep_unregister* to explicitly remove an endpoint before it terminates.

(g) Three ways to manage the endpoints in the endpoint map:

i. Use the *rpc_ep_register* and *rpc_ep_unregister* routines;

ii. Use the *rpccp* program;

iii. Use the *rpc_mgmt_ep_elt_inq_begin* routine and other RPC runtime routines in a separate management application.

g. Listening for RPCs

(a) The *rpc_server_listen* routine puts the server in listen mode and will not return unless the server is requested to stop listening by another process, by one of its own RPs using the *rpc_mgmt_stop_server_listening* routine.

(b) When an error occurs during stub or remote procedure code execution, an exception is raised. The server will immediately terminate unless the user provides code to handle it.

(c) The macros *TRY, FINALLY*, and *ENDTRY* delineate code sections in which exceptions are controlled:

i. If an exception occurs in the *TRY* section, code in the *FINALLY* section is executed to handle any necessary error recovery or cleanup;

ii. The *FINALLY* section contains clean-up code that does things as remove outdated endpoints;

iii. The *TRY* + 1 and *FINALLY* sections both end with the *ENDTRY* macro.

(d) Example: 5-6: listening for RPCs, p.111

i. The *TRY* macro begins a section of code the one expects exceptions to occur. In this example, it contains only one call.

ii. The *rpc_server_listen* routine is a required. The first parameter sets the number of threads that the RPC runtime library to process RPCs.

iii. The *rpc_server_inq_bindings* routine obtain a set of binding handles referring to all of the server's binding info.

iv. The *rpc_ep_unregister* routine removes the server endpoints from the local endpoint map.

v. The *rpc_binding_vector_free* routine is called in pair with the routine *rpc_server_inq_bindings*.

(3) Writing Remote Procedures

a. Issues in writing RPs:

(a) Memory management

i. Is especially important because the client and the RP are not in the same address space. Memory requested to be allocated by the client cannot be freed by the client.

ii. Memory management for RPs is done through the special *stub support routines* in RPs. Stub support routines enable the server stub to free memory allocated in RPs after the RPs finish their execution.

(b) Threads

i. The number of threads that a server uses to process RPCs is decided by the 1st parameter to the routine *rpc_server_listen*.

ii. *Thread safe*: multiple threads of the same RP will not cause unexpected or inconsistent results.

iii. The implementation of a RP should be thread safe if more than one thread is allowed for the RP. Locks may be used in the implementation.

(c) Client binding handle: *Client binding info* is used in server code to inquire and authenticate the client. This info includes:

i. The object UUID requested by the client; Can be simply nil (more to be discussed in Ch.7);

ii. The RPC protocol sequence used by the client for the call;

iii. The network address of the client;

iv. The optional client authentication and authorization information.

Client binding info is stored in the *Client binding handle*. If the client binding handle is available, it is the 1st parameter of the RP. If one requires CBI, and the procedure declarations in the IF def. do not have a binding handle as the 1st parameter, one must generate the server stub and header file using an ACF with the *explicit_handle* attribute.

b. Managing memory in remote procedures

(a) Typical C applications use *malloc* and *free* to allocate and deallocate memory.

(b) In RPC servers, if a remote procedure returns a pointer to newly allocated memory to the client, use stub support routines to manage memory in the RP.

(c) Use the RPC routine *rpc_ss_allocate*, not the C routine *malloc*. The former allows the RPC runtime library to do bookkeeping about memory management. It also ensures that the allocated memory will be automatically freed after the remote procedure completes.

(d) For ref pointers, memory on the client side must already exists, and hence no memory management required for RPs whose output parameters are ref pointers:

i. After one makes the RPC, first the server stub automatically allocates necessary memory and copies the data for the ref pointers into the new memory;

ii. Then it calls the implementation of the RP.

iii. Finally, the RP completes, output data is transmitted back to the client stub and the server stub frees the memory it allocated.

(e) Full pointers need more complex memory management on both clients and servers.

i. If a RP allocates memory for an output parameter, the server stub copies and marshalls the data. After completion of the RPC, the stub frees the memory that was allocated in the RP.

ii. When the client receives the data, the client stub allocates memory and copies the data into the new memory. It is the client application's responsibility to free the memory allocated by the client stub.

(f) Example 5-7: Memory management in RPs, p.114

i. The additional char is needed to accommodate the null char;

ii. The RP calls the stub support routine *rpc_ss_allocate* to allocate memory in the RP.

iii. When the RP finishes, the server stub automatically frees memory allocated by *rpc_ss_allocate*.

iv. When the RP returns to the client, the client stub automatically allocates memory for the returned string.

v. The client application code must explicitly free the memory allocated by the client stub.

vi. Stub support routine *rpc_ss_free* is more sophisticated than *free*.

(g) Memory used by context handles will not be allocated or freed with *rpc_ss_allocate* or *rpc_ss_free*.

c. Allocating memory for conformant arrays

* The *whatare_subparts* procedure of the inventory application allocates memory for a conformant array in a structure, and returns a copy of the conformant structure to the client.
* That procedure is shown on p.115.

i. The output parameters are ref pointers, which must have memory allocated in the client prior to the call.

ii. Therefore we have to use a full pointer in order for new memory to be automatically allocated by client stub for the ∗∗*subparts* structure where *whatare_subparts* procedure returns.

iii. A pointer to pointer is required so that the ref pointer points to a full pointer, which in turn points the structure.

* Example 5-8: Conformant array allocation in a RP,p.116

i. The allocated memory includes the size of the conformant array plus enough memory for all the elements of the conformant array. The conformant structure generated by the IDL compiler already has an array of one element, so the new memory allocated for the array elements is one less than the number in the array;

ii. Use RPC stub support routine *rpc_ss_allocate* to allocate memory so bookkeeping is maintained for memory management and the server stub will free the memory automatically after the RPC completes.

iii. When the data for conformant structure returns to the client, the client stub allocates memory and copies the data into the new memory. The client application code uses the data and frees the memory (p.117, top).

(4) Compiling and linking servers

a. Files and libraries used to produce a server: Fig.5-2,p.117

b. Server auxiliary files: caused by the following features in an IF:

* Self-referencing pointers in the IF def;
* The *pipe* data type in the IF def;
* The *out_of_line* attribute in an ACF.

c. Compile all server application modules with the C compiler;

d. Link the server application and server stub object files with the DCE libraries to produce the executable server.

6. Using a name service

(1) Naming in DCE

(2) Environment Variables

(3) Server Entries

(4) Group Entries

(5) Profile Entries