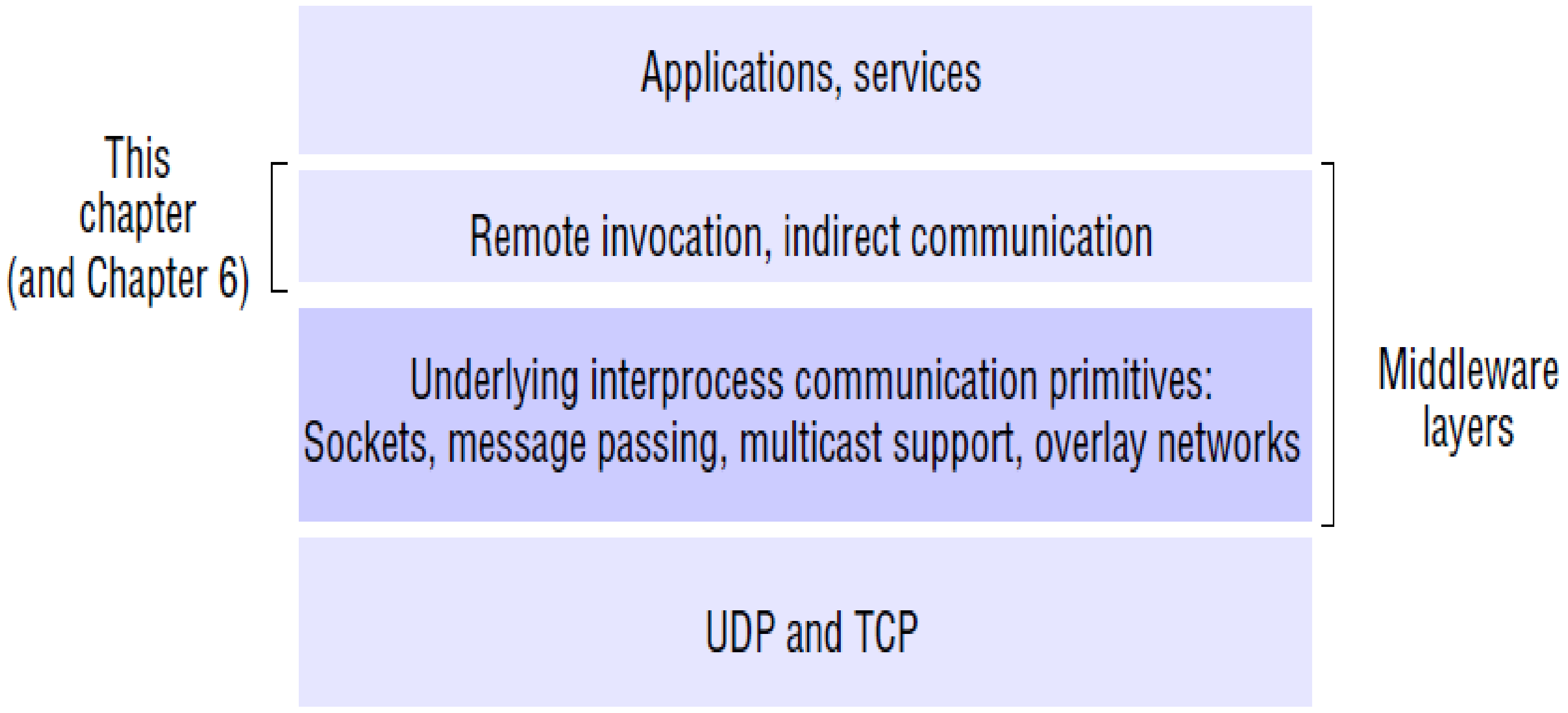
Chapter -5
Remote Invocation

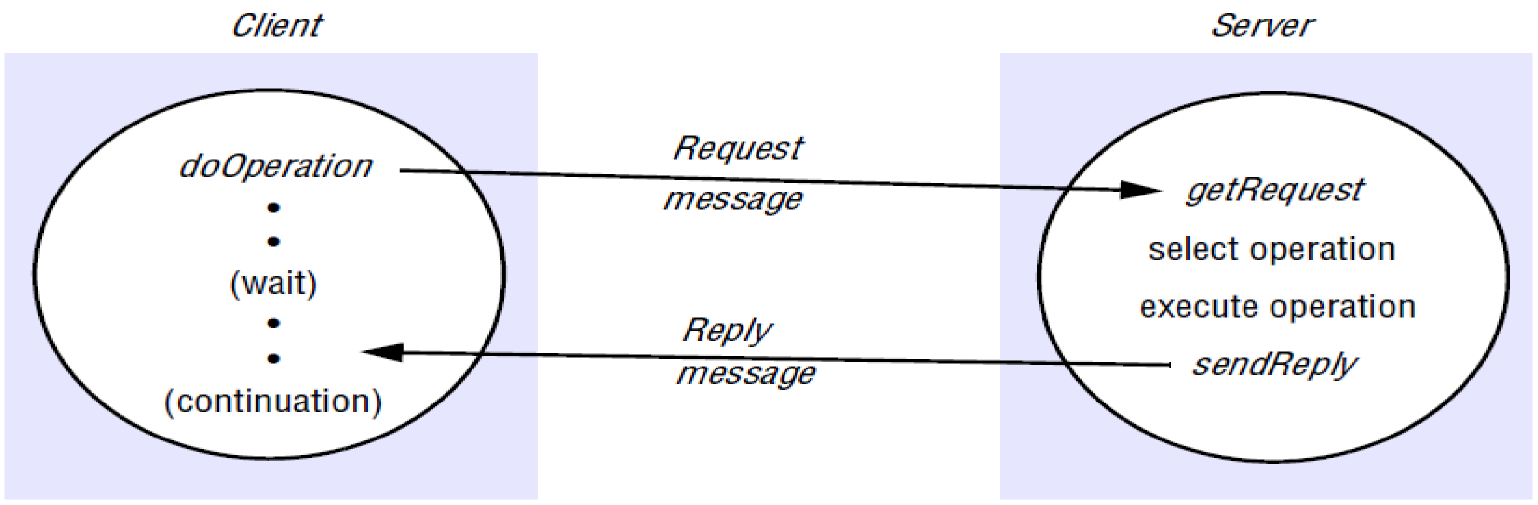
Contents

Figure 5.1 Middleware layers


Applications, services
This chapter (and Chapter 6)
Remote invocation, indirect communication
Underlying interprocess communication primitives: Sockets, message passing, multicast support, overlay networks
Middleware layers
UDP and TCP

This chapter is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system,

Figure 5.2 Request-reply communication



Request-reply protocols represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing.

In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation, and also provide direct support for RPC and RMI.

Middleware

- A systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware.
- Its primary role is to
  1. Functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate.
  2. Enable and simplify the integration of components developed by multiple technology suppliers.

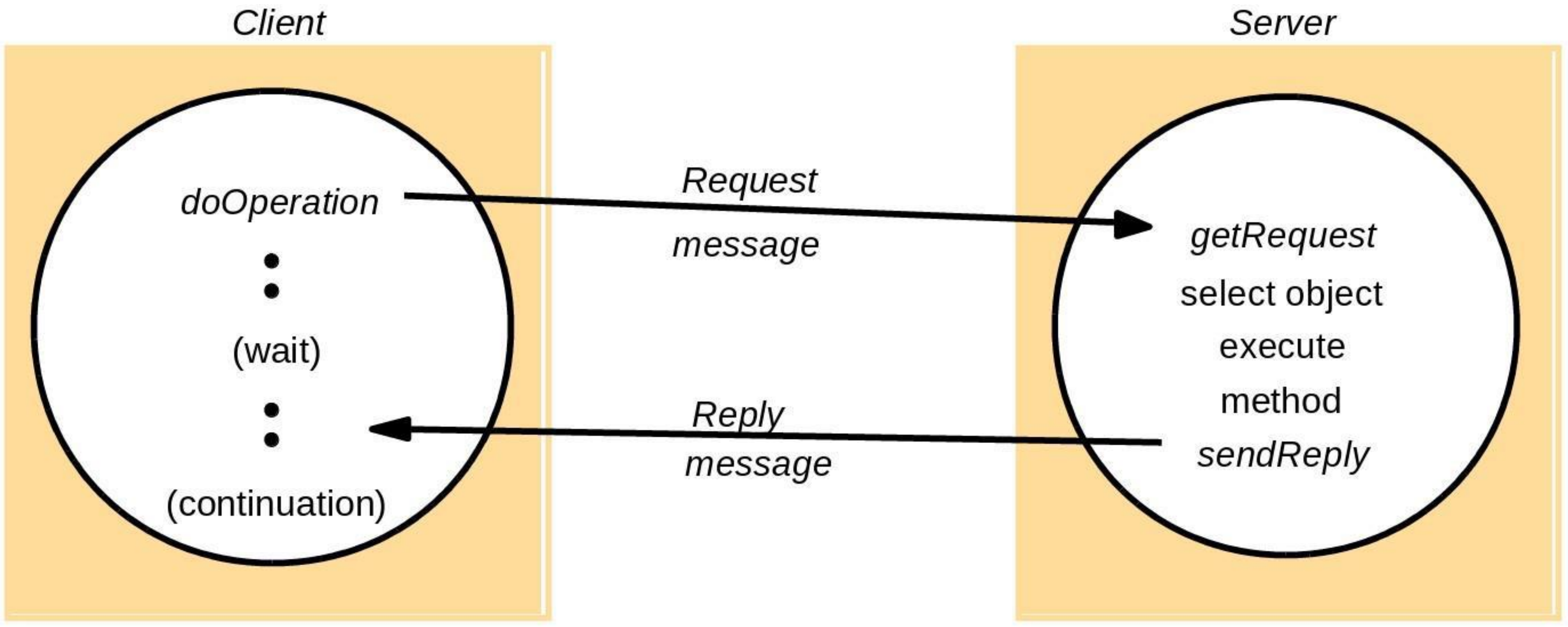
Distribution Middleware

- Defines higher-level distributed programming models whose reusable APIs and components automate and extend the native OS network programming capabilities encapsulated by host infrastructure middleware.
- Enables clients to program distributed applications much like stand-alone applications, i.e., by invoking operations on target objects without hard-coding dependencies on their location, programming language, OS platform, communication protocols and interconnects, and hardware.
- Common Object Request Broker Architecture (CORBA) which is an open standard for distribution middleware that allows objects to interoperate across networks regardless of the language in which they were written or the platform on which they are deployed.

The request-reply protocol

- Support low-level client-server interactions.
- Usually synchronous and reliable
- Built on top of send and receive operations
- Usually use UDP datagrams, could use TCP streams
- Three primitives
  - doOperation by clients to invoke remote op.; together with additional arguments; return a byte array. Marshaling and unmarshaling!
  - getRequest by server process to acquire service requests; followed by
  - sendReply send reply to the client

The request-reply protocol



Operations of the request-reply

- ***public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)***
  - sends a request message to the remote server and returns the reply.
  - The arguments specify the remote server, the operation to be invoked and the arguments of that operation.
- ***public byte[] getRequest ()***
  - acquires a client request via the server port.
- ***public void sendReply (byte[] reply, InetAddress clientHost, intclientPort)***
  - sends the reply message reply to the client at its Internet address and port.

Request-reply message structure

MessageType	*int (0=Request, 1= Reply)*
requestId	*Int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*array of bytes*

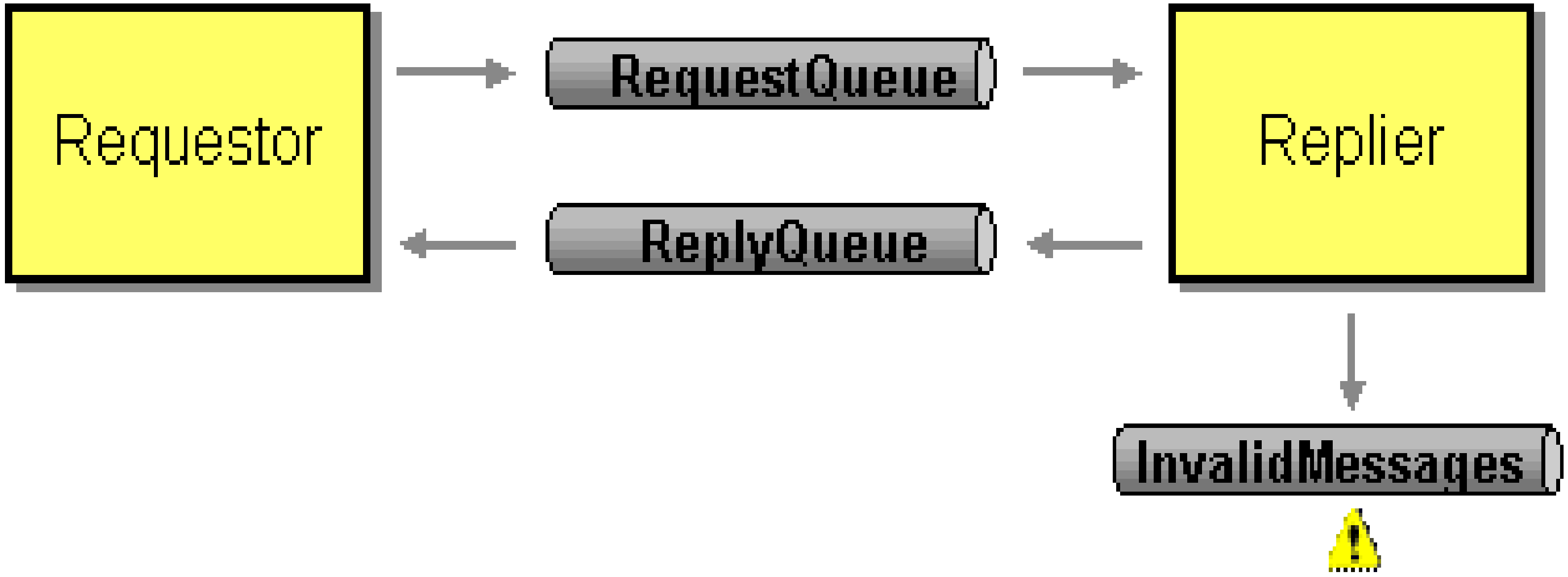
Request-reply protocols (cont.)

- Message identifiers must identify request uniquely:
  - requestId: usually a sequence counter (makes unique at client)
  - Client/sender identifier endpoint (with requestId, globally unique)
- Failure model
  - Over UDP: omission, misordering
  - Over UDP or TCP: server crash failure
- Timeouts: doOperation uses when blocked for reply
- Duplicate request msgs: server may get >1 times

Example of Request-reply protocol

- An example of how to use messaging, implemented in Java Message Service (JMS).
- JMS API is a messaging standard that allows application components based on the Java Platform Enterprise Edition (Java EE) to create, send, receive, and read messages.
- It enables distributed communication that is loosely coupled, reliable, and asynchronous.

JMS Request/Reply Example



•It shows how to implement request-reply, where a requestor application sends a request, a replier application receives the request and returns a reply, and the requestor receives the reply.
• It also shows how an invalid message will be rerouted to a special channel.

JMS Request/Reply Example

- Two main classes:
  - **Requestor** — A Message Endpoint that sends a request message and waits to receive a reply message as a response.
  - **Replier** — A Message Endpoint that waits to receive the request message; when it does, it responds by sending the reply message.
- The Requestor and the Replier will each run in a separate Java virtual machine (JVM), which is what makes the communication distributed.
- This example assumes that the messaging system has these three queues defined:
  - **jms/RequestQueue** — The Queue the Requestor uses to send the request message to the Replier.
  - **jms/ReplyQueue** — The Queue the Replier uses to send the reply message to the Requestor.
  - **jms/InvalidMessages** — The Queue that the Requestor and Replier move a message to when they receive a message that they cannot interpret.

JMS Request/Reply Example

When the Requestor is started in a command-line window, it starts and prints output like this:

Sent Request:

```
request Time: 1048261736520 ms
Message ID: ID:_XYZ123_1048261766139_6.2.1.1
Correl. ID: null (Correlation id)
 Reply to: com.sun.jms.Queue: jms/ReplyQueue
Contents: Hello world.
```

NB: (This shows is that the Requestor has sent a request message.

Notice that this works even though the Replier isn't even running and therefore cannot receive the request.)

JMS Request/Reply Example

When the Replier is started in another command-line window, it starts and prints output like this:

Received Request:

```
request Time: 1048261766790 ms
Message ID: ID:_XYZ123_1048261766139_6.2.1.1
Correl. ID: null (Correlation id)
 Reply to: com.sun.jms.Queue: jms/ReplyQueue
Contents: Hello world.
```

Sent Reply

```
request Time: 1048261766850 ms
Message ID: ID:_XYZ123_1048261758148_5.2.1.1
Correl. ID: ID:_XYZ123_1048261766139_6.2.1.1
 Reply to: null
Contents: Hello world.
```

JMS Request/Reply Example

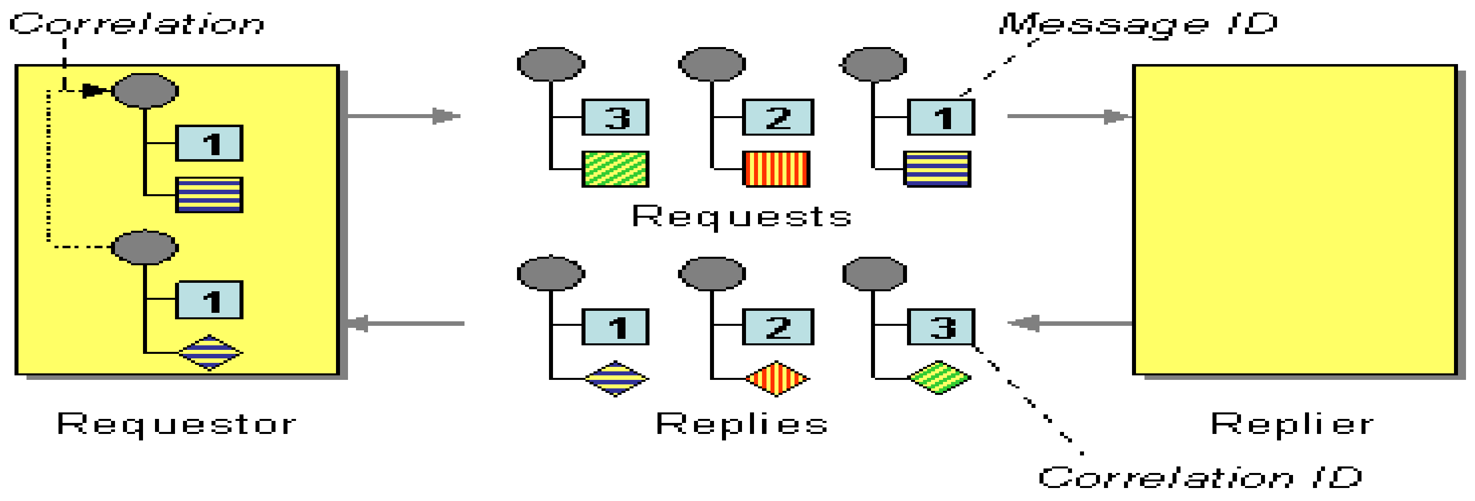
- This shows that the Replier received the request message and sent a reply message.
- Four interesting points to notice in this request and reply messages.
  - First, notice the request send and received timestamps; the request was received after it was sent (30270 ms later).
  - Second, notice that the message ID is the same in both cases, because it's the same message.
  - Third, notice that the contents, "Hello world," are the same, which is very good because this is the data being transmitted and it has got to be the same on both sides.
  - The queue named "jms/ReplyQueue" has been specified in the request message as the destination for the reply message (an example of the Return Address pattern).

JMS Request/Reply Example

- Comparing the output from receiving the request to that for sending the reply.
- Five points to be noticed:
- First, the reply was not sent until after the request was received (60 ms after).
- Second, the message ID for the reply is different from that for the request; this is because the request and reply messages are different, separate messages.
- Third, the contents of the request have been extracted and added to the reply.
- Forth, the reply-to destination is unspecified because no reply is expected (the reply does not use the Return Address pattern).
- Fifth, the reply's correlation ID is the same as the request's message ID.

JMS Request/Reply Example

- Correlation Identifier
  - How does a requestor that has received a reply know which request this is the reply for?



Each reply message should contain a *Correlation Identifier*, a unique identifier that indicates which request message this reply is for.

JMS Request/Reply Example

Finally, back in the first window, the requester received the reply:

Received reply

```
request Time: 1048261797060 ms
Message ID: ID:_XYZ123_1048261758148_5.2.1.1
Correl. ID: ID:_XYZ123_1048261766139_6.2.1.1
 Reply to: null
Contents: Hello world.
```

- The reply was received after it was sent (30210 ms).
- The message ID of the reply was the same when it was received as it was when it was sent, which proves that it is indeed the same message.
- The message contents received are the same as those sent.
- The correlation ID tells the requestor which request this reply is for.

Request-reply protocol (cont.)

- Lost reply messages
  - Idempotent operation: just redo
  - Else store reply history (how many? How to use?)

Remote Procedure Call (RPC) exchange protocols

- RPC mechanisms are used when a computer program causes a procedure or subroutine to execute in a different address space, which is coded as a normal procedure call without the programmer specifically coding the details for the remote interaction.
- Manages low-level transport protocol, such as User Datagram Protocol, Transmission Control Protocol/Internet Protocol etc.

Communication Protocols For RPC

- The following are the communication protocols that are used:
- Request Protocol (R)
- Request/Reply Protocol (RR)
- The Request/Reply/Acknowledgement-Reply Protocol (RRA)

Request Protocol: RPC

- It is used in Remote Procedure Call (RPC) when a request is made from the calling procedure to the called procedure.
- After execution of the request, a called procedure has nothing to return and there is no confirmation required of the execution of a procedure.
- Asynchronous Remote Procedure Call (RPC) employs the R protocol for enhancing the combined performance of the client and server. By using this protocol, the client need not wait for a reply from the server and the server does not need to send that.
- In an Asynchronous Remote Procedure Call (RPC) in case communication fails, the RPC Runtime does not retry the request. TCP is a better option than UDP since it does not require retransmission and is connection-oriented.

Request/Reply (RR) Protocol: RPC

- The parameters and result values are enclosed in a single packet buffer in simple RPCs. The duration of the call and the time between calls are both briefs.
- This protocol has a concept base of using implicit acknowledgements instead of explicit acknowledgements.
- Here, a reply from the server is treated as the acknowledgement (ACK) for the client’s request message, and a client’s following call is considered as an acknowledgement (ACK) of the server’s reply message to the previous call made by the client.
- To deal with failure handling e.g. lost messages, the timeout transmission technique is used with RR protocol.
- If a client does not get a response message within the predetermined timeout period, it retransmits the request message.
- Exactly-once semantics is provided by servers as responses get held in reply cache that helps in filtering the duplicated request messages and reply messages are retransmitted without processing the request again.
- If there is no mechanism for filtering duplicate messages then at least-call semantics is used by RR protocol in combination with timeout transmission.

Request/Reply/Acknowledgement-Reply (RRA) Protocol

- Exactly-once semantics is provided by RR protocol which refers to the responses getting held in reply cache of servers resulting in loss of replies that have not been delivered.
- RRA Protocol is used to get rid of the drawbacks of the RR (Request/Reply) Protocol.
- The client acknowledges the receiving of reply messages and when the server gets back the acknowledgement from the client then only deletes the information from its cache.
- Because the reply acknowledgement message may be lost at times, the RRA protocol requires unique ordered message identities. This keeps track of the acknowledgement series that has been sent.

Implementation of RMI in JAVA

- A RMI application can be divided into two part, **Client** program and **Server** program.
- A **Server** program creates some remote object, make their references available for the client to invoke method on it.
- A **Client** program make request for remote objects on server and invoke method on them.
- **Stub** and **Skeleton** are two important object used for communication with remote object.

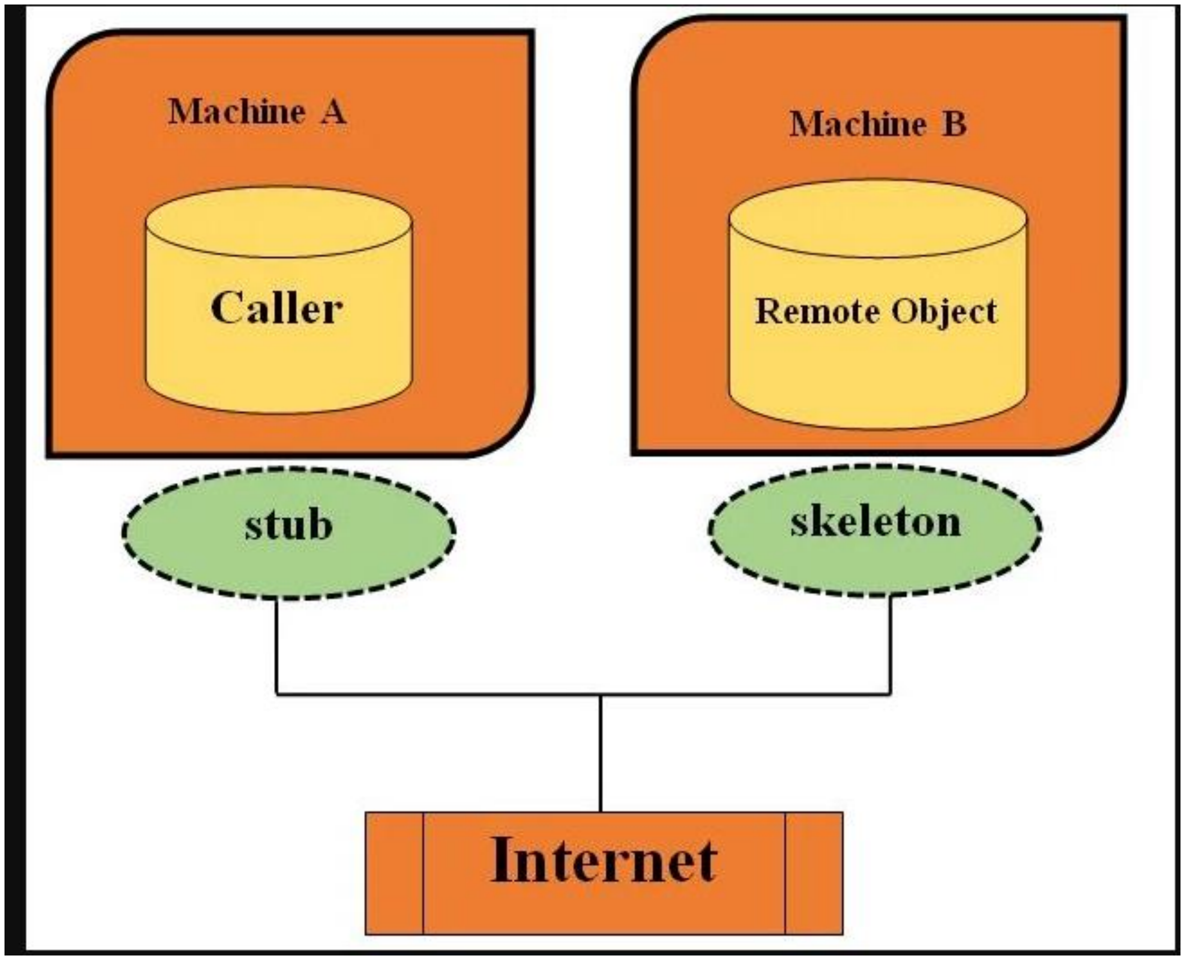
Implementation of RMI in JAVA: Stub

- In RMI, a stub is an object that is used as a Gateway for the client-side.
- All the outgoing request are sent through it.
- When a client invokes the method on the stub object following things are performed internally:
  - A connection is established using Remote Virtual Machine.
  - It then transmits the parameters to the Remote Virtual Machine. This is also known as Marshals
  - After the 2nd step, it then waits for the output.
  - Now it reads the value or exception which is come as an output.
  - At last, it returns the value to the client.

Implementation of RMI in JAVA: Skeleton

- In RMI, a skeleton is an object that is used as a Gateway for the server-side.
- All the incoming request are sent through it.
- When a Server invokes the method on the skeleton object following things are performed internally:
  - All the Parameters are read for the remote method.
  - The method is invoked on the remote object.
  - It then writes and transmits the parameters for the result. This is also known as Marshals.

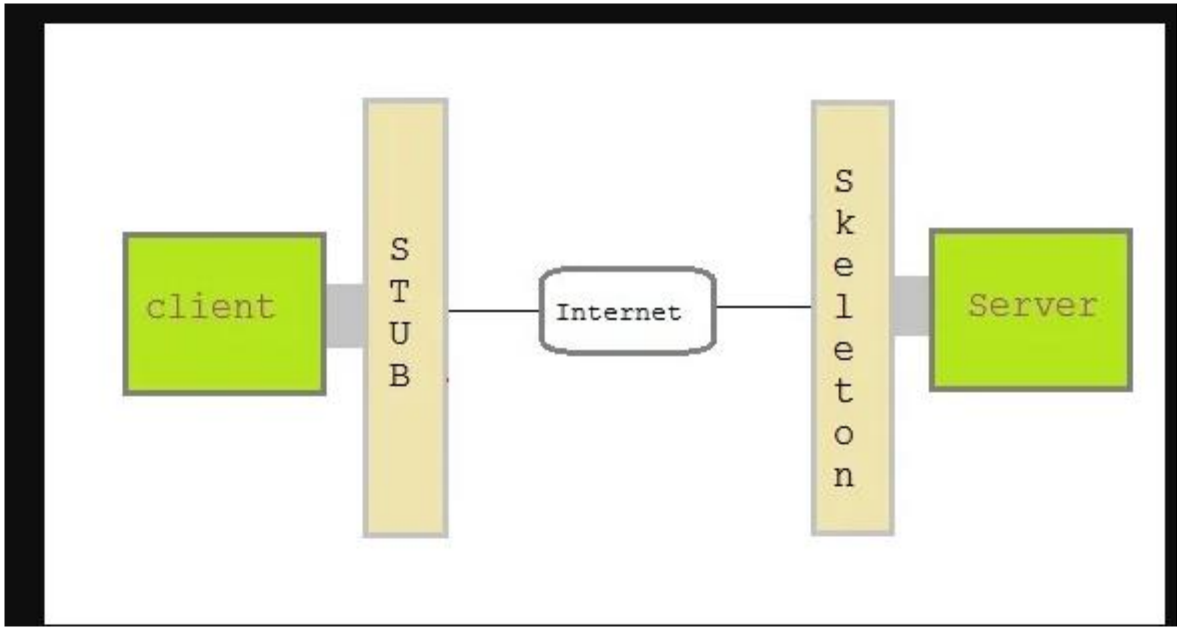
Implementation of RMI in JAVA: Stub & Skeleton



Implementation of RMI in JAVA: Stub and Skeleton

- **Stub** act as a gateway for Client program.
- It resides on Client side and communicate with **Skeleton** object.
- It establish the connection between remote object and transmit request to it.
- Skeleton object resides on server program. It is responsible for passing request from **Stub** to remote object.

Implementation of RMI in JAVA: Stub and Skeleton



Creating a Simple RMI application

- It involves following steps:
  - Define a remote interface.
  - Implementing remote interface.
  - create and start remote application
  - create and start client application

Creating a Simple RMI application: Define a remote interface

- A remote interface specifies the methods that can be invoked remotely by a client.
- Clients program communicate to remote interfaces, not to classes implementing it.
- To be a remote interface, a interface must extend the **Remote** interface of **java.rmi** package.

Creating a Simple RMI application: Define a remote interface

```
import java.rmi.*;
public interface AddServerInterface extends
  Remote
{
public int sum(int a,int b);
}
```

Creating a Simple RMI application: Implementation of remote interface

For implementation of remote interface, a class must either extend **UnicastRemoteObject** or use exportObject() method of **UnicastRemoteObject** class.

```
import java.rmi.*;
import java.rmi.server.*;
public class Adder extends UnicastRemoteObject implements AddServerInterface
{
    Adder()throws RemoteException{
    super();
}
public int sum(int a,int b)
{
    return a+b;
}
}
```

Creating a Simple RMI application: create and start remote application

- You need to create a server application and host rmi service **Adder** in it.
- This is done using rebind() method of **java.rmi.Naming** class.
- rebind() method take two arguments, first represent the name of the object reference and second argument is reference to instance of **Adder**
- *java.rmi.Naming* class contains a method to bind, unbind or rebind names with a remote object present at the remote registry. This class is also used to get the reference of the object present at remote registries or the list of name associated with this registry.
- RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** or **reBind()** methods). These are registered using a unique name known as **bind name**.
- To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).

Creating a Simple RMI application: create and start remote application

Create AddServer and host rmi service

```
import java.rmi.*;
import java.rmi.registry.*;
public class AddServer {
    public static void main(String args[]) {
        try {
                AddServerInterface addService=new Adder();
                Naming.rebind("AddService",addService);         //addService
    object is hosted with name AddService


        }
        catch(Exception e) {
                System.out.println(e);
        }
    }
}
```

Creating a Simple RMI application: create and start client application

- Client application contains a java program that invokes the lookup() method of the **Naming** class.
- This method accepts one argument, the **rmi** URL and returns a reference to an object of type **AddServerInterface**.
- All remote method invocation is done on this object.

Creating a Simple RMI application: create and start client application

```
import java.rmi.*;
public class Client {
    public static void main(String args[]) {
        try{
                AddServerInterface st =
    (AddServerInterface)Naming.lookup("rmi://"+args[0]+"/AddService");
                System.out.println(st.sum(25,8));
        }
        catch(Exception e) {
                System.out.println(e);
        }
    }
}
```

Creating a Simple RMI application: Steps to run this RMI application

- Save all the above java file into a directory and name it as "rmi“

compile all the java files by the command

```
javac *.java
```

- Start RMI registry by the command

```
start rmiregistry
```

- Run Server file by the command

```
java AddServer
```

- Run Client file in another command prompt and pass local host port number at run time
  - `java Client 127.0.0.1`