INTERPROCESS COMMUNICATION

From Chapter 4 of Distributed Systems Concepts and Design,4th Edition,

By G. Coulouris, J. Dollimore and T. Kindberg

Published by Addison Wesley/Pearson Education June 2005

Topics

- INTRODUCTION
- The API for the INTERNET PROTOCOLS
- EXTERNAL DATA REPRESENTATION
- CLIENT-SERVER COMMUNICATION

4.1 Introduction

•A process can be: Independent process or Co-operating process.

•An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.

•Processes running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity.

•**Inter process communication (IPC)** is a mechanism which allows processes to communicate each other and synchronize their actions.

Processes can communicate with each other using these two ways : Shared Memory or Message passing

4.1 Introduction

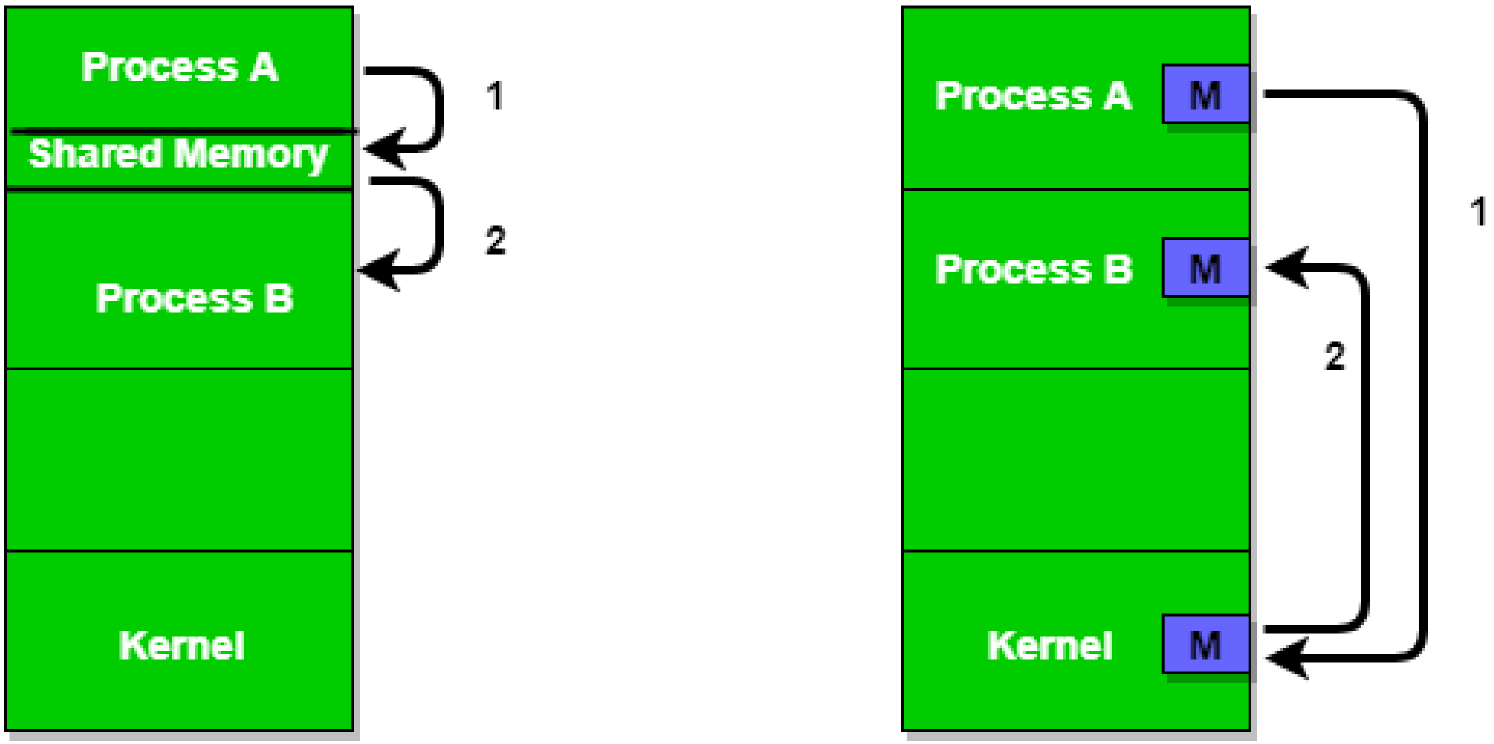


Figure 1 - Shared Memory and Message Passing

4.1 Introduction

- The java API for interprocess communication in the internet provides both datagram and stream communication.
- The communication patterns that are most commonly used in distributed programs:
  - Client-Server communication
    - The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).
  - Group communication
    - The same message is sent to several processes
    - Group multicast communication in which one process in a group transmits the same message to all members of the group

Introduction

- This chapter is concerned with middleware.

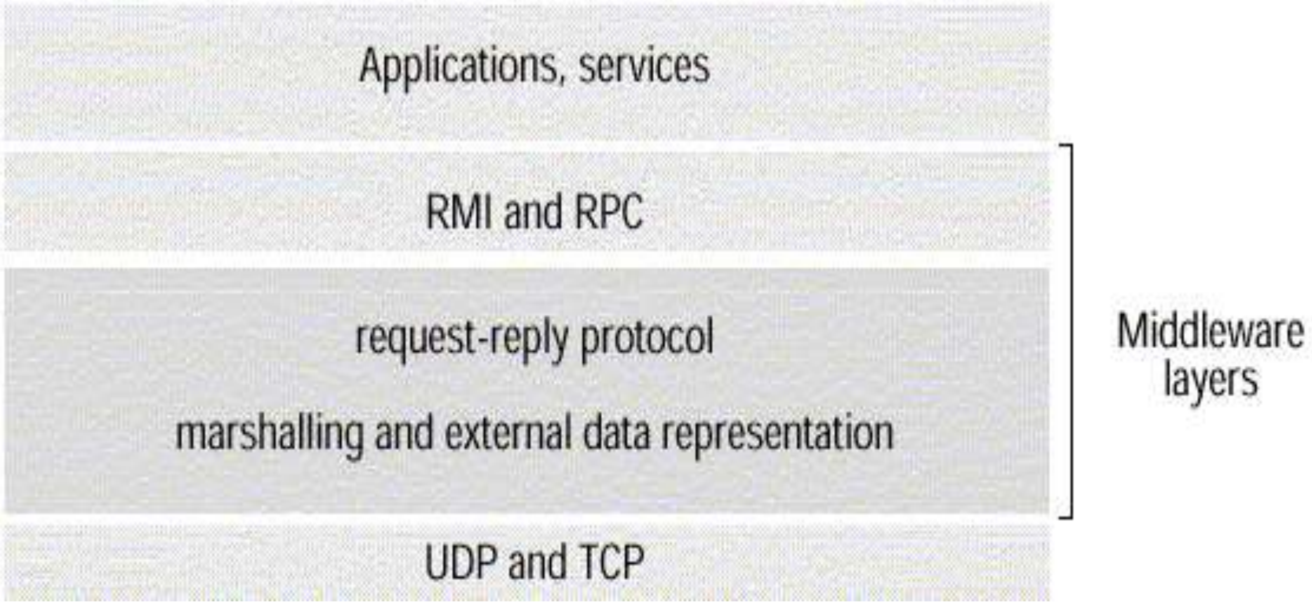


Figure 1. Middleware layers

Introduction

- Remote Method Invocation (RMI)
  - It allows an object to invoke a method in an object in a remote process.
    - E.g. CORBA and Java RMI
- Remote Procedure Call (RPC)
  - It allows a client to call a procedure in a remote server.

Introduction

- The application program interface (API) to UDP provides a message passing abstraction.
    - Message passing is the simplest form of interprocess communication.
    - API enables a sending process to transmit a single message to a receiving process.
    - The independent packets containing theses messages are called datagrams.
    - In the Java and UNIX APIs, the sender specifies the destination using a socket.

Introduction

- Socket is an indirect reference to a particular port used by the destination process at a destination computer.

- The application program interface (API) to TCP provides the abstraction of a two-way stream between pairs of processes.
- The information communicated consists of a stream of data items with no message boundaries.

4.2 The API for the Internet Protocols

- The CHARACTERISTICS of INTERPROCESS COMMUNICATION
- SOCKET
- UDP DATAGRAM COMMUNICATION
- TCP STREAM COMMUNICATION

4.2.1 The Characteristics of Interprocess Communication

1. Synchronous and asynchronous communication

- In the synchronous form, both send and receive are blocking operations. Eg.Continous Chatting
- In the asynchronous form, the use of the send operation is non-blocking and the receive operation can have blocking and non-blocking variants.

  Eg.WhatsApp

The Characteristics of Interprocess Communication

2. Message destinations

- A local port is a message destination within a computer, specified as an integer.
- A port has an exactly one receiver but can have many senders.

The Characteristics of Interprocess Communication

3.Reliability

- A reliable communication is defined in terms of validity and integrity.
- A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost.
- For integrity, messages must arrive uncorrupted and without duplication.

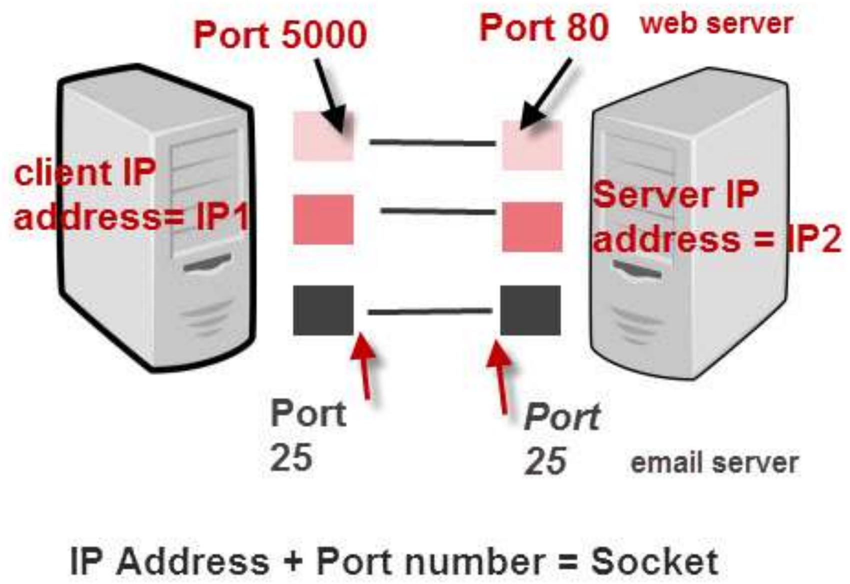
4.Ordering: Some applications require that messages to be delivered in sender order.

4.2 Sockets

- A **socket** can be thought of as an endpoint in a two-way communication channel. Eg:telephone Call
- Internet IPC mechanism of Unix and other operating systems (BSD Unix, Solaris, Linux, Windows NT, Macintosh OS) Processes can send and receive messages via a socket.
- Sockets need to be bound to a port number and an internet address in order to send and receive messages.
- Each socket has a transport layer protocol (TCP or UDP).

Sockets

- Messages sent to some internet address and port number can only be received by a process using a socket that is bound to this address and port number.



TCP/IP Ports And Sockets

Sockets

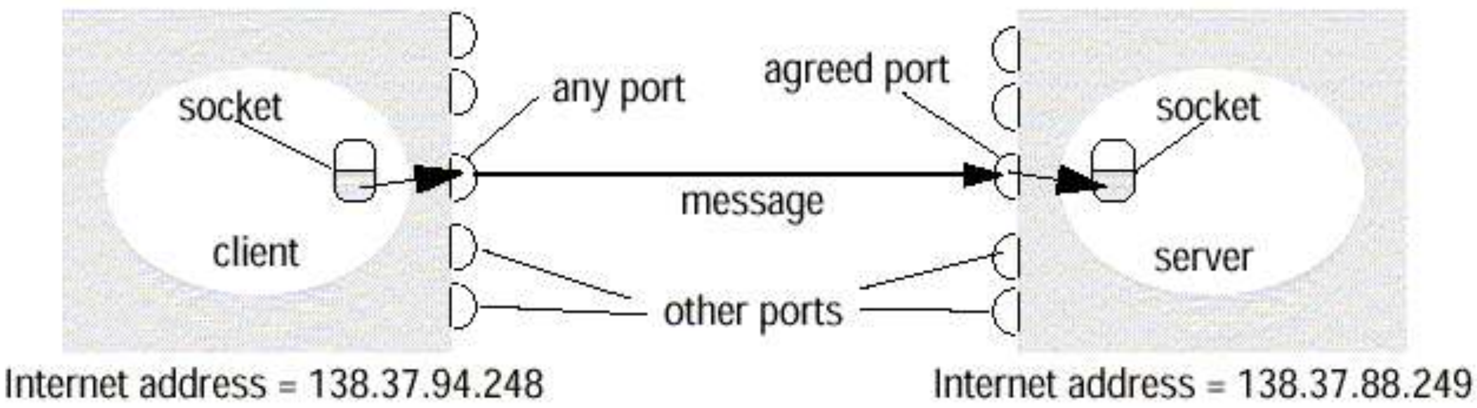


Figure 2. Sockets and ports

4.2.1UDP Datagram Communication

- UDP datagram properties
  - No guarantee of order preservation
  - Message loss and duplications are possible
- Necessary steps
  - Creating a socket
  - Binding a socket to a port and local Internet address
    - A client binds to any free local port
    - A server binds to a server port
- Receive method
  - It returns Internet address and port of sender, plus message.

UDP Datagram Communication

- Issues related to datagram communications are:
  - Message size
    - IP allows for messages of up to 2^{16} bytes.
    - Most implementations restrict this to around 8k bytes.
    - Any application requiring messages larger than the maximum must fragment.
    - If arriving message is too big for array allocated to receive message content, truncation occurs.

UDP Datagram Communication

- Blocking
  - Send: non-blocking
    - upon arrival, message is placed in a queue for the socket that is bound to the destination port.
  - Receive: blocking
    - Pre-emption by timeout possible
    - If process wishes to continue while waiting for packet, use separate thread
- Timeout
- Receive from any

UDP Datagram Communication

- UDP datagram's suffer from following failures:
  - Omission failure
  - Messages may be dropped occasionally
  - Ordering

Java API for UDP Datagrams

- The Java API provides datagram communication by two classes:
  - Datagram Packet
    - It provides a constructor to make an array of bytes comprising:
      - Message content
      - Length of message
      - Internet address
      - Local port number

array of bytes containing message	length of message	Internet address	port number

- It provides another similar constructor for receiving a message.

Java API for UDP Datagrams

- Datagram Socket
  - This class supports sockets for sending and receiving UDP datagram.
  - It provides a constructor with port number as argument.
  - Datagram Socket methods are:
    - send and receive
    - setSoTimeout
    - connect

4.2.2 TCP Stream Communication

- The API to the TCP protocol provides the abstraction of a stream of bytes to be written to or read from.
  - Characteristics of the stream abstraction:
    - Message sizes: Application dependent
    - Lost messages:Ack,Seq No.
    - Flow control: Window size
    - Message duplication & Ordering :Seq No
    - Message destination:One time with IP &Port no.

TCP Stream Communication

- Issues related to stream communication:
  - Matching of data items:
    - data interpretation error w.r.t. use of data stream
  - Blocking: Use of Queue destination Socket
  - Threads: To avoid delay in handling clients

TCP Stream Communication

- Use of TCP
  - Many services that run over TCP connections, with reserved port number are:
    - HTTP (Hypertext Transfer Protocol)
    - FTP (File Transfer Protocol)
    - Telnet
    - SMTP (Simple Mail Transfer Protocol)

TCP Stream Communication

- Java API for TCP streams
  - The Java interface to TCP streams is provided in the classes:
    - ServerSocket
      - It is used by a server to create a socket at server port to listen for connect requests from clients.
    - Socket
      - It is used by a pair of processes with a connection.
      - The client uses a constructor to create a socket and connect it to the remote host and port of a server.
      - It provides methods for accessing input and output streams associated with a socket.

4.3 External Data Representation

- The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes.

- Irrespective of the form of communication used, the data structure must be converted to a sequence of bytes before transmission and rebuilt on arrival.

External Data Representation

- External Data Representation is an agreed standard for the representation of data structures and primitive values.

- Data representation problems are:
  - Using agreed external representation, two conversions necessary
  - Using sender's or receiver's format and convert at the other end

External Data Representation

- Marshalling
- Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- Unmarshalling
- Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.

External Data Representation

- Three approaches to external data representation and marshalling are:
  - CORBA
  - Java’s object serialization

External Data Representation

- Marshalling and unmarshalling activities is usually performed automatically by middleware layer.
- Marshalling is likely error-prone if carried out by hand.

4.3.1 CORBA Common Data Representation (CDR)

- CORBA Common Data Representation (CDR)
  - CORBA CDR is the external data representation defined with CORBA 2.0.
  - It consists 15 primitive types:
    - Short (16 bit)
    - Long (32 bit)
    - Unsigned short
    - Unsigned long
    - Float(32 bit)
    - Double(64 bit)
    - Char
    - Boolean(TRUE,FALSE)
    - Octet(8 bit)
    - Any(can represent any basic or constructed type)
  - Composite type are shown in Figure 8.

CORBA Common Data Representation (CDR)

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Figure 8. CORBA CDR for constructed types

CORBA Common Data Representation (CDR)

Figure 9 shows a message in CORBA CDR that contains the three fields of a struct whose respective types are string, string, and unsigned long.

example: struct with value {'Smith', 'London', 1934}

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1934	*unsigned long*

Figure 9. CORBA CDR message

4.3.2 Java object serialization

- In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.
- An object is an instance of a Java class.
  - Example, the Java class equivalent to the Person struct

```
Public class Person implements Serializable {
        Private String name;
        Private String place;
        Private int year;
        Public Person(String aName ,String  aPlace, int aYear) {
            name = aName;
            place = aPlace;
            year = aYear;
}
//followed by methods for accessing the instance variables
}
```

Java object serialization

The serialized form is illustrated in Figure 10.

Serialized values				Explanation
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

Figure 10. Indication of Java serialization form

Couloris,Dollimore and Kindberg Distributed Systems: Concepts & Design Edn. 4 , Pearson Education 2005

Remote Object References

- Remote object references are needed when a client invokes an object that is located on a remote server.
- A remote object reference is passed in the invocation message to specify which object is to be invoked.
- Remote object references must be unique over space and time.

Remote Object References

- In general, may be many processes hosting remote objects, so remote object referencing must be unique among all of the processes in the various computers in a distributed system.
- generic format for remote object references is shown in Figure 11.

32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface of remote object

Figure 11. Representation of a remote object references

Remote Object References

- internet address/port number: process which created object
- time: creation time
- object number: local counter, incremented each time an object is created in the creating process
- interface: how to access the remote object (if object reference is passed from one client to another)

4.4 Client-Server Communication

- The client-server communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- Asynchronous request-reply communication is an alternative that is useful where clients won't block & can afford to retrieve replies later.

Client-Server Communication

- Protocol often built over UDP datagram's
- UDP protocol avoids unnecessary overheads associated with TCP(Stream) protocol

1. acknowledgements are redundant, since requests are followed by replies;(piggybacked Ack’s)
2. Avoidance of connection establishment overhead which involves two extra pairs of msg’s.
3. No need for flow control due to small amounts of data (arguments/results) are transferred

Client-Server Communication

- The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in Figure 12.
- Illustrates RMI example

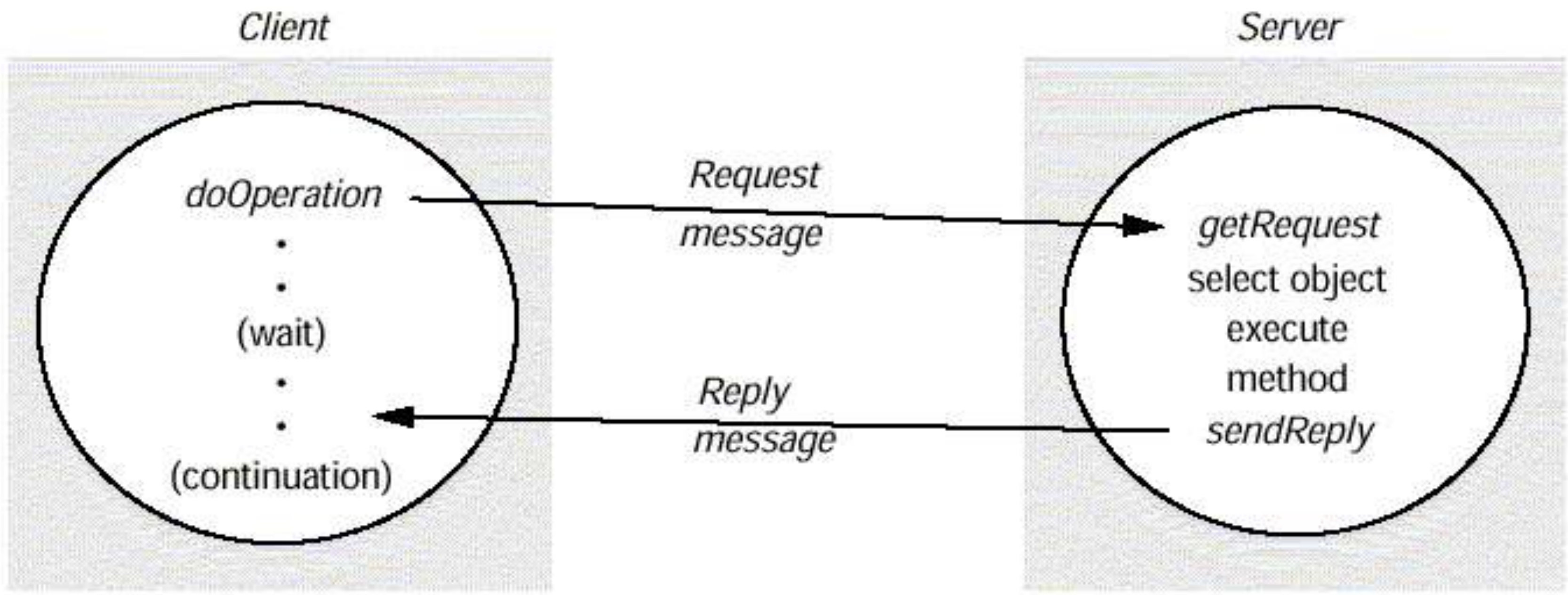


Figure 12. Request-reply communication

Client-Server Communication

- Figure 13 outlines the three communication primitives.

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();
acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
sends the reply message *reply* to the client at its Internet address and port.

Figure 13. Operations of the request-reply protocol

Client-Server Communication

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message *reply* to the client at its Internet address and port.

Figure 13. Operations of the request-reply protocol

Client-Server Communication

Three Premitives:

1.doOperation method

▪ used by the client to invoke remote operations

▪Arguments→ object & method to be invoked

▪Its result is an RMI Reply

▪Client calling doOperation marshals the arguments into an array of bytes & unmarshals the results from the array of bytes

▪Client doOperation is blocked until remote object in the server performs the requested operation & transmits a reply msg back

2.GetRequest

▪-used by a server process to acquire service requests

▪-when server has invoked the method in the object it then uses

3.SendReply is used to send reply to client.

▪-when reply msg is received doOperation is unblocked & client continues to execute

Client-Server Communication

- The information to be transmitted in a request message or a reply message is shown in Figure 14.

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*// array of bytes*

Figure 14. Request-reply message structure

Couloris,Dollimore and Kindberg Distributed Systems: Concepts & Design Edn. 4 , Pearson Education 2005

Client-Server Communication

- In a protocol message
  - The first field indicates whether the message is a request or a reply message.
  - The second field request id contains a message identifier.
  - A message identifier consists of two parts:
    - A requestId, which is taken from an increasing sequence of integers by the sending process
    - An identifier for the sender process, for example its port and Internet address
  - The third field is a remote object reference .
  - The forth field is an identifier for the method to be invoked followed by arguments

Client-Server Communication

- Failure model of the request-reply protocol
  - If these three primitives are implemented over UDP they have the same communication failures

- Omission failure (link failures, drops/losses, missed/corrupt addresses)
- Out-of-order delivery
- Node/process down

Solved by

- Timeouts with retrans until reply is received/confirmed
- Discards of repeated requests by requestId (by server process)
- On lost reply messages, server repeats idempotent operations(eg.adding an element to set)
- Maintain history (reqid, message, client-id) or buffer replies and retrans – memory intensive

Client-Server Communication

- RPC exchange protocols(failure handling)
  - Three protocols are used for implementing various types of RPC.
    - The request (R) protocol.
    - The request-reply (RR) protocol.
    - The request-reply-acknowledge (RRA) protocol.

(Figure 15)

Client-Server Communication

Name	Messages sent by		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Figure 15. RPC exchange protocols

Client-Server Communication

- In the R protocol, a single request message is sent by the client to the server.
- The R protocol may be used when there is no value to be returned from the remote method.
- The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol.
- RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply.

Client-Server Communication

- HTTP: an example of a request-reply protocol
  - HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.
  - Client requests specify a URL that includes DNS Host name+Port no.+resource identifier on that port
  - HTTP Allows→Content Negotiation & Authentication

○Content negotiation→ negotiating for appropriate data representations between client & server

○Password Style Authentication

Client-Server Communication

- HTTP protocol steps for C/S interaction:
  - Connection establishment between client and server at the default server port or at a port specified in the URL
  - client sends a request
  - server sends a reply
  - connection closure

Client-Server Communication

- Need to establish & close connection for every requqst-reply exchange is expensive
- Request & reply are marshalled into msgs as ASCII text
- Resources can have MIME(Multipurpose Internet Mail Extension)-like structures in arguments and results
- Data is prefixed with Mime type so that recipient will know how to handle it
- Mime type specifies a type and a subtype, for example:
  - text/plain, text/html, image/gif, image/jpeg

Client-Server Communication

- HTTP methods
- Client Rqst=method+URL
    - GET
        - Requests the resource, identified by URL as argument.
        - If the URL refers to data, then the web server replies by returning the data
        - If the URL refers to a program, then the web server runs the program and returns the output to the client.

method	*URL*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure 16. HTTP request message

Client-Server Communication

- HEAD
  - This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)
  - i.e status line

Client-Server Communication

- POST
  - Specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request.
  - This method is designed to deal with:
    - Providing a block of data to a data-handling process
    - Posting a message to a bulletin board, mailing list or news group.
    - Extending a dataset with an append operation

Client-Server Communication

- PUT
  - Supplied data to be stored in the given URL as its identifier.
- DELETE
  - The server deletes an identified resource by the given URL on the server.
- TRACE
  - The server sends back the request message
- OPTIONS
  - A server supplies the client with a list of methods.
  - It allows to be applied to the given URL

Client-Server Communication

- A reply message specifies
  - The protocol version
  - A status code
  - Reason
  - Some headers
  - An optional message body

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Figure 17. HTTP reply message

Client-Server Communication

- ## Status codes
- 100 block→Informational→Eg. 103 – checkpoint
- 200 block→Success→ Eg.200-OK, 201-created
- 300 block→Redirection→Eg.302-Found, 304-Not Modified
- 400 block→Client Error→ Eg.404-Not Found 408-Request Timeout
- 500 block→Server error→Eg.500-Internal Server Error, 502-Bad Gateway, 503-Service Unavailable