INTERPROCESS COMMUNICATION

Reference: George Coulouris, Jean Dollimore and Tim Kindberg, "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education, 2012

Topics

- Introduction
- The API For The INTERNET PROTOCOLS
- External Data Representation
- Client-server Communication
- Group Communication

Introduction

- The java API for interprocess communication in the internet provides both datagram and stream communication.
- The two 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).

Introduction

- Group communication
  - The same message is sent to several processes.

Introduction

- This chapter is concerned with middleware.

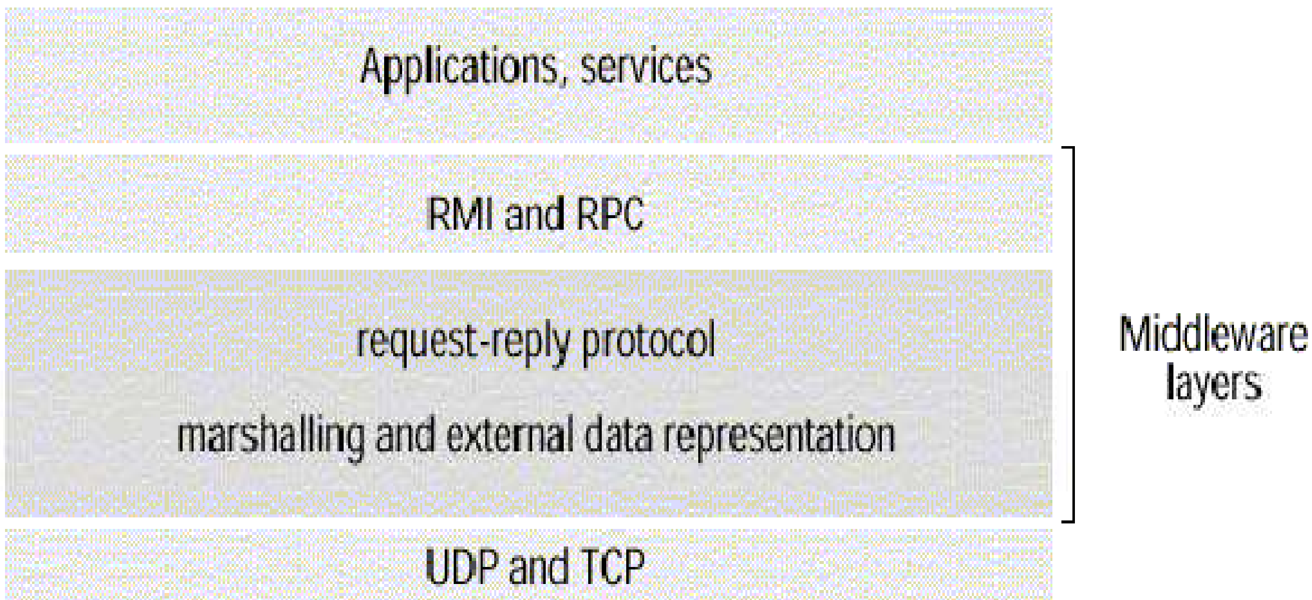

Figure 1. Middleware layers

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 these 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.

Introduction

- Request-reply protocols are designed to support client-server communication in the form of either RMI or RPC.

- Group multicast protocols are designed to support group communication.

Introduction

- Group multicast is a form of interprocess communication in which one process in a group of processes transmits the same message to all members of the group.

The API for the Internet Protocols

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

The Characteristics of Interprocess Communication

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

The Characteristics of Interprocess Communication

- 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

- 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.

The Characteristics of Interprocess Communication

- Ordering
  - Some applications require that messages be delivered in sender order.

Sockets

- Internet IPC mechanism of Unix and other operating systems (BSD Unix, Solaris, Linux, Windows NT, Macintosh OS)
- Processes in the above OS 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 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.
- Processes cannot share ports (exception: TCP multicast).
- 2^16 of possible port numbers for local processes to receive messages.

Sockets

- Both forms of communication, UDP and TCP, use the socket abstraction, which provides an endpoint for communication between processes.

- Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process.

(Figure 2)

Sockets

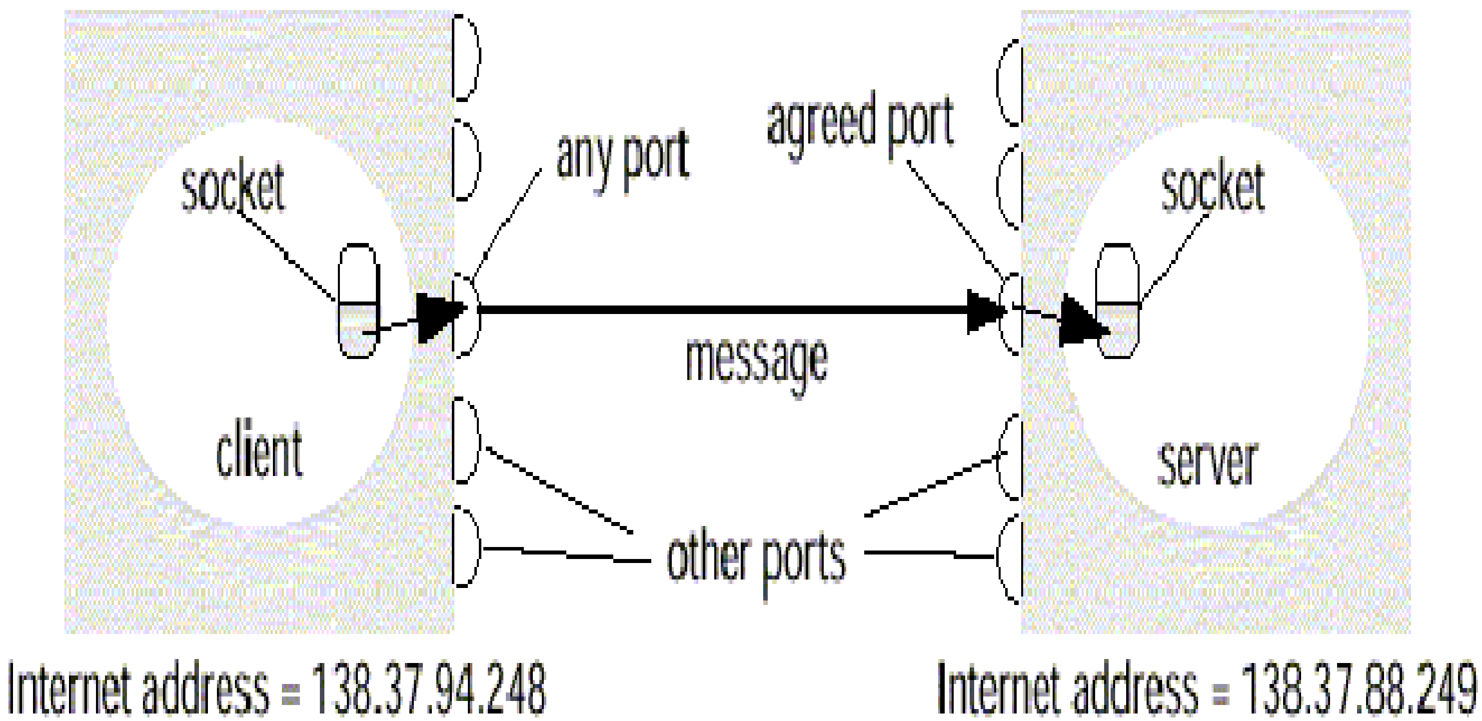

Figure 2. Sockets and ports

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

UDP 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

UDP Datagram Communication

- Receive method
  - It returns Internet address and port of sender, plus message to receipt to check from where message came.

UDP Datagram Communication

- Issues related to datagram communications are:
  - Message size
    - IP allows for messages of up to 216 bytes.
    - Most implementations restrict this to around 8 kbytes.
    - 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

UDP Datagram Communication

- Timeout
- Receive from any Origin

UDP Datagram Communication

- UDP datagrams 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:
  - `DatagramPacket`
    - It provides a constructor to make an array of bytes comprising:
      - Message content
      - Length of message
      - Internet address
      - Local port number
    - It provides another similar constructor for receiving a message.

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

Java API for UDP Datagrams

- `DatagramSocket`
  - This class supports sockets for sending and receiving UDP datagram.
  - It provides a constructor with port number as argument.
  - No-argument constructor is used to choose a free local port.
  - DatagramSocket methods are:
    - `send` and `receive`
    - `setSoTimeout`
    - `connect`

Java API for UDP Datagrams

```
import java.net.*;
import java.io.*;
public class UDPClient{
public static void main(String args[]){
// args give message contents and destination hostname
try {
DatagramSocket aSocket = new DatagramSocket(); // create socket
byte [] m = args[0].getBytes();
InetAddress aHost = InetAddress.getByName(args[1]); // DNS lookup
int serverPort = 6789;
DatagramPacket request =
new DatagramPacket(m, args[0].length(), aHost, serverPort);
aSocket.send(request); //send nessage
byte[] buffer = new byte[1000];
DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
aSocket.receive(reply); //wait for reply
System.out.println("Reply: " + new String(reply.getData()));
aSocket.close();
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
} finally{if (aSocket !=null)aSocket.close()}
}
}
```

Figure 3. UDP client sends a message to the server and gets a reply

Java API for UDP datagrams

```
import java.net.*;
import java.io.*;
public class UDPServer{
public static void main(String args[]){
DatagramSocket aSocket = null;
try {
      aSocket = new DatagramSocket(6789);
byte []buffer = new byte[1000];
While(true){
DatagramPacket request =new DatagramPacket(buffer, buffer.length);
aSocket.receive(request);
DatagramPacket reply = new DatagramPacket(request.getData();
    request.getLength(),request.getAddress(), request.getPort();
aSocket.send(reply);
}
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
}finally{if (aSocket !=null)aSocket.close()}
}
}
```

Figure 4. UDP server repeatedly receives a request and sends it back to the client

Java API for UDP Datagrams

Example

– The process creates a socket, sends a message to a server at port 6789 and waits to receive a reply.

Java API for UDP Datagrams

Example

– The process creates a socket, bound to its server port 6789 and waits to receive a request message from a client.

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
    - Lost messages
    - Flow control
    - Message Duplication and Ordering
    - Message destinations

TCP Stream Communication

- Issues related to stream communication:
  - Matching of data items
  - Blocking
  - Threads

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.

Java API for UDP Datagrams

Example

– The client process creates a socket, bound to the hostname and server port 6789.

Java API for UDP Datagrams

Example

– The server process opens a server socket to its server port 6789 and listens for connect requests.

TCP Stream Communication

```
import java.net.*;
import java.io.*;
public class TCPServer {
      public static void main (String args[]) {
         try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
             while(true) {
             Socket clientSocket = listenSocket.accept();
             Connection c = new Connection(clientSocket);
             }
            } catch(IOException e) {System.out.println("Listen socket:"+e.getMessage());}
}
}
```

Figure 6. TCP server makes a connection for each client and then echoes the client's request

TCP Stream Communication

```
class Connection extends Thread {
DataInputStream in;
DataOutputStream out;
Socket clientSocket;
public Connection (Socket aClientSocket) {
   try {
   clientSocket = aClientSocket;
    in = new DataInputStream( clientSocket.getInputStream());
    out =new DataOutputStream( clientSocket.getOutputStream());
    this.start();
} catch(IOException e){System.out.println("Connection:"+e.getMessage());}
}
public void run(){
   try {                   // an echo server
      String data = in.readUTF();
      out.writeUTF(data);
  } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
 } catch (IOException e) {System.out.println("readline:"+e.getMessage());}
} finally {try{clientSocket.close();}catch(IOException e){/*close failed*/}}
}
}
```

Figure 7. TCP server makes a connection for each client and then echoes the client's request

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 of 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
  - XML

External Data Representation

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

CORBA Common Data Representation (CDR)

- CORBA Common Data Representation (CDR)
  - CORBA CDR is the external data representation for the structured and primitive types.
- It can be used by a variety of programming languages
  - 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)

- Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 8.

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.

CORBA Common Data Representation (CDR)

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

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

Java object serialization

- Java's object serialization, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

- In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.

Java object serialization

- 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
}
```

To serialize the *Person object,* create an instance of the class ***ObjectOutputStream*** *and invoke its* ***writeObject*** *method,* passing the *Person object as its argument.*
To deserialize an object from a stream of data, open an ***ObjectInputStream*** *on the stream and use its* ***readObject*** *method to reconstruct*
the original object.

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

XML

- XML (Extensible Markup Language), which defines a textual format for representing structured data.
- It was originally intended for documents containing textual self-describing structured data.
- Documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services

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)