Lecture 5
Interprocess Communications
(July-21,22,27,2020. Chapter 4)

Lecture Outline

1. Introduction
2. TCP/IP API and Java UDP/TCP API
3. External data representation and marshalling
4. Remote object reference
5. Group communications
6. Network virtualization
7. Message passing example: MPI

1. Introduction

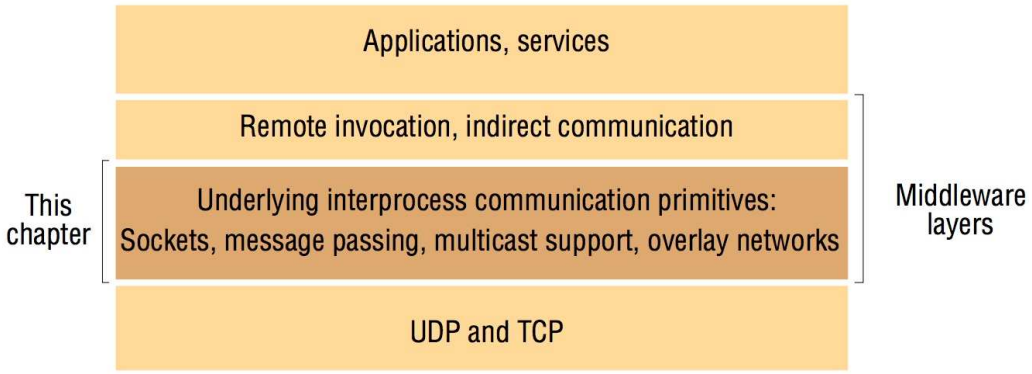

Figure 27: Middleware layers (Fig.4.1, p.146)

(1) Facilities for interprocess communications are essential for DSs.

(2) Facilities for interprocess communications can be classified in terms of abstraction levels:

https://en.wikipedia.org/wiki/Transport_Layer_Interface

a. Basic TCP/IP APIs: *Berkeley sockets* and *SVR4 TLIs*. (provided by AT&T UNIX System V)
https://en.wikipedia.org/wiki/Berkeley_sockets

b. RMI (remote method invocation) supported by certain object oriented programming languages or development tools such as $C++$, Java, and CORBA.什么是CORBA?

c. Remote procedure calls: SUN RPC, OSF DCE (distributed computing environment) RPC

(3) Characteristics of IPC

a. IPC involves message exchanges among processes

b. Messages

   (a) A message consists of two parts: the data part, provided by the user, and the control part, attached by the system.

   (b) Messages can be structured or unstructured

   (c) Basic message passing primitives

      i. **send** *expression-list* **to** *destination-identifier*

      ii. **receive** *variable-list* **from** *source-identifier*

c. Synchronous (blocking) and asynchronous (non-blocking) communications.

   (a) Two queues, a *send queue* and a *receive queue* may be associated with each process.

   (b) The action of sending a message causes the message to be transmitted and eventually to be queued to the receive message queue of destination process.

   (c) A communication is *synchronous* if the the sender and receiver synchronize at every message. Both *send* and *receive* are block operations.

      i. A *send* operation will block the issuing process until the corresponding *receive* operation is performed at the receiver.

      ii. A *receive* operation will block the issuing process until the a message from the specified destination arrives.

   (d) A communication is *asynchronous* if the *send* operation will not block the issuing process: the process can proceed as soon as the the message to be transmitted is copied to a local buffer. The *receive* operation can be either blocking or non-blocking.

      i. The blocking receive operation behaves the same as in synchronous comm.

      ii. The non-blocking receive, if a desired message hasn't arrived yet, after a receiving process issues a *receive* operation, the process will continue to proceed with its other operations. The system will fill a buffer (provided by the receiving process in the receive operation) later and notify the process after that.

   (e) Discussions

      i. Non-blocking communication is more flexible and efficient. However implementation of receiving processes is much harder. Buffers *may be* needed.

      ii. Blocking communications are just the opposite.

      iii. For the non-blocking receiving variant of the non-blocking communication, an extra notification is needed, which interrupts the running process. Few DSs adopts this variant.

      iv. Parallelism may be lost in blocking communications

      v. In systems where support of threads is available, *threads* can be used in synchronous communications to enhance performance.

(4) Quality of IPC

a. *Reliability*. An IPC facility that delivers messages un-corrupted and un-duplicated is *reliable*.

b. *Ordering*. Messages sent by a sending process may or may not arrive same as their sending order.

c. *Usability*. An IPC facility is *highly usable* it supports high degree of *location transparency*. 这里什么意思?

d. *Timeness and efficiency*. This demands quick and efficient delivery of messages.

e. Notes:

(a) Quality of IPC facilities is closely related to the quality of services at the transport layer.

(b) IPC facilities can improve their service qualities on top of services by the transport layer.

2. TCP/IP API and Java UDP/TCP API

(1) Identification of message destinations

a. Destinations of messages are identified by the *IP address* and *port number*: (*remote IP address, remote port*).

(a) In most cases a message also has to carry (*local IP address, local port*) to allow a remote destination to send replies.

(b) A process may use multiple ports to send receive messages simultaneously.

(c) Port numbers of services should be publicized by servers to clients.

b. Other means of destination identification. The use of a (*IP address, port*) pair voids the location transparency. Several other methods support location transparency:

(a) A *binder* may be supported in some DC application.

· A binder is a *service resolver*: given the name of service, it will return the server locations. https://en.wikipedia.org/wiki/Open_Network_Computing_Remote_Procedure_Call https://en.wikipedia.org/wiki/Portmap

· Sometimes the availability of a binder is essential. This is particularly true in a cloud. https://baike.baidu.com/item/portmap/8511094

· Example of binders: portmapper service; Java RMI RMIRegistry; DCE name services https://en.wikipedia.org/wiki/DARPA https://en.wikipedia.org/wiki/Distributed_Computing_Environment

(b) Some OSs such Mach provide location independent identifiers for message destinations, translating identifiers into lower-level addresses such as (IP address, port). https://en.wikipedia.org/wiki/Mach_(kernel)

原理
https://www.tutorialspoint.com/java_rmi/java_rmi_introduction.htm
https://www.mi1k7ea.com/2019/09/01/Java-RMI原理与使用/
https://developer.aliyun.com/article/66673

例子
https://www.yiibai.com/java_rmi/java_rmi_application.html
https://sites.cs.ucsb.edu/~cappello/lectures/rmi/helloworld.shtml

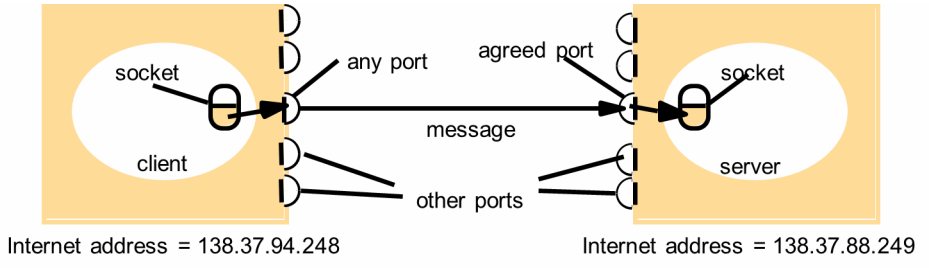

Figure 28: Sockets and ports (Fig.4.2, p.149)

(2) The Berkeley socket API

a. The *socket* API: a socket is an end point of a communication. *socket* API originated from Berkeley UNIX and are now available on Windows and Macintosh OSs.

b. Socket supports both reliable communications (connection oriented) through TCP API and un-reliable comm. (connectionless) through UDP API.

c. Illustration of *ports* and *sockets*: Fig.4.2, p.149.

d. UDP datagram communications

(a) Characteristics

i. UDP is a connectionless transport layer comm. protocol. It provides unreliable datagram delivery service.

ii. *Message sizes*: a message can be up to 2^{16} bytes (however, many implementations impose a smaller size 8 kbytes). A receiving process has to specify the maximum size of messages to be received.

iii. UDP employs *non-blocking send* and *blocking receive*.

iv. UDP relies IP to deliver datagrams. It does not handle timeouts by itself. Applications should handle timeout according to their own requirements.

v. UDP API provides several different primitives (functions) to send and receive messages. Several pairs of them allow *receive from any* semantics: they do not specify an origin of messages to be received through a socket. Any messages can be delivered and the system will fill the (IP address, port) pair of the sending process.

(b) Major UDP API functions:

i. *socket, bind* – for comm. management. 有什么区别

ii. *sendto, recvfrom, send, recv, read, write, sendv, writev, sendmsg, recvmsg* – for sending and receiving messages.

(c) Failure model:

i. Cf. Lecture 3 for failure model: it defines the ways in which failures may occur. Two properties of reliable communications: *validity* (messages are eventually delivered) and *integrity* (corrupted messages and duplicate messages can be detected).

ii. UDP provides an optional checksum that lets a receiving application verify if the contents of a datagram is corrupted. Notice checksum may not guarantee all possible corruptions to be found.

iii. *Omission failures*: messages (i.e. UDP datagrams) may not arrive at their destinations, validity is not met.

iv. *Ordering*: as individual datagrams may be delayed, they could arrive out of sender order. Since in general UDP does not do automatic retransmission, duplicate datagrams will not appear.

UDP applications that require reliable communications must provide their own acknowledgments, timeouts, and sequence numbers.

e. TCP stream communications.

(a) Characteristics

i. TCP is a connection-oriented transport layer comm. protocol. It provides reliable message delivery service.

ii. *Message sizes*: applications can decide the number of bytes to write to a stream or read from it. The underlying implementation maintains local write buffers and read buffers. The buffer sizes determines how much data to collect from a application before actually transmitting it, or how much data to accumulate from an incoming stream before delivering to an application. Applications can force data to be transmitted immediately.

https://youtu.be/LnbvhoxHn8M

https://en.wikipedia.org/wiki/Sliding_window_protocol

iii. *Lost messages*: TCP employs the well-known *slide window* protocol to handle message transmissions. Messages are acknowledged and lost messages are retransmitted. In terms of slide window protocol concept, each byte is counted as a TCP packet. For example, when a client declares to a server window size $8k$, that means the server can send $8k$ bytes continuously before having to wait for acks from the client.

数据包

什么意思?

iv. *Flow control*: TCP attempts to *synchronize* the transmission speeds of a client and a server involved in a comm. in the sense that if one of them is slow, TCP will try to inform the other to slow down.

v. *Message duplication and ordering*: use of slide window protocol by TCP ensures that duplicated messages will be detected and discarded. In addition, messages delivering order will be the same as they are sent.

(b) Major TCP API functions:

i. *socket, bind, listen, connect, accept, close* – for comm. management.

ii. *read, write, readv, sendv* – for sending and receiving messages.

什么是comm. management

(c) Failure model:

i. TCP employs checksums, timeouts on individual messages, and retransmissions, in attempt to provide reliable communications.

ii. TCP also uses an overall timeout for a connection. If TCP cannot communicate with its peer for a prolonged period, it will assume that connection is broken. Hence, TCP *does not* provide reliable communications under our failure model, because it cannot guarantee delivery of messages in the faces of all possible problems. 为什么?

iii. When a TCP connection is broken, the sender process does not know if its messages have been delivered; and both the sender and receiver processes do not know whether disconnection is caused by a network failure or failure of its peer.

(3) Java UDP and TCP API

a. The package that provides TCP/IP API in Java is *java.net*.

(a) Java TCP/IP API (called networking API in java) is implemented using sockets. It can be regarded as on a little higher level than the original socket API. For instance, programmers do not have to know the socket structures and applications do not have to explicitly initialize individual fields of such structures.

(b) For both UDP and TCP, Java TCP/IP API provides a sequence of methods corresponding to those in original socket API.

b. Java UDP API

(a) A Java UDP datagram packet: consisting four fields (p.151):

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

(b) An Java UDP example (Fig.4.3, p.152, Fig.4.4, p.153): a client sends a char string to a server, which sends it back as reply. The client is invoked with syntax

java UDPClient char_string server_IP_name

(c) The UDP client code (Fig.4.3, p.152)

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

(d) Major Java UDP API classes and methods:

i. Class *DatagramSocket*: represents a datagram socket. There are three constructors for this class:

(i) *DatagramSocket()*;

(ii) *DatagramSocket(int port)*;

(iii) *DatagramSocket(int port, InetAddress laddr)*.

ii. Major methods in class *DatagramSocket*:

(i) *void close()* closes the datagram socket.

(ii) *void send(DatagramPacket p)*: sends a datagram packet through this socket.

(iii) *void receive(DatagramPacket p)*: receives a datagram packet from this socket;

(iv) *void setSoTimeout(int timeout p)*: enable/disable SO_TIMEOUT with the specified timeout in units of milliseconds. A zero value means infinite timeout (disabled). When a timeout occurs, a *java.io.InterruptedIOException* is thrown;

iii. class *DatagramPacket*: represents a datagram packet. There are four constructors for this class:

(i) *DatagramPacket(byte[] buf, int length)*: constructs a DatagramPacket for receiving packets of length *length*.

(ii) *DatagramPacket(byte[] buf, int length, InetAddress address, int port)*: constructs a DatagramPacket for sending to the specified port number on the specified host.

(iii) *DatagramPacket(byte[] buf, int offset, int length)*: constructs a DatagramPacket for receiving packets of length *length*, specifying an offset into the buffer.

这句话什么意思? specifying an offset into the buffer

(iv) *DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)*: this method combines method 2 and 3 above.

iv. Major methods in class *DatagramPacket*:

(i) *byte[] getData()*: returns the data in bytes from the packet.

(ii) *int getLength()*: returns the length of the data to be sent or length of data received.

The data and length fields are illustrated by the structure of an Java UDP packet on p.151.

(e) Other components in the class.

i. The *getByName* method in class *InetAddress* takes an IPv4 address or name as argument and return the IPv4 address as return value (a NULL argument will return the local computer's IP address).

ii. A *try* block encloses a section of code that will be monitored for exceptions that the corresponding *catch* clauses will try to catch.

iii. An *IOException* object is thrown by both the client and server programs when there is an error caused by failed or interrupted I/O operations.

iv. Similarly a *SocketException* object is thrown when there is any error in the underlying protocol of the corresponding socket. *IOException* class is a superclass of *SocketException* class. Therefore if a program does not catch a SocketException, an IOException will be raised.

(f) The UDP server code (Fig.4.4,p.153)

```
import java.net.*;
import java.io.*;
public class UDPServer{
   public static void main(String args[]){
      DatagramSocket aSocket = null;
      try{
         aSocket = new DatagramSocket(6789);
         // create socket at agreed port
         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();}
  }
}
```

c. Java TCP API

(a) Unlike Java UDP, Java TCP will not create datagram packets for sending or preparing a datagram packet buffer for receiving. Java TCP creates *DataInputStream* and *DataOutputStream* for receiving and sending stream of bytes.

(b) An Java TCP example (Fig.4.5, p.156, Fig.4.6, p.157):

i. The client

(i) It is invoked through a command line like:

java TCPClient char_string server_IP_name

(ii) The client creates a *communication socket* by calling a constructor of the *Socket* class.

(iii) The client opens a *DataInputStream* for reading and a *DataOutputStream* for writing.

(iv) The client sends the first command-line parameter to the server by calling the *writeUTF* method of the DataOutputStream. It attempts

to receive reply from the server by calling the *readUTF* method of the DataInputStream.

ii. The server

(i) The server calls *ServerSocket* class to create a server listen socket instance. It then enters *listen* mode and calls the *accept* method of the listen socket.

(ii) When a new connection request arrives, the server creates a new thread by calling the application defined *Connection* class, passing a client socket obtained from return of the accept method.

(iii) The constructor of the Connection class creates two data streams using the socket passed by the main process. It then calls *this.start()* method to put the thread in the *ready* state (when a thread is created, it is initially in *born* state).

(iv) The thread just reads a line of input from the client and sends it back to the client. It then terminates by calling the *close* method of the sockets it back to the client. It then terminates by calling the *close* method of the socket.

(c) The TCP client code (Fig.4.5, p.156)

```
import java.net.*;
import java.io.*;
public class TCPClient {
   public static void main (String args[]) {
   // arguments supply message and hostname
      Socket s = null;
      try{
         int serverPort = 7896;
         s = new Socket(args[1], serverPort);
         DataInputStream in = new
                     DataInputStream(s.getInputStream());
         DataOutputStream out = new
                     DataOutputStream(s.getOutputStream());
         out.writeUTF(args[0]);  // UTF is a string encoding (cf.4.3)
         String data = in.readUTF();
         System.out.println("Received: "+ data);
      }catch (UnknownHostException e){
                System.out.println("Socket:"+e.getMessage());
      }catch (EOFException e)
                {System.out.println("EOF:"+e.getMessage());
      }catch (IOException e)
                {System.out.println("readline:"+e.getMessage());}
      } finally {if (s!=null) try {
```

```
            s.close();} catch (IOException e){/*close failed */}}
   }
```

(d) The TCP server code (Fig.4.6, p.157)

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

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(); // read a line of data from the stream
            out.writeUTF(data);
         }catch (EOFException e)
                   {System.out.println("EOF:"+e.getMessage());
         }catch (IOException e)
                   {System.out.println("IO:"+e.getMessage());}
         } finally { try { clientSocket.close();}
                         catch (IOException e) {/* close failed */}}
      }
```

```
}
```

3. External data representation and marshalling
https://en.wikipedia.org/wiki/Marshalling_(computer_science)

(1) The need of external data representation

a. The need of EDR

(a) Different programming languages support different primitive data types and data structures. Even for the same data type, such as floating point numbers, two different languages on two different platform may implement them differently.

(b) The set of codes used to present characters can be different according to platforms. UNIX systems use ASCII coding and take one byte per character. UTF-8 (universal transfer format) code takes two bytes per character. For integers, there are also the so called *big-endian* (most significant byte first) verse *little-endian* issue.

(c) These different primitive data types and data structures, plus potential different set of codes, have to be communicated through messages.

b. Two approaches to handle the above diversities

(a) The values are converted to an agreed external format before transmission and converted to the local format upon receipt.

(b) The values are transmitted in the sender's format together with an indication of the format used, and the recipient converts the values if necessary.

The first approach is commonly used.

c. *External data representation*: an agreed standard for the representation of primitive data types and other data structures.

d. *Marshalling* and *unmarshalling*:

(a) *Marshalling* is the action of taking a collection of data items to be used in a communication and transform them into items in a form suitable for transmission in a message.

(b) *Unmarshalling* is the opposite action of *marshalling*. It is the action that takes a received message, extract individual data items from the message, and produce equivalent data items that can be used by the receiving process.

Clearly marshalling and unmarshalling are based on EDR.

e. EDRs and marshalling discussed here:

(a) SUN RPC XDR

(b) DCE RPC interface

(c) CORBA's common data representation

(d) JAVA RMI

(e) JAVA IDL

(2) SUN RPC

a. SUN RPC's XDR (External Data Representation) provides a uniform type conversion mechanism. SUN 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.

b. *Serializing and deserializing*: In SUN RPC's terms, the action of converting from a particular machine representation to XDR format is called *serializing*. The reverse action is called *deserializing*.

c. SUN RPC provides three level interfaces to applications. XDR is used at the intermediate and low levels.

d. *Marshalling* (i.e. *serializing*) in SUN RPC:

(a) Each data type that is allowed to be transmitted in an RPC call must have a corresponding function (called an *XDR filter*) defined for it. The XDR filter handles the serializing and deserializing of that data type.

(b) XDR filters are written in XDR language, which is similar to C language in syntax and semantics. The XDR compiler *rpcgen* compiles an XDR file (each ends with suffix *xdr*) into C files, which can be compiled and linked with clients and servers.

(c) XDR filters can be built in a structured fashion. SUN RPC provides 11 built-in type XDR filters: xdr_int(), xdr_u_int(), xdr_long(), ... Users can define an XDR filter based on these built-in XDR filters or any existing XDR filters.

e. Example: For an application defined structure:

```
struct simple {
   int a;
   short b;
} simple;
```

then you would call callrpc() as

callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);

where xdr_simple() is written as:

```
#include <rpc/rpc.h>
```

```
xdr_simple(xdrsp, simplep)

XDR *xdrsp; struct simple *simplep; {

   if (!xdr_int(xdrsp, &simplep->a))
      return (0);
   if (!xdr_short(xdrsp, &simplep->b))
      return (0);
   return (1);
}
```

(3) DCE RPC

https://baike.baidu.com/item/外部数据表示法

a. DCE RPC applications must first define their interfaces which are written in the IDL (interface definition language) language. An IDL file ends with suffix .idl and has syntax similar to the C language.

b. An IDL interface definition defines all data types that can be used by both client and server and all remote functions/procedures available as services. Each remote function/procedure declared in the interface contains complete specification of their arguments, their orderings, and return values.

c. All servers that support the interface must implement the remote function/procedures using the same function/procedure names, data types, arguments, and ordering of arguments. All clients must call a remote function/procedure consistent with its declaration in the interface.

d. *Marshalling* in DCE RPC:

   (a) Compilation of an interface file produces a C header file, a client stub file in C, and a server stub file in C. The C header file produced contains for each remote function/procedure declared in the IDL file its corresponding C declaration.

   (b) The client stub file performs all marshalling/unmarshalling actions for the client. It should be compiled and linked with other client files (such as the file that contains the client's main function) to produce an executable client program.

   (c) Similar to the server stub file.

   The DCE IDL compiler is supported by the IDL library in producing client and server stubs. Thus IDL compiler provides transparency in EDR and EDR marshalling. 什么意思??

(4) CORBA's common data representation (CDR)

a. CORBA CDR is an EDR.

(a) It defines 15 primitive data types (such as *short* (16-bit), *long* (32-b), *unsigned short*, *unsigned long*, *float* (32-b), *double* (64-b), *char*, *boolean*, *octet* (8-b), and *any* which can represent any basic or constructed data type.

(b) It also defines 6 composite data types (Fig.4.7, p.160) that are commonly used in programming languages:

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

Figure 4.7 CORBA CDR for constructed types

(c) For primitive data types, CDR supports both big-endian and little-endian orderings. The specific ordering of each value is attached in the containing message.

(d) A value of the composite type will be represented as a sequence of bytes according to the ordering as specified by Fig.4.7, and the ordering of its underline primitive data types.

b. *Marshalling* in CDR.

(1) Like in DCE RPC, CORBA supports its own IDL. Interfaces written in CORBA IDL can be compiled by its IDL compiler to produce client and server stubs that will perform marshalling and unmarshalling automatically with no need of interferences from users.

(2) For a structure defined by

```
struct Person {
    string name;
    string place;
    long year;
};
```

CORBA IDL will flatten a Person struct with value: {'Smith', 'London', 1984} to a sequence of $4 \times 7 = 28$ bytes (Fig.4.8, p.161):

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	1984	*unsigned long*

Figure 4.8 CORBA CDR message

(5) Java RMI (remote method invocation, since JDK1.1)

a. Basics

(a) Java RMI is Java's implementation of RPC for Java-object-to-Java-object distributed communications.

(b) A method of a Java object can be registered through RMI and will become remotely accessible by other Java methods thereafter.

i. The syntax of invoking a remote method is identical to that of invoking a method in the same program. 在说什么？？？？

ii. Parameter marshalling/unmarshalling is done by RMI.

(c) The *java.rmi* package contains classes that support RMI.

b. *Marshalling* in Java RMI

(a) Marshalling in RMI is based on Java's concept of serialization and *deserialization*.

i. *Serialization* refers to the activity of *flattening* an object into certain form of serial bytes so that it can be transmitted as an argument or result of RMI invocation.

ii. As the recipient of a serialized object does not know the original object, additional information has to be attached in the serialized form of an object to help the deserialization operation.

iii. For a class, the information is the name of the class and its version number. The version number can be supplied by a programmer, or can be calculated through class name hash.

iv. Any object referenced by an object will be serialized when the latter is serialized. Referenced objects are serialized as *handles*, which is a reference to an object within the serialized form.

(b) Example. For a structure

Person p = new Person("Smith", "London", 1984)

The corresponding serialized form is shown in Fig.4.9, p.163.

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*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h_0 and h_1 are handles

Figure 4.9 Indication of Java serialized form

(c) Serialization and deserialization actions on arguments and results of remote invocations are performed by Java RMI support (which is part of the middleware).

(d) Programmers may write special methods to fulfill special serialization requirements if desired.

i. The *writeObject* method in an instance of the *ObjectOutputStream* class can be invoked to serialize an object (such as the *Person* structure) that is passed to it.

ii. Similarly, the *readObject* method in an instance of the *ObjectInputStream* class can be invoked to deserialize (restore) an object.

(6) XML (Extensible markup language)

a. XML vs HTML

(a) HTML: hypertext markup language. It is a tagged language – elements in it are tagged by an open tag and a corresponding end tag. The elements in HTML contain infromation as how a web browser should display the text contents.

(b) XML is also a tagged language. However, XML elements contain information both for displaying the data and describing the logical structure of the data.

(c) Both XML and HTML belongs to class of "markup" languages that are defined by applying SGML (standard Generalized Markup Language). Each application of SGML is a markup language.

b. Main usage of XML:

   (a) XML supercedes HTML as a web markup language. It can be used for web services to define their interfaces and other properties.

   (b) XML can be used for clients to communicate with web servers.

   (c) XML can be used for information archival and retrieval. Portable across platforms.

   (d) XML can be used for interface and services specifications of any applications.

c. Extensible XML

   (a) XML can define its own tags. HTML cannot.

   (b) SOAP (Simple Object Access Protocol) is a protocol for foundation layer of a web services protocol stack. XML is the standard language for SOAP. Tags for SOAP are published for use by web clients and servers.

d. Self-describing languages: those that can describe their own features such the meaning of a field.

   (a) CORBA CDR is an example that is not self-describing. Prior knowledge or convention is necessary for the meaning of each field of Fig.4.8 example.

   (b) XML is a self-describing language – it's intended to be used in many applications. Tags and XML namespace make self-describing possible. Tags can also allow selective processing.

e. XML documents are readable by both humans and computer software.

f. XML elements and attributes: Fig.4.10, p.165 shows the XML definition for the *Person* structure used in CORBA CDR and Java RMI.

```
<person id="123456789"/>
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!- -a comment- ->
```

g. Parsing and well-formedness of XML documents

   (a) Well-formedness: the tags are paired and properly nested: they form a string in a context-free language.

   (b) Special characters or data.
   Example: the "<" character can be expressed as "&lt".

   (c) CDATA: special data section that should not be parsed by XML processor.

   (d) For a string: *King's Cross*:
   method 1: <*place*>*King&apos Cross*</*place*>
   method 2: <*place*><![CDATA[*King's Cross*]]></*place*>

h. XML namespaces: Extending the URL to within XML

(a) The *xmlns* attribute, with value of URL that refers to the file containing the name space definitions:
xmlns:pers = "http://www.cdk5.net/person"

(b) Then the *person* structure can be changed to:

<person id="123456789" xmlns:pers="http://www.cdk5.net/person">
<name>Smith</name>
<place>London</place>
<year>1984</year>
</person>

The name *pers* after *xmlns* can be used as prefix to refer to the elements in particular namespace.

i. XML schemas

(a) XML schemas like data types. An XML schema defines the names of the names and attributes of a XML definition, together with the nested structure of the elements within.

(b) A single XML schema can be shared by many differebt (字写错了...) documents. XML documents that are defined to conform to a particular schema can be verified.

(c) Example: Fig.4.12: An XML schema for the *Person* structure.

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions  >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
               <xsd:element name = "name"  type="xs:string"/>
               <xsd:element name = "place"  type="xs:string"/>
               <xsd:element name = "year"  type="xs:positiveInteger"/>
              </xsd:sequence>
        <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

Fig.4.12 An XML schema for the Person structure

j. API for processing XML: available software written in Java, Python, and other popular languages. As different object languages different class/object differently, users should choose the software according to their needs.

(7) Java IDL (since JDK1.2)

a. Java IDL is intended for Java applications and applets to communicate with objects written in any languages that support CORBA. RMI on the other hand is restricted to Java-to-Java communications.

b. Like DCE IDL, Java IDL is an interface definition language. Java IDL supports a compiler (idlj on UNIX, idlj.exe on Windows) to compile Java IDL files into Java files. The use of a standard interface definition language provides transparency to marshalling/unmarshalling operations. 什么意思?

为何要compile?
黑人问号????

c. *Marshalling* in Java IDL applications. Similar to DCE IDL, the first step of writing a Java IDL application is developing a Java IDL interface. For an interface named *Xyz* and an IDL file named *Xyz.idl*, the IDL compiler idlj.exe will compile it and produce the following files:

(a) A *server skeleton* file _XyzImplBase.java , which provides basic CORBA functionality for the server. It implements the interface. This is the server stub file.

(b) A *client stub* file _XyzStub.java, which provides basic CORBA functionality for the client.

(c) A file Xyz.java which is the Java version of the IDL interface.

(d) A file XyzHelper.java which provides auxiliary functionality required to cast CORBA object references to their proper types.

(e) XyzHolder.java which holds a public instance member of type Xyz/. It provides operations for *out* and *inout* arguments, which CORBA has but which do not map easily to Java's semantics.

A sample Java IDL file *Hello.idl*:

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

4. Remote object references

(1) In RMI, there is a need of unique way to identify a particular remote object that implements the invoked method. 什么意思?

(2) Remote object reference is an identifier that is unique in the whole distributed/cloud environment and refers to unique remote object.

(3) Such a reference hence has to include the IP number of the computing hosting the remote object, a port number that refers a particular service. The use of IP number also allows relocation of remote objects (at the expense of location transparency). 这是什么意思?

(4) In addition, the following should be part of a remote object reference as well (Fig.4.13, p.169):

看不懂～

a. Time: the time when the remote object created;

b. Local object number: the remote object is a local object on the hosting computer;

c. interface of remote object.

* This could be interface name
* The information is useful to any process that receives a remote object reference as an argument, or as the result of a remote invocation, as the process needs to know the method offered by the remote object (Cf.Sect.5.4.2).

5. Group communications

(1) Multicasting: sending or receiving messages to or from a targeted group of processes. Major issues in group communications:

a. Group membership managements.

b. Message delivery mechanisms.

c. Reliability and validity issues.

d. APIs.

(2) Example applications of multicast protocols. Group communication is another important way of message exchanges in DSs.

a. *Fault tolerance based on replicated services*: A service may be replicated and provided by multiple servers in a DS. A client may send a request (such as modify a data file) to all servers. Reply from any of them will satisfy the client. As long as one of the servers is still available, the service will be available to all clients.

b. *Better performance through replicated data*: for a set of replicated data, multicast is used to send a update request to all processes that manage a copy of the replicated data.

c. *Finding the discovery servers in spontaneous networking*: a server of a RPC service or a client may locate available discovery services in order to register its interface or look up interfaces of certain services in a spontaneous networking environment.

这里在做什么?

d. *Propagation of event notifications*: certain events such as unavailability or reavailability of a comm. link can be sent to a group of routers through multicasting.

(3) IP multicast

a. IP multicast is based on the support of IP.

   (a) IP normally supports *unicast* communications (from one computer to another computer). However, in IPv4 an application can specify a *class D* IPv4 address (an address with prefix 1110) in an IP packet to indicate that the packet should be delivered to a group of computers.

   (b) Each class D IPv4 address can be used to represent a *multicast group*.

b. Multicast API.

   (a) The API to programmers is UDP. A process can perform the following steps to let the hosting computer become a member of a multicast group (Note: multicast membership is with respect to computers, not individual processes on a computer):

      i. Obtain a UDP *socket* descriptor and *bind* the descriptor to a *port number*.

      ii. Uses *setsockopt* function to set the *IP_ADD_MEMBERSHIP* option with the value of a class D IPv4 address.

   If successful, after above operations the hosting computer becomes a member of the multicast group represented by the class D IPv4 address. The computer will be able to send multicast packets to and receive multicast packets from other members in the group.

   (b) A process that initiated the membership of a multicast group can ask the hosting computer to leave a multicast group by calling *setsockopt* function to set the *IP_DROP_MEMBERSHIP* option.

   (c) Multiple processes on a computer can initiate membership joining operations. The host will remain a member of a specific multicast group as long as at least one process who initiated a membership joining operation before hasn't initiated the corresponding membership dropping operation.

   (d) Each multicast packet sent by a multicast process will also be copied to the hosting computer itself as an multicast packet input, although this can be explicitly disabled.

   (e) A process can send a multicast packet to a multicast group even if it hasn't initiated an appropriate membership joining operation. However, such a process is not able to receive multicast packets to that multicast group.

c. Multicast on LAN and WAN. Members of a multicast group can be anywhere on a WAN.

   (a) Management of multicast groups is done through *multicast routing protocols*.

   (b) When a computer joins a multicast group, the computer sends a special control message (a IGMP message) to some attached multicast routers, which in turn will exchange this info with other multicast routers about the change.

(c) Similar actions will be taken when a computer leaves a multicast group.

(d) Informally, the multicast routers for a specific multicast group form a tree. This tree can grow when new members join and be pruned when some member leaves.

(e) On the datalink level (such as Ethernet), when a computer joins a multicast group, the datalink interface of that computer will be instructed to receive datalink packets destined to a special datalink address. For Ethernet, that address is 01:00:5e:00:01:01, which is the Ethernet multicast address. Ethernet packets with this address will be picked up by the interface. The IP layer will then examine the multicast IP address inside.

这段没看懂?

d. Failure model of IP multicasting.

(a) As IP multicasting uses UDP and is based on IP, it is *unreliable*.

i. Any multicast packet may not be guaranteed to be delivered to all members of the multicast group.

ii. Multicast packets may be delivered not in FOFI order.

(b) IP multicast is a basic multicast. Reliable multicast can be implemented by enhancing a basic multicast (Ch.12).

(4) Consequences of IP multicast failure model on different applications.

a. *Fault tolerance based on replicated services* and *Better performance through replicated data*: The applicability of IP multicast depends on the nature of the replicated data and replicated services. Newsgroups data files and services do not have to be consistent all the time. A user may succeed in posting an article to one news server while failing to post the same article to another news server. But that is acceptable in most cases. On the other hand, a banking account file and service have to be consistent.

b. *Finding the discovery servers in spontaneous networking*: IP multicast is still usable here.

c. *Propagation of event notifications*: This is again application dependent. Failing to notify certain routers about the current condition a comm. link is acceptable, while failing to notify a consensus to some processes in a distributed consensus algorithm (Ch.11) is not acceptable.

什么意思?

(5) Example application of Java IP multicast API (Fig. 4.20, p.166)

a. After compilation, the program can be run by the command:

java MulticastPeer char_string multicast_IP_number

where *char_string* is a char string to be multicast to group members. Note: Java program takes the first command-line parameter as *args[0]*, the second command-line parmeter as *args[1]*, and so on. Different from C/C++.

b. The constructor of the *MulticastSocket* class creates a multicast socket (a UDP socket) and binds the socket to port 6789.

c. The program joins the IPv4 multicast group with group IPv4 number 228.5.6.7 through the *joinGroup* method of object *s* of class *MulticastSocket*.

d. The After joining the group, it multicasts a message (a char string, which is the first command-line argument) and then iterates in a *for* loop three times to receive three multicast messages.

e The process then leaves the group by invoking the *leaveGroup* method of object *s*.

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
   public static void main(String args[]){
   // args give message contents and destination multicast
   //               group (e.g. "228.5.6.7")
   MulticastSocket s = null;
   try {
      InetAddress group = InetAddress.getByName(args[1]);
      s = new MulticastSocket(6789);
      s.joinGroup(group);
      byte [] m = args[0].getBytes();
      DatagramPacket messageOut = new DatagramPacket(m,m.length,group,6789);
      s.send(messageOut);
      // get messages from others in group
      byte[] buffer = new byte[1000];
      for(int i=0; i< 3;i++) {
         DatagramPacket messageIn = new DatagramPacket(buffer,buffer.length);
         s.receive(messageIn);
         System.out.println("Received:"+ new String(messageIn.getData()));
      }
      s.leaveGroup(group);
   }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
   }catch (IOException e){System.out.println("IO: " + e.getMessage());}
   }finally { if (s!=null) s.close();}
}
```

6. Network virtualization: overlay networks

(1) The need of network virtualization

a. Internet comm. protocols have to support all different types of applications

b. Different applications desire different preferred support of comm. protocols

c. It's impossible to alter comm. protocols to support all these different protocols

(2) Network virtualization: contruction of various *virtual networks* that are supported by the underlying existing networks (such as Internet).

a. Each virtual network is designed with the goal of supporting a particular distributed/cloud application or classes of such applications. Examples:

* Multimedia streaming virtual network that supports applications such as BBC iPlayer, Boxee TV (boxee.tv), or Hulu (hulu.com).

b. Each virtual network has its own addressing schemes, protocols, and routing schemes. They are tailored to facilitate the targeted applications

Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].

Figure 29: Type of overlay (Part 1, Fig.4.15, p.176)

(3) Overlay networks (section 4.5.1)

a. An overlay network is a virtual network consisting of nodes and virtual links. Such a network sits on top of an underlying network (such as IP network) and offers features that are not provided by the underlying network (Fig.4.15, p.176):

* A service that is tailored towards the need of class of applications or a particular higher-level service. Examples: multimedia content distribution, stricter authentication.
* More efficient operation in a given networked environment, such as routing in an ad hoc mobile network

Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

Figure 30: Type of overlay (Part 2, Fig.4.15, p.176)

- Other additional features such as more efficient multicast or secure communication.

b. Main advantages of overlay networks
   (a) Providing additional services without need of changing underlying network protocols, which is difficult to change due to the need of being generic and standardization.
   (b) Facilitating experiments and deployment of new network services and applications and customization of services to particular classes of applications.
   (c) Providing a more open and extensible network architeture. Multiple overlays (both vertical and horizontal) can be defined and can coexist.

c. Problems of overlay networks
   (a) Extra level of indirection and added complexity of network services.
   (b) Extra overhead

d. Network layers vs overlay networks
   (a) Existing network layers are generic and covers the core of standard parts of networks.
   (b) Overlays are on top of existing network layers (outside the standard network layer architecture. Hence they can exploit the resultant degree of freedom.
   (c) Overlay designers and developers have freedom of proposing core parts of a network as desired.

(d) Example: Distributed hash tables introduce a new style of addressing based on a keyspace. It results in a network topology in such a way that each node in such a network either owns the key or connects through a link to a node that is closer to the owner. This often leads to ring topology suitable and efficient for certain applications.

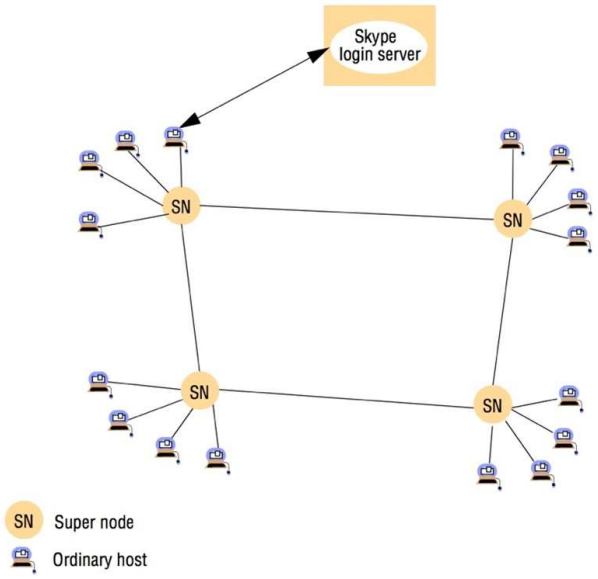

Figure 31: Skype overlay architecture (Fig.4.16, p.177)

(4) Skype: example of an overlay network

a. A peer-to-peer application offers VoIP (voice over IP) service, 2003. Other services:

(a) Instant messaging

(b) Video conferencing

(c) Interfacing to standard telephony service (through features of SkypeIn and SkypeOut)

370 million users at end of 2009.

b. Architecture and operation: Fig.4.16, p.177

(a) Peer-to-peer application

(b) SN (super node). Super nodes are ordinary hosts, selected to carry out enhanced activities/duties. Selection is based on multiple parameters such as bandwidth availability, reachability, and availability. Other nodes are ordinary hosts (pure clients).

(c) User connection: user login is authenticated through a well-known login server, in cooperation with a seleted super node. Each client maintains a cache of super node identifiers (IP address and port numbers) and the cache is updated periodically.

(d) Search for users: this is another extremely critical function. This is done through cooperation of super nodes. Normally on average eight super nodes are involved to complete a search.

(e) Voice connection: after a user is discovered, TCP is used to signalling call requests and termination. Voice signals are carried using either TCP or UDP. UDP is preferred (due to flexibility and voice communication nature).

7. Message passing example: MPI

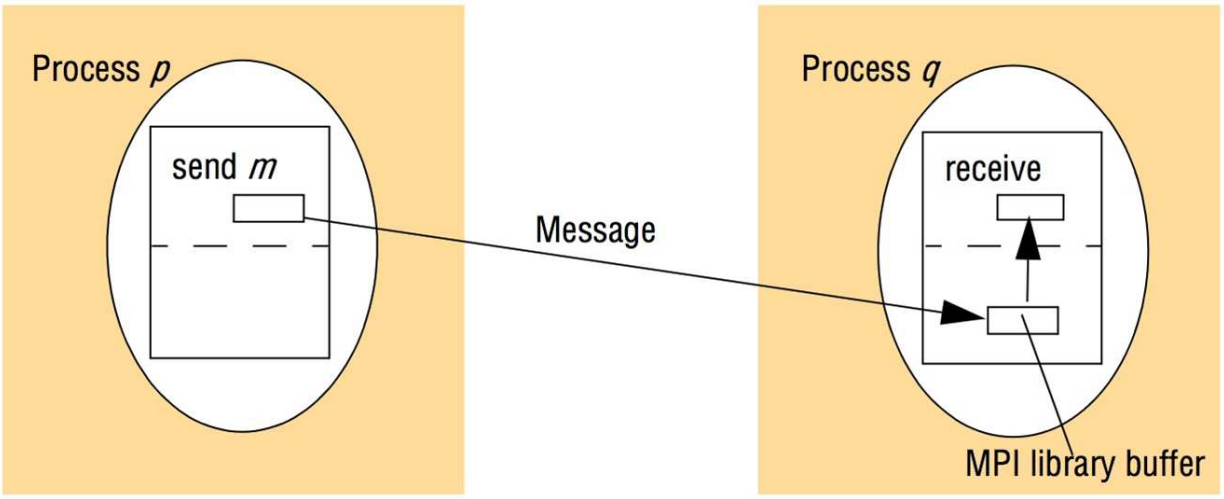

Figure 32: An overview of point-to-point communications in MPI (Fig.4.17, p.179)

(1) MPI (Message Passing Interface, 1994): an effort to provide uniform, efficient message passing mechanism for high performance computing, independent of proprietary implementations of various high performance programming software/language (such as C/C++, Fortran).

(2) Overview of MPI message passing: Fig.4.17, p.179.

   a. A send message m is sent and copied to the MPI library buffer at receiver's end.

   b. The receiver can execute a corresponding receive operation to copy the message.

(3) Selected send operations in MPI: Fig.4.18, p.180

   a. *Generic*: *MPI_Send*

b. *Synchronous*: *MPI_Ssend*

c. *Buffered*: *MPI_Bsend*

d. *Ready*: *MPI_Rsend*

Send operations	*Blocking*	*Non-blocking*
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via *MPI_Wait* or *MPI_Test.*
Synchronous	*MPI_Ssend*: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend*: as with *MPI_Isend*, but with *MPI_Wait* and *MPI_Test* indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend*: the sender explicitly allocates an MPI buffer library (using a separate *MPI_Buffer_attach* call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend*: as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend*: the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend*: the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

Figure 33: Selected send operations in MPI (Fig.4.18, p.180)