Chapter 4: Interprocess Communication

• Introduction
• The API for the Internet protocols
• External data representation and marshalling
• Client-Server communication
• Group communication
• Case study: interprocess communication in UNIX
• Summary
The characteristics of interprocess communication

- Synchronous and asynchronous
  - a queue associated with message destination,
    Sending process add message to remote queue,
    Receiving process remove message from local queue
  - Synchronous: send and receive are blocking operations
  - asynchronous: send is unblocking, receive could be blocking or unblocking (receive notification by polling or interrupt)

The characteristics of interprocess communication

- 一般的处理方法如下：
- 将需要等待的部分写在底层的类里面：
  1）对于send，底层向上提供调用的send接口，上层调用了send后立即返回，实际消息在底层类的发送对立队列中，有底层类中的专门线程负责逐个发送消息。
  2）对receive，可以使用windows的消息机制，如果单纯使用C++实现，则做成一个观察者模式，上层模块把消息的处理类注册到底层类中，消息到来后，由底层类调用高层类的相应处理消息的方法。
The characteristics of interprocess communication

- Message destination
  - Internet address + local port
  - service name: help by name service at run time
  - location independent identifiers, e.g. in Mach
- Reliability
  - validity: messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost
  - Integrity: messages arrive uncorrupted and without duplication
- Ordering
  - the messages be delivered in *sender order*
Socket

- Endpoint for communication between processes
- Both forms of communication (UDP and TCP) use the socket abstraction
- Originate from BSD Unix, be present in most versions of UNIX
- Be bound to a local port (2^16 possible port number) and one of the Internet address
- A process cannot share ports with other processes on the same computer

UDP datagram communication

- UDP datagrams are sent without acknowledgement or retries
- Issues relating to datagram communication
  - Message size: not bigger than 64k in size, otherwise truncated on arrival
  - Blocking: non-blocking sends (message could be discarded at destination if there is not a socket bound to the port) and blocking receives (could be timeout)
  - Timeout: receiver set on socket
  - Receive from any: not specify an origin for messages
- Failure model
  - Omission failure: message be dropped due to checksum error or no buffer space at sender side or receiver side
  - Ordering: message be delivered out of sender order
  - Application maintains the reliability of UDP communication channel by itself
Java API for UDP datagrams

- **DatagramPacket**
- **DatagramSocket**
  - `send` and `receive`: transmit datagram between a pair of sockets
  - `setSoTimeout`: receive method will block for the time specified and then throw an `InterruptedIOException`
  - `connect`: connect to a particular remote port and Internet address

- **Examples**
  - be acceptable to services that are liable to occasional omission failures, e.g. DNS

TCP stream communication

- **The API to the TCP**
  - provide the abstraction of a stream of bytes to which data may be written and from which data may be read
- **Hidden network characteristics**
  - message sizes
  - lost messages
  - flow control
  - message duplication and ordering
  - message destinations
- **issues related to stream communication**
  - *Matching of data items*: agree to the contents of the transmitted data
  - *Blocking*: send blocked until the data is written in the receiver’s buffer, receive blocked until the data in the local buffer becomes available
  - *Threads*: server create a new thread when it accept a connection
TCP stream communication … continued

• failure model
  – integrity and validity have been achieved by checksum, sequence number, timeout and retransmission in TCP protocol
  – connection could be broken due to unknown failures
    • Can’t distinguish between network failure and the destination process failure
    • Can’t tell whether its recent messages have been received or not

Java API for TCP Streams (skip)

• ServerSocket
  – accept: listen for connect requests from clients
• Socket
  – constructor
    • not only create a socket associated with a local port, but also connect it to the specified remote computer and port number
  – getInputStream
  – getOutputStream
• Examples
Chapter 3: Interprocess Communication

• Introduction
• The API for the Internet protocols
• External data representation and marshalling
• Client-Server communication
• Group communication
• Case study: interprocess communication in UNIX
• Summary

External data representation and marshalling introduction

• Why does the communication data need external data representation and marshalling?
  – Different data format on different computers, e.g.,
    big-endian/little-endian integer order, ASCII (Unix) / Unicode character coding

• How to enable any two computers to exchange data values?
  – The values be converted to an agreed external format before transmission and converted to the local form on receipt
  – The values are transmitted in the sender’s format, together with an indication of the format used, and the receipt converts the value if necessary
External data representation and marshalling introduction

- **External data representation**
  - An agreed standard for the representation of data structures and primitive values

- **Marshalling (unmarshalling)**
  - The process of taking a collection of data items and assembling them into a form suitable for transmission in a message
  - Usage: for data transmission or storing in files

- **Two alternative approaches**
  - CORBA’s common data representation / Java’s object serialization

### CORBA’s Common Data Representation (CDR)

- **Represent all of the data types that can be used as arguments and return values in remote invocations in CORBA**

- **15 primitive types**
  - Short (16bit), long(32bit), unsigned short, unsigned long, float, char, …

- **Constructed types**
  - Types that composed by several primitive types

- **A message example**

- **The type of a data item is not given with the data representation in message**
  - It is assumed that the sender and recipient have common knowledge of the order and types of the data items in a message.
  - For RMI and RPC, each method invocation passes arguments of particular types, and the result is a value of a particular type.
### CORBA CDR for constructed types

<table>
<thead>
<tr>
<th>type</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Length(unsigned) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>Length(unsigned long) followed by characters in order</td>
</tr>
<tr>
<td>array</td>
<td>Array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>In the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>Unsigned long (the value are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>Type tag followed by the selected member</td>
</tr>
</tbody>
</table>

### CORBA CDR message

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

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>4 bytes</th>
<th>notes on representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>5</td>
<td>length of string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘Smith’</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td>length of string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘London’</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h___&quot;</td>
<td>unsigned long</td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td></td>
</tr>
<tr>
<td>20–23</td>
<td>&quot;on___&quot;</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
<td></td>
</tr>
</tbody>
</table>

The flattened form represents a `Person` struct with value: `{‘Smith’, ‘London’, 1934}`
Java object serialization

- Serialization (deserialization)
  - The activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on the disk or transmitting in a message
  - Include information about the class of each object and a version number
  - Handles: references to other objects are serialized as handles
    - Each object is written once only
  - Make use of Java serialization
    - ObjectOutputStream.writeObject, ObjectInputStream.readObject

Java object serialization

- The use of reflection
  - *Reflection*: The ability to enquire about the properties of a class, and also enables classes to be created from their properties.
  - Reflection makes it possible to do serialization (deserialization) in a completely generic manner
Indication of Java serialized form

```java
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
}
Person p = new Person("Smith", "London", 1934);
```

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>h0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith</td>
<td>6 London</td>
</tr>
</tbody>
</table>

The true serialized form contains additional type markers; h0 and h1 are handles.

class Couple implements Serializable {
    private Person one;
    private Person two;
    public Couple(Person a, Person b) {
        one = a;
        two = b;
    }
    Couple t1 = new Couple(new Person("Smith", "London", 1934),
                           new Person("Jones", "Paris", 1945));
```
<table>
<thead>
<tr>
<th>Couple</th>
<th>8 byte version number</th>
<th>h0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Person one</td>
<td>Person two</td>
</tr>
<tr>
<td>Person</td>
<td>8 byte version number</td>
<td>h1</td>
</tr>
<tr>
<td>3</td>
<td>int year</td>
<td>java.lang.String name</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith</td>
<td>6 London</td>
</tr>
<tr>
<td>1945</td>
<td>5 Jones</td>
<td>5 Paris</td>
</tr>
</tbody>
</table>

**Remote object reference**

- Representation of a remote reference

<table>
<thead>
<tr>
<th>Internet address</th>
<th>port number</th>
<th>time</th>
<th>object number</th>
<th>interface of remote object</th>
</tr>
</thead>
</table>

- a remote object reference must be unique in the distributed system and over time. It should not be reused after the object is deleted.
- the first two fields locate the object unless migration or re-activation in a new process can happen
- the fourth field identifies the object within the process
- its interface tells the receiver what methods it has (e.g. class *Method*)
- a remote object reference is created by a *remote reference module* when a reference is passed as argument or result to another process
  - it will be passed in request messages to identify the remote object whose method is to be invoked
Chapter 4: Interprocess Communication

- Introduction
- The API for the Internet protocols
- External data representation and marshalling
- Client-Server communication
- Group communication
- Case study: interprocess communication in UNIX
- Summary

The request – reply protocol

- The request-reply protocol
- Overheads associated with the TCP protocol
  - Acknowledgements are redundant since requests are followed by replies
  - Establishing a connection involves two extra pairs of messages in addition to the pair required for a request and a reply
  - Flow control is redundant for the majority of invocations, which pass only small arguments and results
- Request-reply message structure
  - requestID: prevent duplicated request and delayed reply
- Message identifiers
  - A requestID
  - An identifier for the sender process, e.g. its port and Internet address
Request-reply communication

**Client**

- `doOperation`
- `(wait)`
- `(continuation)`

**Server**

- `getRequest`
- `execute`
- `sendReply`

---

`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.

---

Request-reply message structure

| messageType | int (0=Request, 1=Reply) |
| requestID | int |
| objectReference | RemoteObjectRef |
| methodID | int or Method |
| arguments | array of bytes |
The request – reply protocol … continued

• Failure model (for UDP)
  – Omission failure and non-ordering, process crash
  – Timeout
    • doOperation return exception when repeatedly issued
      requests are all timeout
  – Duplicate request messages: filter out duplicates by
    requestID
    • if the server has not yet sent the reply, transmit the
      reply after finishing operation execution
    • If the server has already sent the reply, execute the
      operation again to obtain the result. Note idempotent
      operation, e.g., add an element to a set, and a contrary
      example, append an item to a sequence
  – History: server contains a record of reply messages that
    have been transmitted to avoid re-execution of
    operations

RPC Exchange Protocol

• The request (R) protocol
• The request-reply (RR) protocol
• The request-reply-acknowledge replay (RRA) protocol
• Explain the design choice that are relevant to
  minimizing the amount of reply data held at a
  server.
• Suppose that a server using RRA re-transmits the reply message after a delay and consider the case where the client has sent an acknowledgement which was late or lost. This requires
• (i) the client to recognize duplicate reply messages and send corresponding extra acknowledgements and
• (ii) the server to handle delayed acknowledgments after it has re-transmitted reply messages.

The request – reply protocol … continued

• Implement the request-reply protocol on TCP
  – Costly, but no need for the request-reply protocol to deal with retransmission and filtering
  – Successive requests and replies can use the same stream to reduce connection overhead
Chapter 4: Interprocess Communication

- Introduction
- The API for the Internet protocols
- External data representation and marshalling
- Client-Server communication
- Group communication
- Case study: interprocess communication in UNIX
- Summary

The usage of Multicast

- Fault tolerance based on replicated services
  - Client request are multicast to all the members of the group, each of which performs an identical operation
- Better performance through replicated data
  - Data are replicated to increase the performance of a service, e.g., Web Cache. Each time the data changes, the new value is multicast to the processes managing the replicas
- Finding the discovery servers in spontaneous networking
  - Multicast message can be used by servers and clients to locate available discovery services to register their interfaces or to look up the interfaces of other services
IP Multicast – an implementation of group communication

• A multicast group is specified by a class D Internet address
  – Built on top of IP
  – Available only via UDP
• The membership of a group is dynamic
  – It is possible to send datagram to a multicast group without being a member
• IPv4
  – Multicast routers
    • use the broadcast capability of the local network
    • MTTL - specify the number of routers a multicast message is allowed to pass
  – Multicast address allocation
    • Permanent group – 224.0.0.1 to 224.0.0.255
    • Temporary group – the other addresses, set TTL to a small value
• Failure model: due to UDP, so it is a unreliable multicast

Chapter 4: Interprocess Communication

• Introduction
• The API for the Internet protocols
• External data representation and marshalling
• Client-Server communication
• Group communication
• Case study: interprocess communication in UNIX
• Summary
**UNIX socket**

- **Datagram communication**
  - Datagram *Socket*
  - *Bind*
  - *Sendto*
  - *recvfrom*

- **Stream communication**
  - stream *socket*, *bind*
  - *Accept*
  - *Connect*
  - *Write* and *read*

---

**Sockets used for datagrams**

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
bind(s, ClientAddress)
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM)
bind(s, ServerAddress)
amount = recvfrom(s, buffer, from)
```

*ServerAddress* and *ClientAddress* are socket addresses
Chapter 4: Interprocess Communication

- Introduction
- The API for the Internet protocols
- External data representation and marshalling
- Client-Server communication
- Group communication
- Case study: interprocess communication in UNIX
- Summary
Summary

• Two alternative building blocks
  – Datagram Socket: based on UDP, efficient but suffer from failures
  – Stream Socket: based on TCP, reliable but expensive

• Marshalling
  – CORBA’s CDR and Java serialization

• Request-Reply protocol
  – Base on UDP or TCP

• Multicast
  – IP multicast is a simple multicast protocol

Outline the design of a scheme that uses message retransmissions with IP multicast (or UDP datagram communication) to overcome the problem of dropped messages.

Your scheme should take the following points into account:

1) there may be multiple senders;
2) generally only a small proportion of messages are dropped;
3) unlike the request-reply protocol, recipients may not necessarily send a message within any particular time limit.

Assume that messages that are not dropped arrive in sender ordering.
project1的检验方案：

1. 多个发送方检测：在一个结点启动多个或在多个结点上各启动一个消息发送进程，向一组目标机器（2个即可）同时发送一组消息，检查目标结点是否都接收到所有消息无遗漏；
   - 为了方便起见，这组消息内容可以是：
   - 1) 发送顺序的数字编号，可以是全局统一编号或每个发送方独立编号。检验标准是在接收方屏幕上能打印所有消息编号。如果是独立编号，要求接收方能明确表示该消息来自哪个发送方。
   - 2) 多个可执行文件，将文件内容分段填入消息中依次发送，在接收方接收消息后重新拼接成文件，检验标准是可执行文件能成功执行无误。

2. 消息重传检测：以方案一为例，要求同学为发送方和接收方发送的消息分别设置一组可调的丢失概率，实现时可以不发送某一随机数编号的消息来模拟消息丢失，调节随机数选取的范围和选取频率可以控制消息丢失的概率，这些概率设定值必须能在程序外部调节。检验标准是当消息丢失概率小于一定值时（这个值老师可以设定一个要求），在发送方和接收方的屏幕上清晰地显示消息丢失事件和消息重传收到事件。
   - 提交实现报告
• 同学的经验：
• 这份作业涉及到了多线程程序设计（线程、同步、互斥）、socket网络编程（UDP）、一些常用的数据结构（链表、队列、堆）还有一个简单的广播算法。
HTTP: an example of a request – reply protocol

- Over TCP (manage resources as data, and as program)
- Each client-server interaction consists of the following steps
  - The client requests and the server accepts a connection at the default server port or at a port specified in the URL
  - The client sends a request message to the server
  - The server sends a reply message to the client
  - The connection is closed
- Persistent connection
  - Connections that remain open over a series of request-reply exchanges between client and server
- Marshalling
  - Request and replies are marshalled into messages as ASCII text string
  - Resources are represented as byte sequences and may be compressed

### HTTP request / reply messages

<table>
<thead>
<tr>
<th>method</th>
<th>URL or pathname</th>
<th>HTTP version</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td><a href="http://www.dcs.qmw.ac.uk/index.html">http://www.dcs.qmw.ac.uk/index.html</a></td>
<td>HTTP/1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HTTP version</th>
<th>status code</th>
<th>reason</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
<td>OK</td>
<td></td>
<td>resource data</td>
</tr>
</tbody>
</table>
import java.net.*;

public class UDPClient{
    public static void main(String args[]){
        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();}
    }
}

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();}
    }
}
public class TCPClient {
    public static void main(String[] args) {
        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 see Sn 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data);
            }catch (UnknownHostException e){ System.out.println("Sock:"+e.getMessage());}
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());}
        }catch(IOException e){System.out.println("IO:"+e.getMessage());}
        finally {if(s!=null) try {s.close();}catch (IOException e){System.out.println("close:"+e.getMessage());}}
    }
}

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 :"+e.getMessage());}
    }
}

// this figure continues on the next slide
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("IO:"+e.getMessage());
        } finally {try {clientSocket.close();}catch (IOException e) {}}}
}

The usage of Multicast

- Propagation of event notification
  - Multicast to a group may be used to notify processes when something happens, e.g., the Jini system uses multicast to inform interested clients when new lookup services advertise their existence
- special application
  - video conference, chat room
import java.net.*;
import java.io.*;

public class MulticastPeer{

    public static void main(String args[]){
        // args give message contents & 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();}
    }
}

Multicast peers example… continued

// 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();}