Chap 4. Inter-Process Communication

Road map

- 4.1. Intro
- 4.2. API for the Internet Protocols
- 4.3. External data representation and marshalling
- 4.4. Client-Server Communication
- 4.5. Group communication (self-read)
- 4.6. Case study (self-read)
4.1. Intro

- **Focus:**
  - Characteristics of protocols for communication between processes to model distributed computing architecture
    - Effective means for communicating objects among processes at language level
  - **Java API**
    - Provides both datagram and stream communication primitives/interfaces – building blocks for communication protocols
  - **Representation of objects**
    - Providing a common interface for object references
  - **Protocol construction**
    - Two communication patterns for distributed programming: C-S using RMI/RPC and Group communication using ‘broadcasting’
4.1. Intro

- **In Chapter 3**, we covered Internet transport (TCP/UDP) and network (IP) protocols without emphasizing how they are used at programming level.
- **In Chapter 5**, we cover RMI facilities for accessing remote objects’ methods AND the use of RPC for accessing the procedures in a remote server.
- **Chapter 4** is on how TCP and UDP are used in a program to effect communication via socket (e.g. Java sockets) – the Middle Layers – for object request/reply invocation and parameter marshalling/representation.
4.2. API for internet protocols

- IPC primitives
  - message passing between a pair of processes can be supported by two message communication operations: `send` and `receive`
  - `send (destination, &msg);`      `receive (source, &buf)`
  - Destination and source can be process id or port number (single receiver); Or mailbox (multiple receivers)
    - Typically, (IP, port#) pair

- Use of sockets as API for UDP and TCP implementation – much more specification can be found at `java.net`

Message:  

```
| Header | Body |
```
4.2. API for internet protocols

- Synchronous communication
  - *Send* and *receive* processes synchronize at every message
  - Both *blocking*:
    - whenever a *send* is issued, blocks until corresponding *receive* is issued; whenever a *receive* is issued, blocks until message arrives

- Asynchronous communication
  - *Send* from client is *non-blocking*, proceeds as soon as the message has been copied to a local buffer
  - *Receive* could be non-blocking or *blocking*, the latter has no disadvantages in system environment supporting multi-threading like Java
    - *Non-blocking receive* is not very useful (you cannot proceed without message)
4.2. API for internet protocols

   comparison

- Blocking
  Advantages: Ease of use and low overhead of implementation
  Disadvantage: low concurrency

- Non-blocking
  Advantages: Flexibility, parallel operations
  Disadvantages: Programming tricky: Program is timing-dependent (interrupt can occur at arbitrary time, and execution irreproducible)

→ Use blocking versions with multiple threads
4.2. API for internet protocols

Using blocking operation without penalty

- Some threads may be blocked while others continue to be active

requests (e.g., for web pages)
4.2. API for internet protocols

- TCP/IP layers

<table>
<thead>
<tr>
<th>Layers</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Messages (UDP) or Streams (TCP)</td>
</tr>
<tr>
<td>Transport</td>
<td>UDP or TCP packets</td>
</tr>
<tr>
<td>Internet</td>
<td>IP datagrams</td>
</tr>
<tr>
<td>Network interface</td>
<td>Network-specific frames</td>
</tr>
<tr>
<td>Underlying network</td>
<td></td>
</tr>
</tbody>
</table>
4.2. API for internet protocols

- The programmer’s conceptual view of a TCP/IP Internet

Recall: UDP, TCP, API

- **UDP (User Datagram Protocol):** offers no guarantee of delivery, a datagram protocol
- **TCP (Transmission Control Protocol):** reliable connection-oriented protocol, establishment of bi-directional communication channel
- **API (Application programming interface)**
4.2. API for internet protocols

Message destinations

- Messages are sent to a pair (Internet address, local port)
- Local port: an integer, message destination within a computer
  - Each computer has $2^{16}$ possible ports available to local processes for receiving messages
  - 0-1023: well-known for restricted use of privileged processes
  - 1024-49151: registered ports, holds service descriptions
  - 49152-65535: private purposes
  - In practice, non-restricted ports can be used for private purposes, then, cannot use registered services simultaneously
  - a port has exactly one receiver (except for multicast ports)
  - receiving process can have many ports for different message types
  - server processes usually publish their service-ports for clients
4.2. API for internet protocols

- **Sockets**: originated from BSD Unix
  - Provide an abstraction of endpoints for both TCP and UDP communication
  - Inter-process communication consists of transmitting a message between a socket in one process and a socket in another process
  - A socket must be bound to a local port and an IP address
  - Processes may use the same socket for sending and receiving messages
  - Sockets are typed/associated with a particular protocol, either TCP or UDP
4.2. API for internet protocols

- Java API for internet protocols

- For either TCP or UDP, Java provides an InetAddress class, which contains a method: getByName(DNS) for obtaining IP addresses, irrespective of the number of address bits (32 bits for IPv4 or 128 bits for IPv6) by simply passing the DNS hostname. For example, a user Java code invokes:

```java
InetAddress aComputer = InetAddress.getByName("www.cs.sfu.ca")
```

- The class encapsulates the details of the representation of the IP address
4.2. API for internet protocols

- **UDP Datagram communication**
  - **Steps:**
    - Client finds an available port for UDP connection
    - Client binds the port to local IP (obtained from InetAddress.getByName(DNS))
    - Server finds a designated port, publicizes it to clients, and binds it to local IP
    - Sever process issues a receive method and gets the IP and port # of sender (client) along with the message
  - **Issues**
    - **Message size** – receiver needs to specify a buffer of certain size to receive a massage. If message too big, truncated on arrival
    - **Blocking** – send is non-blocking, returns when the message gets the UDP and IP layers; receive is blocking until a datagram is received or timeout
    - **Timeouts** – reasonably large time interval can be set on receiver sockets to avoid indefinite blocking if required by program
    - **Receive from any** – no specification of sources (senders)
4.2. API for internet protocols

- **UDP Failure Models:**
  - Omission failure: due to omission of send or receive (either checksum error or no buffer space at source or destination)
  - Ordering failure: due to out-of-order delivery
  - Applications using UDP are left to provide their own checks to achieve the quality of reliable communication they require (why and how?)
    - UDP lacks built-in checks
    - but failure can be modeled by implementing an ACK mechanism
4.2. API for internet protocols

UDP client sends a message to the server & gets a reply

```java
import java.net.*; //defines socket-related classes
import java.io.*; //defines stream-related classes
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();}
    }
}
4.2. API for internet protocols

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

```java
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();}
}
```

2005/9/13
4.2. API for internet protocols

- TCP Stream Communication issues:
  - Message sizes – user application has option to choose how much data it writes to a stream or reads from it
  - Lost messages / Flow control: dealt with by TCP
  - Message duplication and ordering – message identifiers are associated with each IP packet so as for recipient to detect and reject duplicates or re-order
  - Message destinations – a connection is established first. Once established, no IP addresses in packets needed (Each connection socket is bidirectional – using two streams: output/write and input/read)
    - To establish a connection, client sends `connect` request to server, then server sends `accept` request to client
    - Could be a overhead for a single C-S request and reply
4.2. API for internet protocols

- **TCP Stream Communication: other issues**
  - Matching of data items – both client/sender and server/receiver must agree on data types in the stream
  - Threads – server creates a separate thread in accepting a connection, then it can block waiting for input without delaying other clients

- **Failure Model**
  - Integrity: uses checksums for detection/rejection of corrupt data and seq #s for detection/rejection of duplicates
  - Validity: uses timeout with retransmission techniques to take care of packet losses

- Uses – TCP sockets used for such services as: HTTP, FTP, Telnet, SMTP
4.2. API for internet protocols

TCP client makes connection to server, sends request and receives reply

```java
import java.net.*; //Defines socket-related classes
import java.io.*; //Defines stream-related classes
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        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());}}
```
4.2. API for internet protocols

TCP server makes a connection for each client and then echoes the client’s request

```java
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(Exception e) {System.out.println("Listen :"+e.getMessage());}
    }
}

// this figure continues on the next slide
4.2. API for internet protocols

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){/*close failed*/}}
}