SFDV3006
Concurrent Programming

Lecture 8 – Distributed Programming

Distributed Programming

- Not all concurrent programs are distributed – working over many different machines connected by a network
- Distributed programs are concurrent programs where different parts of the programs are running concurrently on different machines
- Distributed programming covers many different architectures and paradigms. In this lecture we cover the following:
  - Client server using sockets
  - RMI

Basic distributed programming using clients and servers

- Clients and servers
  - server accepts connections from one or more clients
  - clients need to connect to server
- Identifying a machine
  - clients need to know the IP address of server
- Identifying a port
  - multiple ports exist on each machine
  - client and server need to agree on port
- Setting up a **socket**
  - the actual connection between the client and server
  - exchange information using Java streams

Internet addresses

- **In order to use a service you must be able to find it. The Internet uses an address scheme for machines so that they can be located.**
- **The address is a 32 bit integer which gives the IP address. This encodes a network ID and more addressing.**
- **The network ID falls into various classes according to the size of the network address.**
- **Example 138.92.6.11**
- **Special IP addresses**
  - 127.0.0.1 is the local machine
  - 192.168.X.X are addresses which can be used inside a LAN, but will not be passed by any router. The "private" addresses are typically used for e.g. home networks where no IP provider has allocated addresses
  - 224.0.0.0-239.255.255.255 are multicast addresses.
  - 164.254.X.X are zero configuration addresses for use when there is no manual configuration and no DHCP server

InetAddress

- The Java class **InetAddress** can be used to get/manipulate internet addresses. Methods include

  static InetAddress getByName(String host)

  static InetAddress getLocalHost() String getHostAddress(); // in dotted form String getHostName();

- Example (see next slide)

InetAddress – example

```
import java.net.InetAddress;
import java.net.UnknownHostException;
public class GetInetInfo{
    public static void main(String[] args){
        if (args.length != 1) {
            System.err.println("Usage: java GetInetInfo address");
            return;
        }
        InetAddress address = null;
        try {
            address = InetAddress.getByName(args[0]);
        } catch(UnknownHostException e) { e.printStackTrace();}

        System.out.println("Host name: " + address.getHostName());
        System.out.println("Host address:"+address.getHostAddress());
}
}
```

Port addresses

- **A service exists on a host, and is identified by its port.**
- **A port is a 16 bit number (0-65,000).**
- **To send a message to a server you send it to the port for that service of the host that it is running on.**
- **This is not location transparency**
- **Certain of these ports are "well known". On Unix, they are listed in the file /etc/services. For example,**
  - ftp: 21/tcp
  - http: 80/tcp
  - time: 37/tcp
  - time: 37/udp
  - finger 79/tcp
- Ports in the region 1-255 are reserved by the IETF (Internet Engineering Task Force). The system may reserve more. User processes (in Unix) may have their own ports above 1023.

Sockets

- A socket is a reliable connection for the transmission of data between two hosts.
- Sockets isolate programmers from the details of packet encodings, lost and retransmitted packets, and packets that arrive out of order.
- A socket programming construct can make use of either the UDP or TCP protocol.
- Sockets that use UDP for transport are known as **datagram sockets**, while sockets that use TCP are termed **stream sockets**.
- There are limits. Sockets are more likely to throw IOExceptions than files, for example.

Socket Operations

- There are four fundamental operations a socket performs. These are:
  1. Connect to a remote machine
  2. Send data
  3. Receive data
  4. Close the connection
- A socket may not be connected to more than one host at a time.
- A socket may not reconnect after it is closed.

The java.net.Socket class

- The *java.net.Socket* class allows you to create socket objects that perform all four fundamental socket operations.
- You can connect to remote machines; you can send data; you can receive data; you can close the connection.
- Each *Socket* object is associated with exactly one remote host. To connect to a different host, you must create a new *Socket* object.

Sending and Receiving Data

- Data is sent and received with output and input streams.
- There are methods to get an input stream for a socket and an output stream for the socket.

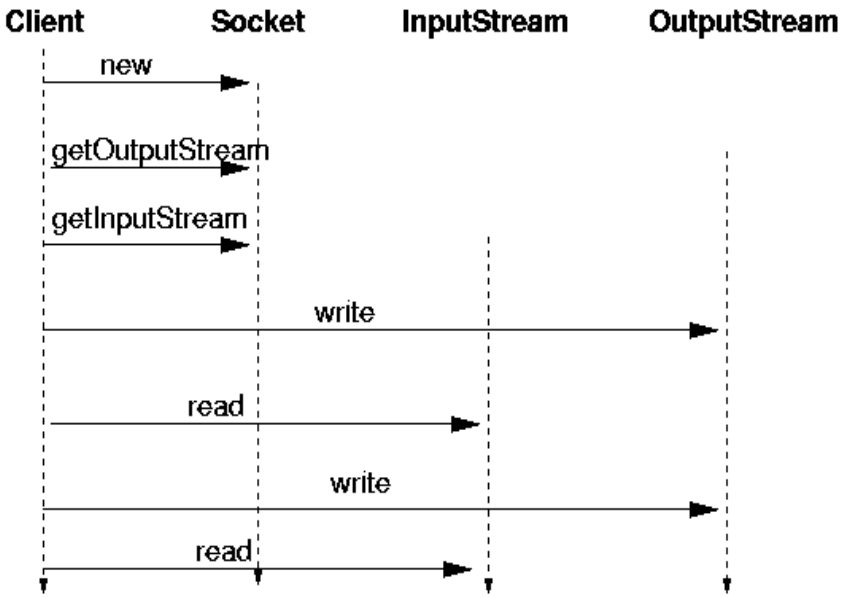
  public InputStream getInputStream() throws IOException
  public OutputStream getOutputStream() throws IOException
- There's also a method to close a socket:

  public synchronized void ***close()*** *throws IOException*
- Writing Output to a Socket
  - The *getOutputStream()* method returns an output stream which writes data to the socket.
  - Most of the time you'll chain the raw output stream to some other output stream or writer class to more easily handle the data.
- Reading Input from a *Socket*
  - The *getInputStream()* method returns an *InputStream* which reads data from the socket.
  - You can use all the normal methods of the *InputStream* class to read this data.
  - Most of the time you'll chain the input stream to some other input stream or reader object to more easily handle the data.

Connection oriented (TCP)

- One process (server) makes its socket known to the system using "bind". This will allow other sockets to find it.
- It then "listens" on this socket to "accept" any incoming messages.
- The other process (client) establishes a network connection to it, and then the two exchange messages.
- As many messages as needed may be sent along this channel, in either direction.
- The client typically creates a Socket, gets I/O streams from it, and then does consecutive writes and reads.

Client



Socket example - TimeClient.java

```
import java.io.*;
import java.net.*;

public class TimeClient{
    public static void main(String args[]) throws IOException{

        //Create a socket to connect to the server
       Socket socket = new Socket("192.168.1.3", 37);

        //create an InputStream to read the response
       BufferedReader br =  new BufferedReader(
                                new InputStreamReader(socket.getInputStream()));

        String line;
        while((line = br.readLine()) != null)
            System.out.println(line);

        br.close();
        socket.close();
    }
}
```

Compare with server on slide 20

This client simply connects to time server gets the time and prints it and then closes the connection

Server

- A server uses a **ServerSocket** to bind to a port and listens on it.
- There are two ends to each connection: the client, that is the host that initiates the connection, and the server, that is the host that responds to the connection.
- Clients and servers are connected by sockets.
- A server, rather than connecting to a remote host, a program waits for other hosts to connect to it.
- When a new client request is accepted, it returns a **Socket** which is connected to the client.
- The server uses this socket to talk to the client, typically reading requests and responding to them

The java.net.ServerSocket Class

- The *java.net.ServerSocket* class represents a server socket.
- A *ServerSocket* object is constructed on a particular local port. Then it calls *accept()* to listen for incoming connections.
- *accept()* blocks until a connection is detected. Then *accept()* returns a *java.net.Socket* object that performs the actual communication with the client.
- There are three constructors that let you specify the port to bind to, the queue length for incoming connections, and the IP address to bind to:

  public ServerSocket(int port) throws IOException
  public ServerSocket(int port, int backlog) throws IOException
  public ServerSocket(int port, int backlog, InetAddress networkInterface) throws IOException

ServerSocket - contd

- Normally you only specify the port you want to listen on, like this:

```
try {
    ServerSocket ss = new ServerSocket(7979);
 } catch (IOException e) { System.err.println(e); }
```

- When a *ServerSocket* object is created, it attempts to bind to the port on the local host given by the port argument.
- If another server socket is already listening to the port, then a *java.net.BindException*, a subclass of *IOException*, is thrown.
- No more than one process or thread can listen to a particular port at a time. This includes non-Java processes or threads.
- For example, if there's already an HTTP server running on port 80, you won't be able to bind to port 80.
- On Unix systems (but not Windows or the Mac) your program must be running as root to bind to a port between 1 and 1023.
- 0 is a special port number. It tells Java to pick an available port.

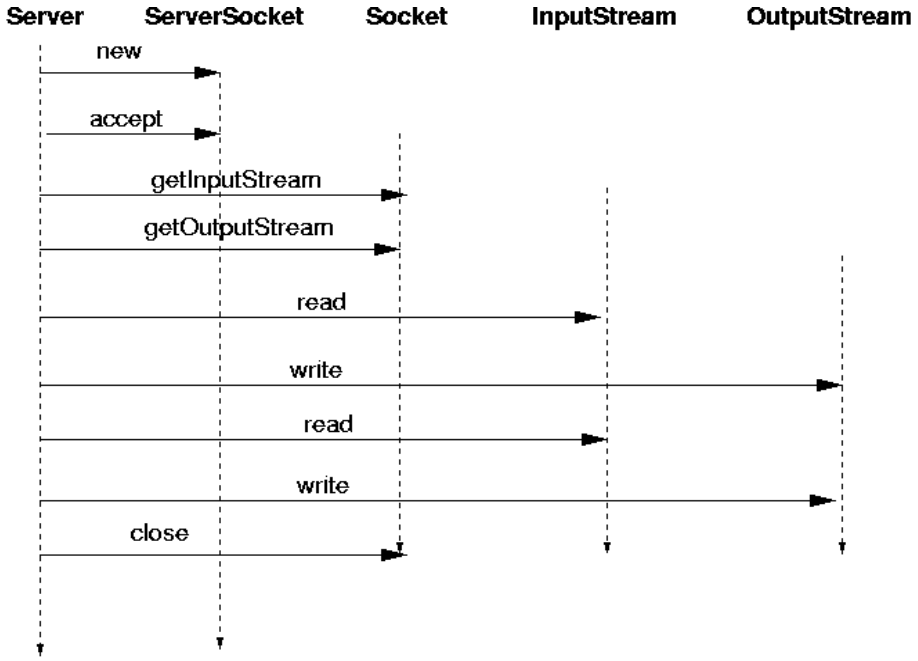
Reading data with a ServerSocket

- *ServerSocket* objects use their *accept()* method to connect to a client.

  public Socket accept() throws IOException
- There are no getInputStream() or getOutputStream() methods for ServerSocket.
- *accept()* returns a *Socket* object, and its *getInputStream()* and *getOutputStream()* methods provide streams.
- Example

```
try {
    ServerSocket ss = new ServerSocket(2345);
    Socket s = ss.accept();
    PrintWriter pw = new PrintWriter(s.getOutputStream());
    pw.println("Hello There!");
     pw.println("Goodbye now.");
    s.close();
} catch (IOException e) { System.err.println(e); }
```

Server



ServerSocket - TimeServer.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
public class TimeServer{
        public static void main(String args[]) throws Exception{
             //bind this server to a port
             ServerSocket server = new ServerSocket(9876);
            while (true){
                  //create a Socket object which will talk to the client
                 client = server.accept(); //blocks until a client connects

                 //get an OutputStream to write to the client
                 PrintStream ps = new PrintStream( client.getOutputStream() );
                  //write to the client
                 ps.println("Time from server:" + new Date());

                 //close the client socket
                 client.close();
             }
        }
}
```

Compare with client on slide 14

Timeout

- The server may wish to timeout a client if it does not respond quickly enough i.e. does not write a request to the server in time. This should be a long period (several minutes), because the users may be taking their time.
- The client may want to timeout the server (after a much shorter time).

```
try {
    socket.setSoTimeout(10000); // 10 seconds
} catch(SocketException e) { e.printStackTrace(); }
```

- Staying alive: A client may wish to stay connected to a server even if it has nothing to send. It can **setKeepAlive(true)** on the socket

Handling multiple clients

- The *TimeServer* example above handles one client at a time – this can cause delay and waiting for client if there are many clients to be serviced
- All types of servers need to be highly threaded – to handle multiple concurrent clients
- To handle multiple clients
  - In the server, create a thread for each client
  - Pass socket to constructor of thread so that thread has access to (client) socket
  - No difference in client, the client does not know that there are many other clients being serviced concurrently

Why multi thread the servers?

- When a client sends a request to the server, the server accepts the request, does some processing and sends a response back to the client.
- Processing can take time and the server is blocked during processing. Any other request to the server is denied until the server completes the current request. This can lead to timeouts and disconnects by other clients which are in the queue.
- A multithreaded server can handle many requests at a time without blocking and gives reasonable response time to clients.
- Threads allow server to handle many requests concurrently.
- Threading makes servers scalable

How to multi thread a server?

- Server program is now divided to into parts – the ***main thread*** which listens for connections from the clients
- And the ***request handler*** – the thread which actually services the client request
- The *main thread* listens for and accepts connections. Whenever a client connects, a new thread is created to handle the client request. This new thread handles processing the request and communication with the client.
- The main thread does not block and continues listening for connections.
- The main thread and the service threads run concurrently.
- In actual use servers use ***thread pools*** instead of creating new threads on the fly

Multithreading a server example – Time server

```
import java.io.*;
import java.net.*;
import java.util.Date;
public class TimeServerMT {
    public static void main(String args[]) throws Exception {
        //bind this server to a port
        ServerSocket server = new ServerSocket(9876);

        while (true) {
            //create a Socket object which will talk to the client
            //blocks until a client connects
            Socket client = server.accept();

            RequestHandler rh = new RequestHandler(client);
            rh.start();
        }
    }
}
```

For every incoming request the server creates a thread to handle the client request

Multithreading a server – the Request Handler

```
import java.io.*;
import java.net.*;
import java.util.Date;

public class RequestHandler extends Thread {
    private Socket clientSocket = null;

    public RequestHandler(Socket _clientSocket) {
        clientSocket = _clientSocket;
    }

    public void run() {
        try {
            //get an OutputStream to write to the client
            PrintStream ps = new PrintStream( clientSocket.getOutputStream() );

            //write to the client
            ps.println("Time from server:" + new Date());
            //close the client socket
            clientSocket.close();
        } catch (IOException ioex) { ioex.printStackTrace();}
    }
}
```

Code to service the client is in the request handler thread

Java Object Serialization

- The Java API provides capability to transmit Java objects over the socket connection. This capability comes from two classes in the *java.io* package: *ObjectInputStream* and *ObjectOutputStream*.
- The **ObjectOutputStream** has the following methods:

  public ObjectOutputStream(OutputStream os);

  *public final void **writeObject**(Object o);*
- The **ObjectInputStream** has the following methods:

  public ObjectInputStream(InputStream is);

  *public final Object **readObject**();*

Objects over sockets

- Object serialization works for sockets as well
- Clients can send object to the servers and servers can respond with objects
- This is better than dealing with raw data using bytes etc
- Also when you have lot of structured data such as student records – it is better to send as a collection of objects since this can be used directly in the program without much conversion

Objects over sockets

- Objects are read and written from socket input and output streams respectively
- A client creates a serializable object and writes to the output stream of the socket using the *writeObject()* method of the *ObjectOutputStream*.
- A server creates an input stream and retrieves the object using the *readObject()* method of the *ObjectInputStream*.
- A client receives the server response also as an object. To retrieve the response objec the client uses the *readObject()* method of the *ObjectInputStream*
- The *readObject()* method always returns an instance of *java.lang.Object* – which is the parent class of all classes in Java. You must cast it to the required object.

Object serialization example with this lecture
SortServer.java and ***SortClient.java***

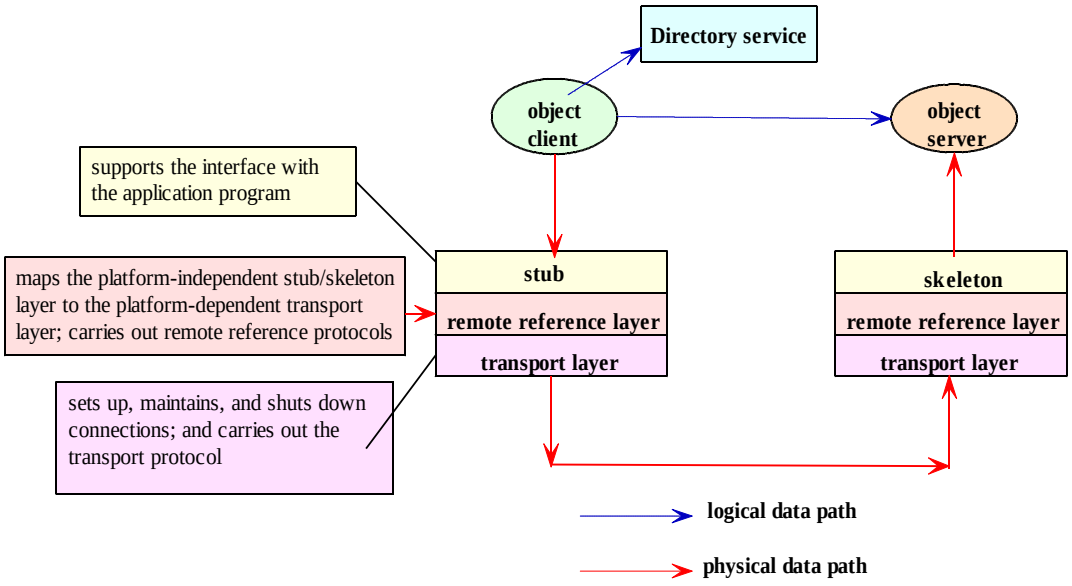
RMI – Remote Method Invocation

- Object oriented version of Remote Procedure Call (RPC)
- Java based. All distributed components are ***Java objects*** (distributed objects)
- Java distributed object model
  - Distributed object applications comprise of two separate programs – a server and a client
  - A **server** application creates ***remote objects***, makes references to those objects accessible and waits for clients to invoke methods.
  - A **client** application gets a ***remote reference*** to one or more remote objects and invokes methods on them.
  - Clients of remote objects interact with ***remote interfaces***, never with the implementation classes of those interfaces.

Java Distributed Object model

- Distributed object applications need to:
  - Locate remote objects: an application can register its remote objects with a name server. Ex: RMI registry
  - Communicate with remote objects
  - Load *class bytecodes* for objects that are passed as parameters or return values.

RMI architecture



RMI architecture

- Client side architecture
  - Stub layer: A client process's remote method invocation is directed to a proxy object, known as stub. It forwards them to the next layer below.
  - Remote Reference layer: interprets and manages references made from client to remote service objects and issues IPC (socket) operations to next layer
  - Transport layer: is TCP based. Carries out the IPC, transmitting data representing the call to the remote host

RMI architecture – server side

- Skeleton: carries out conversation with the stub
  - it reads parameters for the method call from the link
  - makes call to the remote service implementation object
  - writes the return value back to the stub
- Remote reference layer: manages and transforms the remote reference originating from the client to local references that are understandable to the skeleton
- Transport layer: connection –oriented transport layer. TCP in TCP/IP network architecture

Object registry

- Also called Naming service, directory service or Name server
- With RMI different directory services can be used for registering a distributed object such as Java Naming and Directory Interface (JNDI)
- *Allows runtime binding of clients to servers*
- *Basically a table*
- *Central place to perform access control*
- *Fail over: If one server fails, use another.*
- *Need to know where the name server is running*
- *Java Remote objects are represented by:*
  - *Machine (IP address)*
  - *Port number where the server is listening on*
  - *Unique object id inside the machine*
- JDK comes with a directory service called *rmiregistry*

RMI API

- RMI API comes as a part of the standard JDK API in *java.rmi* package
- *java.rmi.**Remote*** interface
  - A remote interface must directly or indirectly extend this interface
  - Each method declaration in a remote interface must include the exception *java.rmi.RemoteException*
- *java.rmi.server.**RemoteObject***
  - Provides implementation for hashcode(), equals() and toString() methods.
  - Provides implementation for the special writeObject() and readObject() methods called by the object serialization mechanism
- *java.rmi.server.**RemoteServer***
  - Abstract class which provides utility functions including logging facilities
- *java.rmi.server.**UnicastRemoteObject***
  - Provides methods to create remote objects and export them (make them available to remote clients)

RMI implementation example

- Key components / class definitions in RMI
  - Name server (provides location information of services)
  - Interface definition of server code
  - Implementation of server
  - Implementation of client
- DateServer example

RMI example: date server interface definition

- Interface must extend the *java.rmi.Remote* interface
- Each method in the interface must throw a *RemoteException*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;
public interface DateServer extends Remote{
    public Date getDate() throws RemoteException();
}
```

Implementation of RMI server

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;
public class DateServerImpl extends UnicastRemoteObject
                                    implements DateServer {

    public DateServerImpl() throws RemoteException {}

    public Date getDate()  throws RemoteException{
          return new Date();
    }

    public static void main(String args[ ]) { //start and bind the server
              DateServerImpl dateServer = new DateServerImpl();
              Naming.bind("DateServer", dateServer);
    }
}
```


Implementation of RMI client

```
import java.rmi.Naming;
import java.util.Date;
public class DateClient {
    public static void main(String args[]){
          DateServer dateServer =
           (DateServer) Naming.lookup("rmi://localhost/DateServer");
          Date now = dateServer.getDate();
          System.out.println(now);
    }
}
```

Thread usage

- A method dispatched by RMI runtime to a remote object may or may not execute in a separate thread
- Calls originating from different client VM's will execute in different threads
- Some calls originating from the same client VM will execute in the same thread and others in different threads
- The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads

Distributed Programming in Java

- Basic network programming involves identifying a machine and port and setting up a socket
- Java provides a lot of support for distributed programming and systems
  - JDBC for platform independent data access
  - Servlets and JSP provides services through a web server
  - RMI and support for CORBA to allow remote procedure calls
  - EJBs to implement servers-side business components
  - Web services
  - Grid computing
  - Cloud computing