Distributed Systems

Inter-Process Communication

Practical Issues

Dynamic and Distributed Information Systems

Today's Agenda

- Programming with UDP
- Programming with TCP
  - Threads
- Programming with RMI
- Marshalling
  - Corba
  - XML
  - Java Serialization
- 1st Lab Assignment

Programming with UDP

Dynamic and Distributed
Information Systems

UDP

- Connectionless protocol
- Communication by individual packets (datagrams)
  - Each datagram is independent
- Multiple clients can be accessing the server intermittently

UDP: Server (1/2)

```
package udp;


import java.io.IOException;
import java.net.*;


public class Server
{
  static DatagramPacket packetIn;
  static DatagramPacket packetOut;

  public static void main(String[] args)
  {
    if (args.length != 1)
    {
      System.err.println("Syntax: udp_server <port>\n");
      System.exit(-1);
    }

    int port = Integer.valueOf(args[0]);
    startServer(port);
  }
```

UDP: Server (2/2)

```
private static void startServer(int serverPort)
{
  try {
    DatagramSocket socket = new DatagramSocket(serverPort);
    byte[] buffer = new byte[1024];

    while (true)
    {
      // Receive client's packet
      packetIn = new DatagramPacket(buffer, buffer.length);
      socket.receive(packetIn);

      // Extract message from the packet and print it
      byte[] data = packetIn.getData();
      int length = packetIn.getLength();
      String str = new String(data, 0, length);
      System.out.println("Client "+packetIn.getSocketAddress()+" sent '"+str+"'");

      // Reply to the client
      InetAddress clientAddr = packetIn.getAddress();
      int clientPort = packetIn.getPort();
      packetOut = new DatagramPacket(
          str.toUpperCase().getBytes(),
          str.getBytes().length,
          clientAddr,
          clientPort);
      socket.send(packetOut);
    }
  }
  catch (SocketException e) {
    System.err.println("SocketException: "+ e);
  }
  catch (IOException e) {
    System.err.println("IOException: "+e);
    System.exit(-1);
  }
 }
}
```

UDP: Client (1/2)

```
package udp;


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.*;


public class Client
{
  static DatagramPacket packetIn;
  static DatagramPacket packetOut;
  static BufferedReader stdin;

  public static void main(String[] args)
  {
    if (args.length != 2)
    {
      System.err.println("Syntax: udp_client <server> <port>\n");
      System.exit(-1);
    }

    String server = args[0];
    int port = Integer.valueOf(args[1]);

    connectServer(server, port);
  }



  private static void connectServer(String server, int serverPort)
  {
    try {
      DatagramSocket socket = new DatagramSocket();
      socket.setSoTimeout(100);
      byte[] buffer = new byte[1024];
```

UDP: Client (2/2)

```
      InetAddress serverAddr = InetAddress.getByName(server);

      stdin = new BufferedReader(new InputStreamReader(System.in));
      while (true)
      {
        // Read message from user
        String myMessage = stdin.readLine();

        // Send it to the server
        packetOut = new DatagramPacket(
            myMessage.getBytes(),
            myMessage.getBytes().length,
            serverAddr,
            serverPort);
        socket.send(packetOut);

        // Receive packet from server
        packetIn = new DatagramPacket(buffer, buffer.length);
        try {
          socket.receive(packetIn);
        }
        catch (SocketTimeoutException e) {
          continue;
        }

        // Extract message from the packet and print it
        byte[] data = packetIn.getData();
        int length = packetIn.getLength();
        String str = new String(data, 0, length);
        System.out.println("Server "+packetIn.getSocketAddress()+" replied '"+str+"'");
      }
    }
    catch (UnknownHostException e) {
      System.out.println("Unknown IP address for server");
      System.exit(-1);
    }
    catch (SocketException e) {
      System.err.println("SocketException: "+ e);
      System.exit(-1);
    }
    catch (IOException e) {
      System.err.println("IOException: "+e);
      System.exit(-1);
    }
  }
}
```

Programming with TCP

Dynamic and Distributed
Information Systems

TCP

- Connection-oriented
- Communication through byte stream
- We need threads to support multiple concurrent clients

TCP: Server (1/2)

```
package tcp;

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

public class Server
{
  private static Socket socket=null;
  private static DataInputStream input=null;
  private static DataOutputStream output=null;

  public static void main(String[] args)
  {
    if (args.length != 1) {
      System.err.println("Syntax: tcp_server <port>\n");
      System.exit(-1);
    }

    int port = Integer.valueOf(args[0]);
    startServer(port);
  }



  private static void startServer(int port)
  {
    ServerSocket serverSocket = null;

    try {
      serverSocket = new ServerSocket(port, 3);
      System.out.println("Server listening to port: "+serverSocket.getLocalPort());
    }
    catch (Exception e) {
      System.out.println( "Could not create server socket: " + e );
      System.exit(-1);
    }
```

TCP: Server (2/2)

```
// chat with the client until he breaks the connection or says "bye"
try {   // deal with catastrophic errors
  while (true) {
    System.out.println("Waiting for client connection.");
    try {  // deal with broken connections to the current client
      socket = serverSocket.accept();
      System.out.println("socket listening to port: "+socket.getLocalPort());

      input = new DataInputStream(socket.getInputStream());
      output = new DataOutputStream(socket.getOutputStream());

      System.out.println("New client "+socket.getRemoteSocketAddress()+" connected.");

      String str = new String("");
      while (!str.equals("bye")) {
        // Receive client's message
        str = input.readUTF();

        // Print client's message
        System.out.println("Client "+socket.getRemoteSocketAddress()+" sent '"+str+"'");

        // Reply to the client
        output.writeUTF(str.toUpperCase());
      }
    }
    catch (Exception e) {
      System.out.println("Connection to current client broken.");
    }
  }
}
catch (Exception e) {
  System.out.println("Fatal server error: " + e);
}
  }
}
```

TCP: Client (1/2)

```
package tcp;

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


public class Client
{
  private static Socket socket=null;
  private static DataInputStream input=null;
  private static DataOutputStream output=null;
  private static BufferedReader stdin=null;

  public static void main (String args[])
  {
    if (args.length != 2)
    {
      System.err.println("Syntax: tcp_client <server> <port>\n");
      System.exit(-1);
    }

    String server = args[0];
    int port = Integer.valueOf(args[1]);

    chat(server, port);
  }
```

TCP: Client (2/2)

```
private static void chat(String server, int port)
{
  try {
    // Setup connection to server, and input/output streams
    try {  // handle bad host/port errors
      socket = new Socket(server, port);
    }
    catch (UnknownHostException e) {
      System.out.println("Unknown IP address for server");
      System.exit(1);
    }
    catch(IOException e) {
      System.out.println("No server found at specified port.");
      System.exit(1);
    }

    input = new DataInputStream(socket.getInputStream());
    output = new DataOutputStream(socket.getOutputStream());
    stdin = new BufferedReader(new InputStreamReader(System.in));

    String myMessage=new String("");
    String response=null;

    while (!myMessage.equals("bye"))
    {
      // Read message from user
      myMessage = stdin.readLine();
      output.writeUTF(myMessage);        // expect something from the server, output when it arrives

      response = input.readUTF();
      System.out.println("Server "+socket.getRemoteSocketAddress()+" replied '"+response+"'");
    }
  }
  catch(IOException e) {
    System.out.println("Broken connection with server.");
    System.exit(1);
  }
}
```

Use of threads on TCP servers

```
while (true)
    accept a connection ;
    create a thread to deal with the client ;  //non-blocking
end while
```

TCP: Multithreaded Server (1/3)

```
package tcp;

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

public class ServerMT
{
  public static void main(String[] args)
  {
    if (args.length != 1)
    {
      System.err.println("Syntax: tcp_server <port>\n");
      System.exit(-1);
    }

    int port = Integer.valueOf(args[0]);

    ServerMT tcpServer = new ServerMT();
    tcpServer.startServer(port);
  }
```

TCP: Multithreaded Server (2/3)

```
private void startServer(int port)
{
  ServerSocket serverSocket = null;

  try {
    serverSocket = new ServerSocket(port, 3);
    System.out.println("Server listening to port: "+serverSocket.getLocalPort());
  }
  catch (Exception e) {
    System.out.println( "Could not create server socket: " + e );
    System.exit(-1);
  }

  // chat with the client until he breaks the connection or says "bye"
  try {   // deal with catastrophic errors
    while (true)
    {
      System.out.println("Waiting for client connection.");

      Socket socket = serverSocket.accept();
      ConnectionHandler c = new ConnectionHandler(socket);
      c.start();
    }
  }
  catch (Exception e) {
    System.out.println("Fatal server error: " + e);
  }
}
```

TCP: Multithreaded Server (3/3)

```
public class ConnectionHandler extends Thread
{
  private Socket socket;

  public ConnectionHandler(Socket socket)
  {
    this.socket = socket;
  }

  public void run()
  {
    try
    {
      System.out.println("socket listening to port: "+socket.getLocalPort());

      DataInputStream input = new DataInputStream(socket.getInputStream());
      DataOutputStream output = new DataOutputStream(socket.getOutputStream());

      System.out.println("New client "+socket.getRemoteSocketAddress()+" connected.");

      String str = new String("");
      while (!str.equals("bye")) {
        // Receive client's message
        str = input.readUTF();

        // Print client's message
        System.out.println("Client "+socket.getRemoteSocketAddress()+" sent '"+str+"'");

        // Reply to the client
        output.writeUTF(str.toUpperCase());
      }
    }
    catch (IOException e)
    {
      System.out.println("Connection to current client broken.");
    }
  }
}
```

TCP with multiple threads

- Increased access transparency:
  - Each user “believes” that he is the only one using the server

TCP: Using object serialization (server)

```
public void run()
    {
      try
      {
        System.out.println("socket listening to port: "+socket.getLocalPort());

        ObjectOutputStream output = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream input = new ObjectInputStream(socket.getInputStream());

        System.out.println("New client "+socket.getRemoteSocketAddress()+" connected.");

        String str = new String("");
        while (!str.equals("bye")) {
          // Receive client's message
          str = (String)input.readObject();

          // Print client's message
          System.out.println("Client "+socket.getRemoteSocketAddress()+" sent '"+str+"'");

          // Reply to the client
          output.writeObject(str.toUpperCase());
        }
      }
```

TCP: Using object serialization (client)

```
ObjectOutputStream output = new ObjectOutputStream(socket.getOutputStream());
      ObjectInputStream input = new ObjectInputStream(socket.getInputStream());
      BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

      String myMessage=new String("");
      while (!myMessage.equals("bye"))
      {
        System.err.println("going to read user's input");
        // Read message from user
        myMessage = stdin.readLine();
        output.writeObject(myMessage);        // expect something from the server, output when it arrives

        String response = (String)input.readObject();
        System.out.println("Server "+socket.getRemoteSocketAddress()+" replied '"+response+"'");
      }
```

Programming with RMI

Dynamic and Distributed
Information Systems

RMI

- API style
- Communication through method calls: Increased transparency!
- Multiple clients can be accessing the server intermittently

RMI: configuration

- Define 3 parts:
  - Server Interface
  - Server Implementation
  - Client Implementation (uses Server Interface)
- The Server Interface
  - Should extend the **java.rmi.Remote** interface.
  - Each method must declare **java.rmi.RemoteException**
- The Server Implementation has the main function, which should:
  - Create and install a **security manager**
  - **Instantiate** one or more server objects
  - Register each server object to the **Naming Registry**

Deploying RMI apps

- Start the Naming Registry
  - rmiregistry (in Java bin directory)
- Store on disk a **policy** file, defining access privileges
- Invoke with –D directives (options):
  - -Djava.rmi.server.codebase=http://myhost/~myusrname/myclasses/
  - -Djava.security.policy=$HOME/mysrc/policy

RMI: Server Interface

```
package rmi;

import java.rmi.Remote;
import java.rmi.RemoteException;


public interface Server extends Remote
{
  public String capitalize(String message) throws RemoteException;
}
```

RMI: Server Implementation (1/2)

```
package rmi;

import java.net.*;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ServerImpl extends UnicastRemoteObject implements Server
{
  public static void main(String[] args)
  {
    try
    {
      if (System.getSecurityManager() == null)
        System.setSecurityManager(new RMISecurityManager());

      String registry = args[0];

      // Instantiate a new server object
      ServerImpl server = new ServerImpl();
      System.out.println("Server instantiated");

      // Bind server to a URL
      InetAddress myAddress = InetAddress.getLocalHost();
      String URL = "//" + registry + "/CapitalizeServer";

      System.out.println("Going to bind server at '"+URL+"'");
      Naming.rebind(URL, server);
      System.out.println("Server bound at '"+URL+"'");
    }
```

RMI: Server Implementation (2/2)

```
    catch (Exception e)
    {
      System.err.println("Something went wrong: "+e);
      System.exit(-1);
    }
  }

  public ServerImpl() throws RemoteException
  {
    super();
  }

  public String capitalize(String message) throws RemoteException
  {
    return message.toUpperCase();
  }

}
```

RMI: Client (1/2)

```
package rmi;

import java.io.*;
import java.net.*;
import java.rmi.Naming;
import java.rmi.RemoteException;


public class Client
{
  public static void main (String args[])
  {
    if (args.length != 1)
    {
      System.err.println("Syntax: client <registry>\n");
      System.exit(-1);
    }
    String registry = args[0];

    Server server = null;

    // Locate server Bind server to a URL
    InetAddress myAddress;
    try
    {
      myAddress = InetAddress.getLocalHost();
      String myHostname = myAddress.getCanonicalHostName();
      String URL = "//" + registry + "/CapitalizeServer";

      System.out.println("Going to lookup server at '"+URL+"'");
      server = (Server)Naming.lookup(URL);
      System.out.println("Server resolved at '"+URL+"'");
    }
    catch (Exception e)
    {
      System.err.println(e);
    }

    chat(server);
  }
```

RMI: Client (2/2)

```
private static void chat(Server server)
{
  BufferedReader stdin =
    new BufferedReader(new InputStreamReader(System.in));

  String myMessage=new String("");
  String response=null;

  try
  {
    while (!myMessage.equals("bye"))
    {
      // Read message from user
      myMessage = stdin.readLine();
      response = server.capitalize(myMessage);
      System.out.println("Server replied '"+response+"'");
    }
  }
  catch (RemoteException e) {
    System.err.println(e);
  }
  catch (IOException e) {
    System.err.println(e);
  }
}
}
```

Marshalling

Dynamic and Distributed
Information Systems

CORBA CDR for constructed types

- CDR: Common Data Representation

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

CORBA CDR message

index in sequence of bytes	4 bytes	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h "	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on "	
24–27	1934	*unsigned long*

The flattened form represents a `Person` struct with value:
`{'Smith', 'London', 1934}`

Indication of Java serialized form

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

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

XML structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```