COP 4610L: Applications in the Enterprise
Spring 2005

Java Networking and the Internet – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cop4610L/spr2005

School of Electrical Engineering and Computer Science
University of Central Florida

Networking

- Java's fundamental networking capabilities are declared by classes and interfaces of the `java.net` package, through which Java offers *stream-based communications*.
- The classes and interfaces of `java.net` also offer *packet-based communications* for transmitting individual packets of information. This is most commonly used to transmit audio and video over the Internet.
- We will focus on both sides of the client-server relationship.
- The client requests that some action be performed, and the server performs the action and responds to the client.

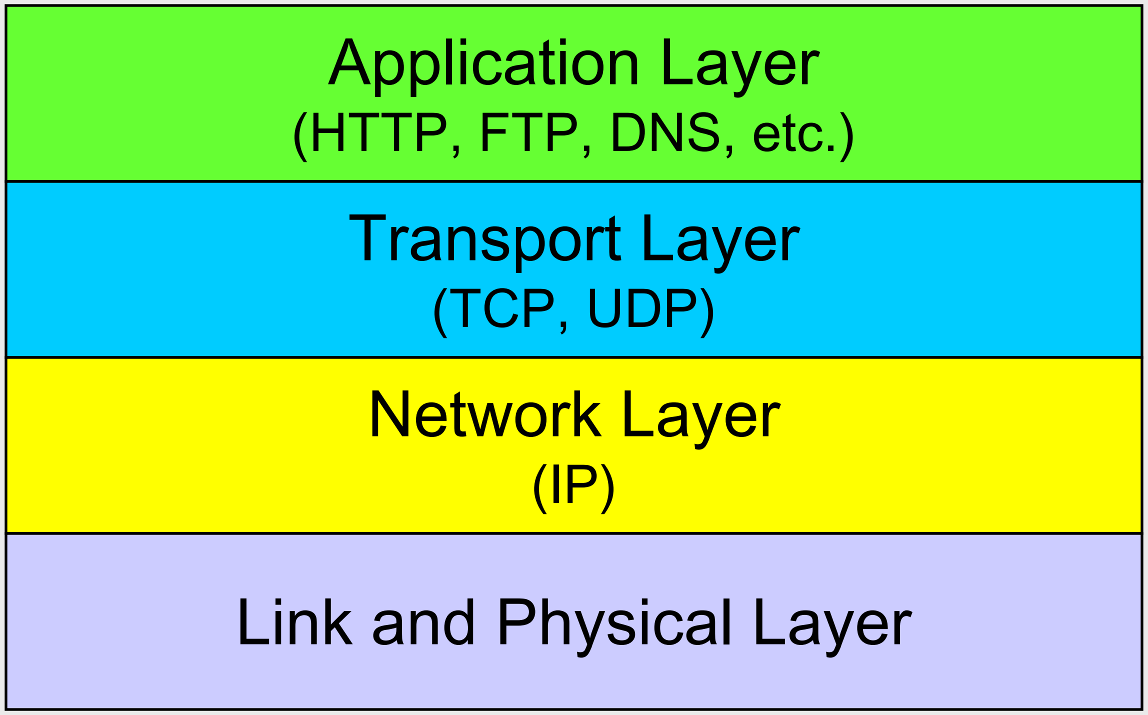
Networking (cont.)

- A common implementation of the request-response model is between Web browsers and Web servers.
  - When a user selects a Web site to browse through a browser (a client application), a request is sent to the appropriate Web server (the server application). The server normally responds to the client by sending the appropriate HTML Web page.

java.net

- “High-level” APIs
  - Implement commonly used protocols such as HTML, FTP, etc.
- “Low-level” APIs
  - Socket-based communications
    - Applications view networking as streams of data
    - Connection-based protocol
    - Uses TCP (Transmission Control Protocol)
  - Packet-based communications
    - Individual packets transmitted
    - Connectionless service
    - Used UDP (User Datagram Protocol)

Internet Reference Model



See page 21 in part 1 for a more detailed version of this diagram.

Sockets

- Java’s socket-based communications enable applications to view networking as if it were file I/O. In other words, a program can read from a socket or write to a socket as simply as reading from a file or writing to a file.

- A socket is simply a software construct that represents one endpoint of a connection.

- Stream sockets enable a process to establish a connection with another process. While the connection is in place, data flows between the processes in continuous streams.

- Stream sockets provide a connection-oriented service. The protocol used for transmission is the popular TCP (Transmission Control Protocol). Provides reliable , in-order byte-stream service

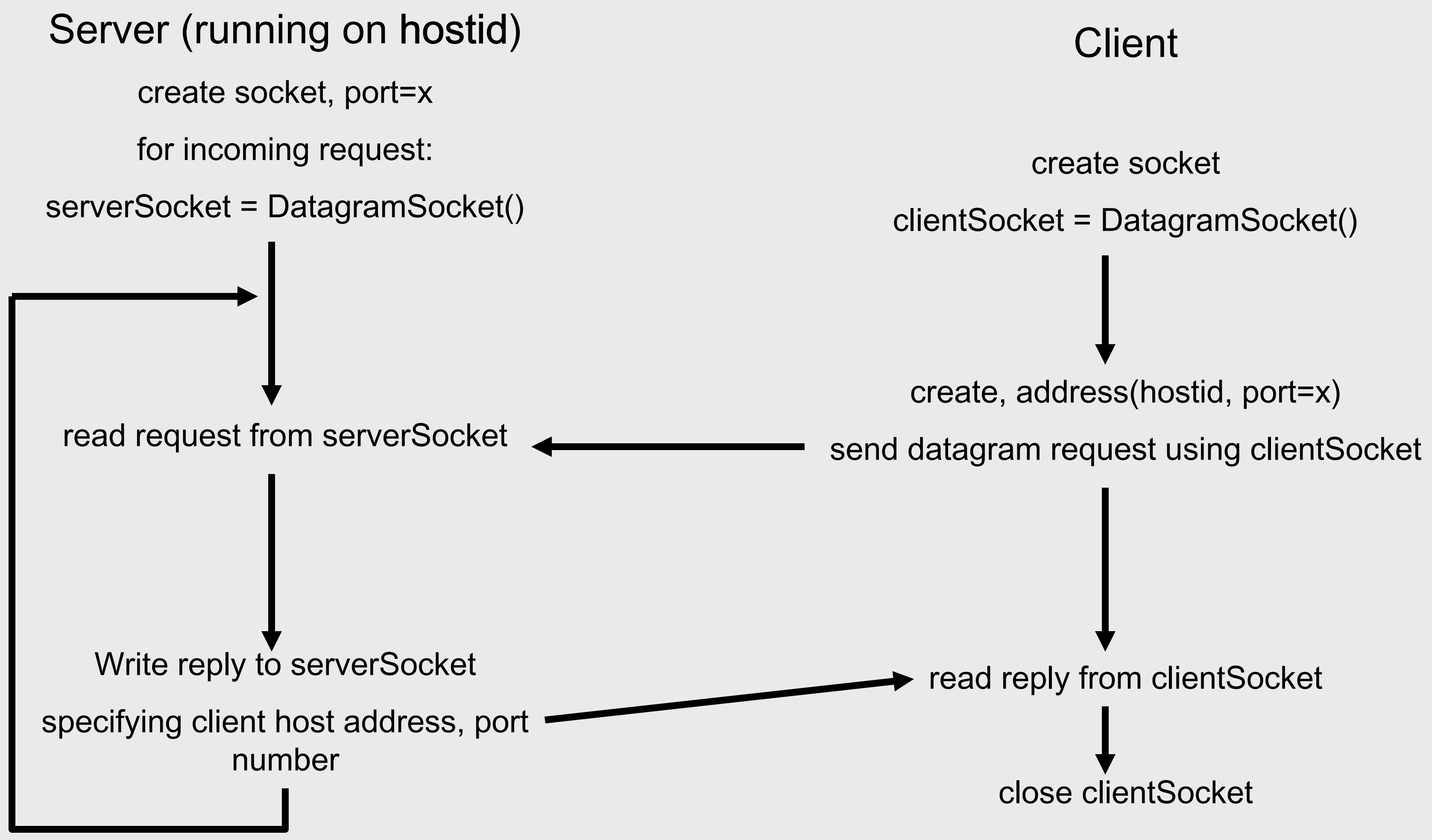
Sockets (cont.)

- Datagram sockets transmit individual packets of information. This is typically not appropriate for use by everyday programmers because the transmission protocol is UDP (User Datagram Protocol).
- UDP provides a connectionless service. A connectionless service does not guarantee that packets arrive at the destination in any particular order.
- With UDP, packets can be lost or duplicated. Significant extra programming is required on the programmer's part to deal with these problems.
- UDP is most appropriate for network applications that do not require the error checking and reliability of TCP.

Sockets (cont.)

- Under UDP there is no "connection" between the server and the client. There is no "handshaking".
- The sender explicitly attaches the IP address and port of the destination to each packet.
- The server must extract the IP address and port of the sender from the received packet.
- From an application viewpoint, UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server.

Example: client/server socket interaction via UDP


Server (running on hostid)
create socket, port=x
for incoming request:
serverSocket = DatagramSocket()
read request from serverSocket
Write reply to serverSocket
specifying client host address, port number
Client
create socket
clientSocket = DatagramSocket()
create, address(hostid, port=x)
send datagram request using clientSocket
read reply from clientSocket
close clientSocket

Example: Java server using UDP

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

class UDPServer {
   public static void main(String args[]) throws Exception
                {
                                //Create datagram socket on port 9876
                                DatagramSocket serverSocket = new DatagramSocket(9876);

                                byte[] sendData = new byte[1024];
                                byte[] receiveData = new byte[1024];

                                while (true)
                                {
                                                //create space for the received datagram
                                                DatagramPacket receivePacket = new
                                                                DatagramPacket(receiveData,
                                                                                                receiveData.length);
                                                //receive the datagram
                                                serverSocket.receive(receivePacket);

                                                String sentence = new String(receivePacket.getData());
```

Example: Java server using UDP (cont.)

```
                //get IP address and port number of sender
                          InetAddress IPAddress = receivePacket.getAddress();
                          int port = receivePacket.getPort();
                                    String capitalizedSentence =
                                                        sentence.toUpperCase();
                          sendData = capitalizedSentence.getBytes();
                          //create datagram to send to client
                          DatagramPacket sendPacket = new
                          DatagramPacket(sendData, sendData.length, IPAddress, port);
                          //write out the datagram to the socket
                          serverSocket.send(sendPacket);
                } //end while loop
        }
}
```

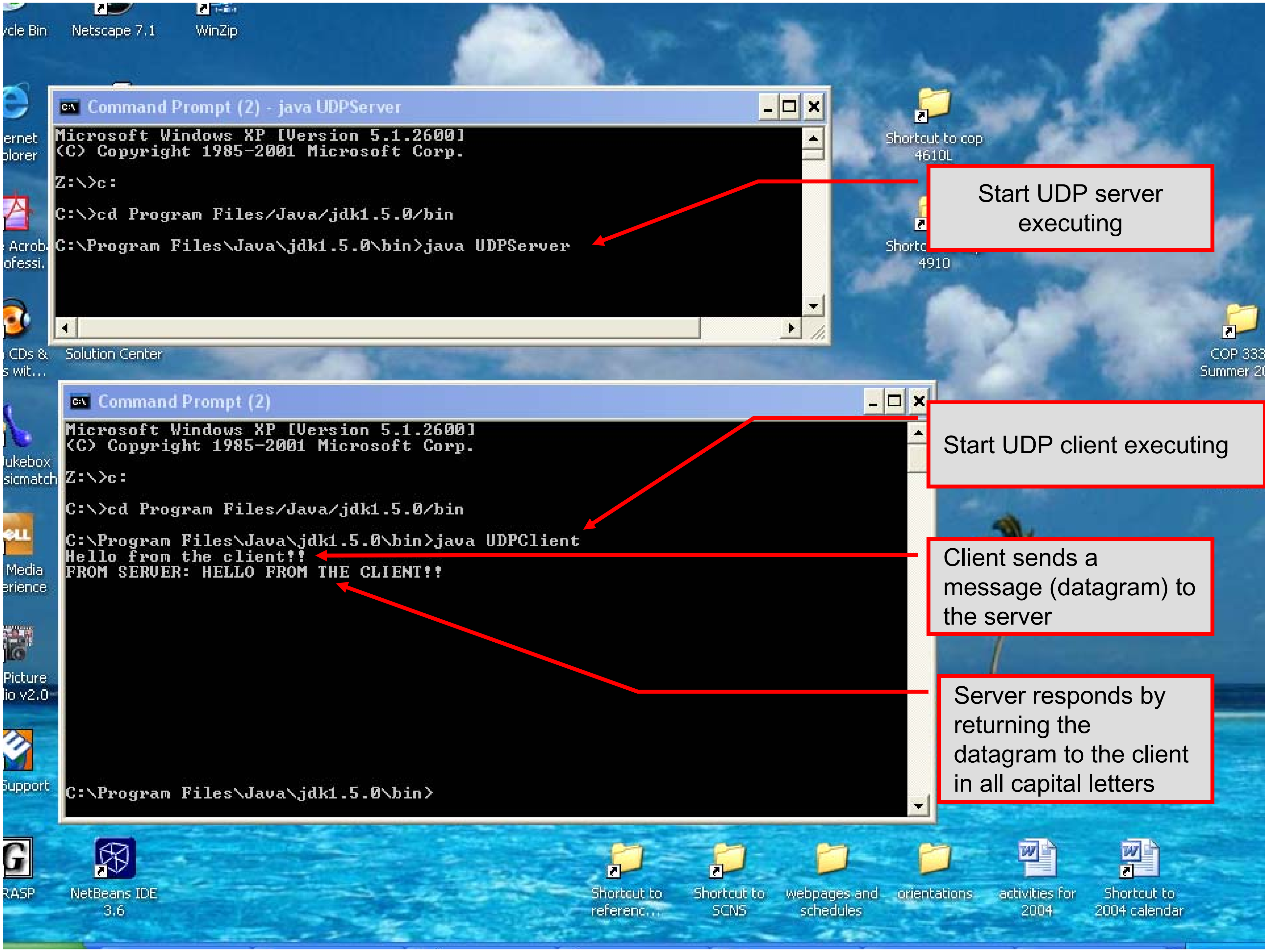
Example: Java client using UDP

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

class UDPClient {
   public static void main(String args[]) throws Exception
                 {
                                //Create input stream
                                BufferedReader inFromUser = new BufferedReader(new
                                                                    InputStreamReader(System.in));
                                //Create client socket
                                DatagramSocket clientSocket = new DatagramSocket();
                                //Translate hostname to IP address using DNS
                                InetAddress IPAddress = InetAddress.getByName("localhost");

                                byte[] sendData = new byte[1024];
                                byte[] receiveData = new byte[1024];

                                String sentence = inFromUser.readLine();
                                sendData = sentence.getBytes();
```

Example: Java client using UDP (cont.)

```
                    DatagramPacket sendPacket = new DatagramPacket(sendData,
                                        sendData.length, IPAddress, 9876);
                    clientSocket.send(sendPacket);

                    DatagramPacket receivePacket = new DatagramPacket(receiveData,
                                        receiveData.length);

                    clientSocket.receive(receivePacket);

                    String modifiedSentence = new String(receivePacket.getData());

                    System.out.println("FROM SERVER: " + modifiedSentence);
                    clientSocket.close();
          }
}
```

Try executing these two applications on your machine and see how it works. I’ve put the code for both on the code page.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>c:

C:\>cd Program Files/Java/jdk1.5.0/bin

C:\Program Files\Java\jdk1.5.0\bin>java UDPServer
```

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>c:

C:\>cd Program Files/Java/jdk1.5.0/bin

C:\Program Files\Java\jdk1.5.0\bin>java UDPClient
Hello from the client!!
FROM SERVER: HELLO FROM THE CLIENT!!



C:\Program Files\Java\jdk1.5.0\bin>
```

Start UDP server executing

Start UDP client executing

Client sends a message (datagram) to the server

Server responds by returning the datagram to the client in all capital letters

Socket Programming with TCP

- Server process must first be running (must have created a socket). Recall that TCP is not connectionless.
- Client contacts the server by creating client-local socket specifying IP address and port number of server process. Client TCP establishes connection to server TCP.
- When contacted by client, server TCP creates a new socket for server process to communicate with client.
  - Allows server to talk with multiple clients
  - Source port numbers used to distinguish clients
- From application viewpoint: TCP provides reliable, in-order transfer of bytes ("pipe") between client and server.

Establishing a Simple Server Using Stream Sockets

Five steps to create a simple stream server in Java:

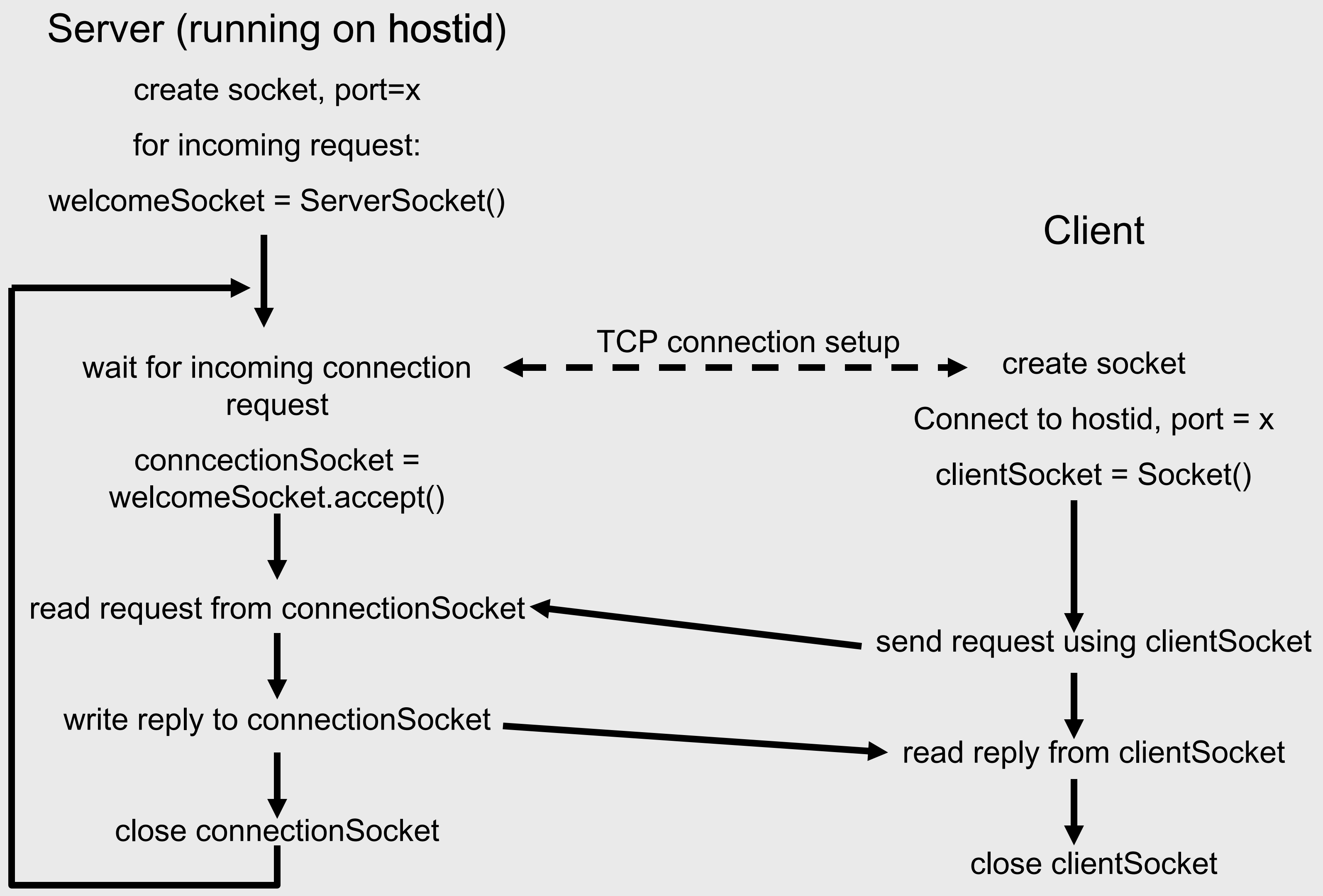
1. `ServerSocket` object. Registers an available port and a maximum number of clients.
2. Each client connection handled with a `Socket` object. Server blocks until client connects.
3. Sending and receiving data
   - `OutputStream` to send and `InputStream` to receive data.
   - Methods `getInputStream` and `getOutputStream` on `Socket` object.
4. Process phase. Server and client communicate via streams.
5. Close streams and connections.

Establishing a Simple Client Using Stream Sockets

Four steps to create a simple stream client in Java:

1. Create a `Socket` object for the client.
2. Obtains `Socket’s InputStream` and `OutputStream`.
3. Process information communicated.
4. Close streams and `Socket`.

Example: client/server socket interaction via TCP


Server (running on hostid)
create socket, port=x
for incoming request:
welcomeSocket = ServerSocket()
Client
TCP connection setup
wait for incoming connection request
create socket
Connect to hostid, port = x
clientSocket = Socket()
conncectionSocket = welcomeSocket.accept()
read request from connectionSocket
send request using clientSocket
write reply to connectionSocket
read reply from clientSocket
close connectionSocket
close clientSocket

Example: Java server using TCP

```
//simple server application using TCP

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

class TCPServer {
        public static void main (String args[]) throws Exception
        {
                String clientSentence;
                String capitalizedSentence;

                //create welcoming socket at port 6789
                ServerSocket welcomeSocket = new ServerSocket(6789);

                while (true) {
                        //block on welcoming socket for contact by a client
                        Socket connectionSocket = welcomeSocket.accept();

                        //create input stream attached to socket
                        BufferedReader inFromClient = new BufferedReader(new
                                InputStreamReader
                                        (connectionSocket.getInputStream()));
```

Example: Java server using TCP (cont.)

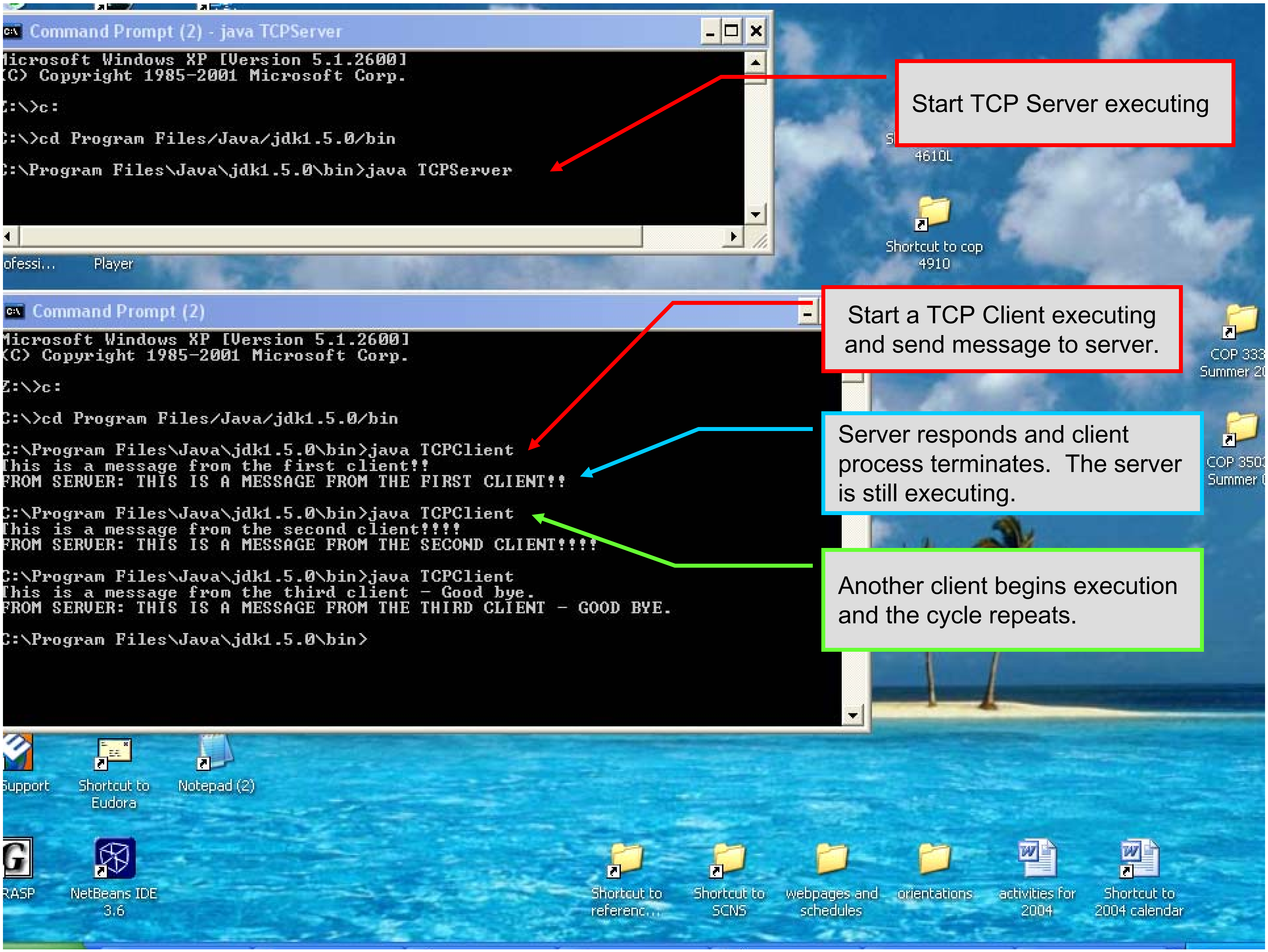
```
                        //create output stream attached to socket
                        DataOutputStream outToClient = new
                        DataOutputStream(connectionSocket.getOutputStream());

                        //read in line from the socket
                        clientSentence = inFromClient.readLine();

                        //process
                        capitalizedSentence = clientSentence.toUpperCase() + '\n';

                        //write out line to socket
                        outToClient.writeBytes(capitalizedSentence);
                }
        }
}
```

Example: Java client using TCP

```
//simple client application using TCP

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

class TCPClient {
        public static void main (String args[]) throws Exception
        {
                String sentence;
                String modifiedSentence;

                //create input stream
                BufferedReader inFromUser = new BufferedReader(new
                                InputStreamReader(System.in));

                //create client socket and connect to server
                Socket clientSocket = new Socket("localhost", 6789);

                //create output stream attached to socket
                DataOutputStream outToServer = new
                        DataOutputStream(clientSocket.getOutputStream());
```

Example: Java client using TCP (cont.)

```
//create input stream attached to socket
                    BufferedReader inFromServer = new BufferedReader(new
                              InputStreamReader (clientSocket.getInputStream()));

                    sentence = inFromUser.readLine();

                    //send line to server
                    outToServer.writeBytes(sentence + '\n');

                    //read line from server
                    modifiedSentence = inFromServer.readLine();

                    System.out.println("FROM SERVER: " + modifiedSentence);

                    clientSocket.close();
          }
}
```



```
icrosoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>c:

C:\>cd Program Files/Java/jdk1.5.0/bin

C:\Program Files\Java\jdk1.5.0\bin>java TCPServer
```

```
icrosoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>c:

C:\>cd Program Files/Java/jdk1.5.0/bin

C:\Program Files\Java\jdk1.5.0\bin>java TCPClient
This is a message from the first client!!
FROM SERVER: THIS IS A MESSAGE FROM THE FIRST CLIENT!!

C:\Program Files\Java\jdk1.5.0\bin>java TCPClient
This is a message from the second client!!!!
FROM SERVER: THIS IS A MESSAGE FROM THE SECOND CLIENT!!!!

C:\Program Files\Java\jdk1.5.0\bin>java TCPClient
This is a message from the third client - Good bye.
FROM SERVER: THIS IS A MESSAGE FROM THE THIRD CLIENT - GOOD BYE.

C:\Program Files\Java\jdk1.5.0\bin>
```

Start TCP Server executing

Start a TCP Client executing and send message to server.

Server responds and client process terminates. The server is still executing.

Another client begins execution and the cycle repeats.

A More Sophisticated TCP Client/Server Example Using GUIs

- Over the next few pages you will find the Java code for a more sophisticated client/server example.
- This example utilizes a GUI and makes things a bit more interesting from the programming point of view.
- Server process appears on pages 25-32. Server test process appears on page 41.
- Client process appears on pages 33-40. Client test process appears on page 42.

Sample Code: Java server using TCP with GUI

Page 1: Server

```
// TCPServerGUI.java
// Set up a TCP Server that will receive a connection from a client, send
// a string to the client, and close the connection.  GUI Version
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class TCPServerGUI extends JFrame
{
   private JTextField enterField; // inputs message from user
   private JTextArea displayArea; // display information to user
   private ObjectOutputStream output; // output stream to client
   private ObjectInputStream input; // input stream from client
private ServerSocket server; // server socket
private Socket connection; // connection to client
private int counter = 1; // counter of number of connections

// set up GUI
public TCPServerGUI()
{
   super( “TCP Server" );

   enterField = new JTextField(); // create enterField
   enterField.setEditable( false );
   enterField.addActionListener(
      new ActionListener()
      {
         // send message to client
         public void actionPerformed( ActionEvent event )
         {
            sendData( event.getActionCommand() );
            enterField.setText( "" );
         } // end method actionPerformed
      } // end anonymous inner class
   ); // end call to addActionListener

   add( enterField, BorderLayout.NORTH );
```

Page 2: Server

```
   displayArea = new JTextArea(); // create displayArea
   add( new JScrollPane( displayArea ), BorderLayout.CENTER );

   setSize( 300, 150 ); // set size of window
   setVisible( true ); // show window
} // end Server constructor

// set up and run server
public void runServer()
{
   try // set up server to receive connections; process connections
   {
      server = new ServerSocket( 12345, 100 ); // create ServerSocket

      while ( true )
      {
         try
         {
            waitForConnection(); // wait for a connection
            getStreams(); // get input & output streams
            processConnection(); // process connection
         } // end try
         catch ( EOFException eofException )
         {
```

Page 3: Server

```
            displayMessage( "\nServer terminated connection" );
         } // end catch
         finally
         {
            closeConnection(); //  close connection
            counter++;
         } // end finally
      } // end while
   } // end try
   catch ( IOException ioException )
   {
      ioException.printStackTrace();
   } // end catch
} // end method runServer

// wait for connection to arrive, then display connection info
private void waitForConnection() throws IOException
{
   displayMessage( "Waiting for connection\n" );
   connection = server.accept(); // allow server to accept connection
   displayMessage( "Connection " + counter + " received from: " +
      connection.getInetAddress().getHostName() );
} // end method waitForConnection
```

Page 4: Server

```
 // get streams to send and receive data
private void getStreams() throws IOException
{
   // set up output stream for objects
   output = new ObjectOutputStream( connection.getOutputStream() );
   output.flush(); // flush output buffer to send header information

   // set up input stream for objects
   input = new ObjectInputStream( connection.getInputStream() );

   displayMessage( "\nGot I/O streams\n" );
} // end method getStreams

// process connection with client
private void processConnection() throws IOException
{
   String message = "Connection successful";
   sendData( message ); // send connection successful message

   // enable enterField so server user can send messages
   setTextFieldEditable( true );
```

Page 5: Server

```
      do // process messages sent from client
      {
         try // read message and display it
         {
            message = ( String ) input.readObject(); // read new message
            displayMessage( "\n" + message ); // display message
         } // end try
         catch ( ClassNotFoundException classNotFoundException )
         {
            displayMessage( "\nUnknown object type received" );
         } // end catch

      } while ( !message.equals( "CLIENT>>> TERMINATE" ) );
   } // end method processConnection

   // close streams and socket
   private void closeConnection()
   {
      displayMessage( "\nTerminating connection\n" );
      setTextFieldEditable( false ); // disable enterField
      try
      {
         output.close(); // close output stream
         input.close(); // close input stream
         connection.close(); // close socket
      } // end try
```

Page 6: Server

Page 7: Server

```
catch ( IOException ioException )
      {
         ioException.printStackTrace();
      } // end catch
   } // end method closeConnection

   // send message to client
   private void sendData( String message )
   {
      try // send object to client
      {
         output.writeObject( "SERVER>>> " + message );
         output.flush(); // flush output to client
         displayMessage( "\nSERVER>>> " + message );
      } // end try
      catch ( IOException ioException )
      {
         displayArea.append( "\nError writing object" );
      } // end catch
   } // end method sendData

   // manipulates displayArea in the event-dispatch thread
   private void displayMessage( final String messageToDisplay )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // updates displayArea
            {
               displayArea.append( messageToDisplay ); // append message
            } // end method run
         } // end anonymous inner class
      ); // end call to SwingUtilities.invokeLater
   } // end method displayMessage

   // manipulates enterField in the event-dispatch thread
   private void setTextFieldEditable( final boolean editable )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // sets enterField's editability
            {
               enterField.setEditable( editable );
            } // end method run
         }  // end inner class
      ); // end call to SwingUtilities.invokeLater
   } // end method setTextFieldEditable
} // end class TCPServerGUI
```

Page 8: Server

Sample Code: Java client using TCP with GUI

Page 1: Client

```
// TCPClientGUI.java
// Client that reads and displays information sent from a Server.
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class TCPClientGUI extends JFrame
{
   private JTextField enterField; // enters information from user
   private JTextArea displayArea; // display information to user
   private ObjectOutputStream output; // output stream to server
   private ObjectInputStream input; // input stream from server
   private String message = ""; // message from server
   private String chatServer; // host server for this application
```

Page 2: Client

```
private Socket client; // socket to communicate with server

// initialize chatServer and set up GUI
public TCPClientGUI( String host )
{
   super( “TCP Client" );

   chatServer = host; // set server to which this client connects

   enterField = new JTextField(); // create enterField
   enterField.setEditable( false );
   enterField.addActionListener(
      new ActionListener()
      {
         // send message to server
         public void actionPerformed( ActionEvent event )
         {
            sendData( event.getActionCommand() );
            enterField.setText( "" );
         } // end method actionPerformed
      } // end anonymous inner class
   ); // end call to addActionListener

   add( enterField, BorderLayout.NORTH );
```

Page 3: Client

```
      displayArea = new JTextArea(); // create displayArea
      add( new JScrollPane( displayArea ), BorderLayout.CENTER );

      setSize( 300, 150 ); // set size of window
      setVisible( true ); // show window
   } // end Client constructor

   // connect to server and process messages from server
   public void runClient()
   {
      try // connect to server, get streams, process connection
      {
         connectToServer(); // create a Socket to make connection
         getStreams(); // get the input and output streams
         processConnection(); // process connection
      } // end try
      catch ( EOFException eofException )
      {
         displayMessage( "\nClient terminated connection" );
      } // end catch
      catch ( IOException ioException )
      {
         ioException.printStackTrace();
      } // end catch
      finally
      {
         closeConnection(); // close connection
      } // end finally
   } // end method runClient

   // connect to server
   private void connectToServer() throws IOException
   {
      displayMessage( "Attempting connection\n" );

      // create Socket to make connection to server
      client = new Socket( InetAddress.getByName( chatServer ), 12345 );

      // display connection information
      displayMessage( "Connected to: " +
         client.getInetAddress().getHostName() );
   } // end method connectToServer

   // get streams to send and receive data
   private void getStreams() throws IOException
   {
      // set up output stream for objects
      output = new ObjectOutputStream( client.getOutputStream() );
      output.flush(); // flush output buffer to send header information
```

Page 4: Client

```
      // set up input stream for objects
      input = new ObjectInputStream( client.getInputStream() );

      displayMessage( "\nGot I/O streams\n" );
   } // end method getStreams

   // process connection with server
   private void processConnection() throws IOException
   {
      // enable enterField so client user can send messages
      setTextFieldEditable( true );

      do // process messages sent from server
      {
         try // read message and display it
         {
            message = ( String ) input.readObject(); // read new message
            displayMessage( "\n" + message ); // display message
         } // end try
         catch ( ClassNotFoundException classNotFoundException )
         {
            displayMessage( "\nUnknown object type received" );
         } // end catch

      } while ( !message.equals( "SERVER>>> TERMINATE" ) );
   } // end method processConnection
```

Page 5: Client

```
// close streams and socket
private void closeConnection()
{
   displayMessage( "\nClosing connection" );
   setTextFieldEditable( false ); // disable enterField

   try
   {
      output.close(); // close output stream
      input.close(); // close input stream
      client.close(); // close socket
   } // end try
   catch ( IOException ioException )
   {
      ioException.printStackTrace();
   } // end catch
} // end method closeConnection

// send message to server
private void sendData( String message )
{
   try // send object to server
   {
      output.writeObject( "CLIENT>>> " + message );
      output.flush(); // flush data to output
      displayMessage( "\nCLIENT>>> " + message );
   } // end try
```

Page 6: Client

Page 7: Client

```
  catch ( IOException ioException )
   {
      displayArea.append( "\nError writing object" );
   } // end catch
} // end method sendData

// manipulates displayArea in the event-dispatch thread
private void displayMessage( final String messageToDisplay )
{
   SwingUtilities.invokeLater(
      new Runnable()
      {
         public void run() // updates displayArea
         {
            displayArea.append( messageToDisplay );
         } // end method run
      }  // end anonymous inner class
   ); // end call to SwingUtilities.invokeLater
} // end method displayMessage
```

Page 8: Client

```
   // manipulates enterField in the event-dispatch thread
   private void setTextFieldEditable( final boolean editable )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // sets enterField's editability
            {
               enterField.setEditable( editable );
            } // end method run
         } // end anonymous inner class
      ); // end call to SwingUtilities.invokeLater
   } // end method setTextFieldEditable
} // end class TCPClientGUI
```

Sample Code: Java server test

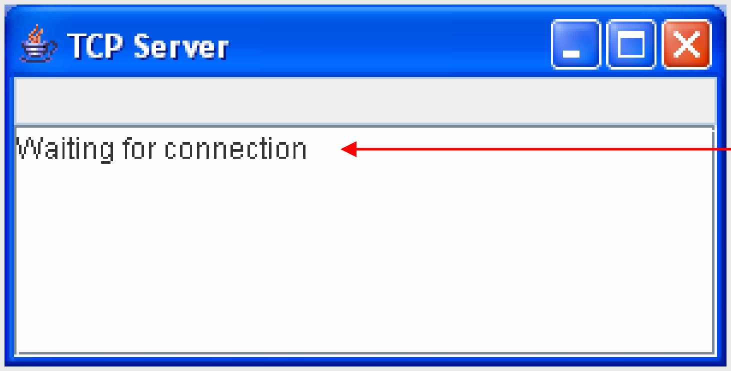
```
// TCPServerTest.java
// Test the TCPServerGUI application.  GUI Version
import javax.swing.JFrame;

public class TCPServerTest
{
   public static void main( String args[] )
   {
      TCPServerGUI application = new TCPServerGUI(); // create server
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.runServer(); // run server application
   } // end main
} // end class TCPServerTest
```

Sample Code: Java client test

```
// TCPClientTest.java
// Test the TCPClientGUI class.  GUI Version
import javax.swing.JFrame;

public class TCPClientTest
{
   public static void main( String args[] )
   {
      TCPClientGUI application; // declare client application

      // if no command line args
      if ( args.length == 0 )
         application = new TCPClientGUI( "127.0.0.1" ); // connect to localhost
      else
         application = new TCPClientGUI( args[ 0 ] ); // use args to connect

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.runClient(); // run client application
   } // end main
} // end class TCPClientTest
```

Special IP address to designate localhost.

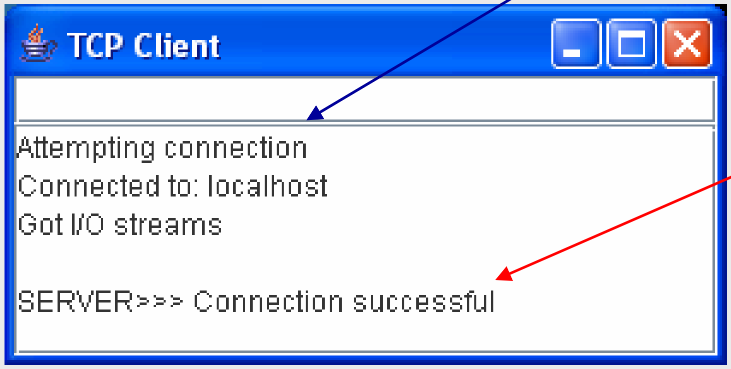
Sample Screen Shots Illustrating Client/Server Processes



```
Waiting for connection
```

Server process initialized and waiting for a client connection.

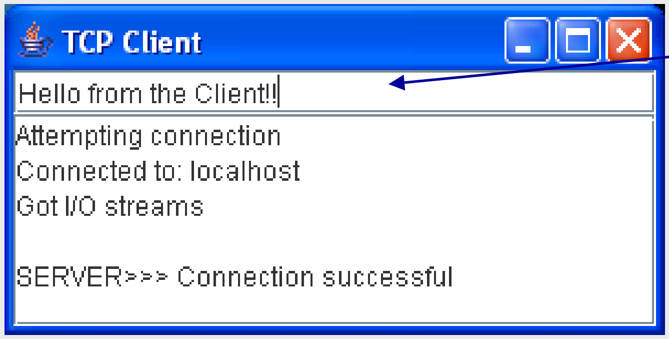
Client process attempts connection to localhost.



```
Attempting connection
Connected to: localhost
Got I/O streams

SERVER>>> Connection successful
```

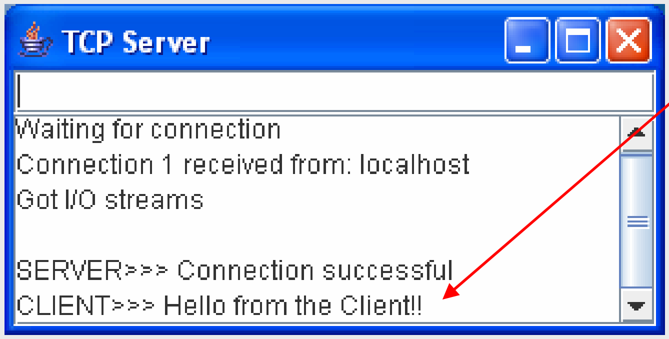
Server responds. Connection to server on localhost is successful. Stream connection is now established between server and client.

Sample Screen Shots Illustrating Client/Server Processes (cont.)



```
TCP Client
Hello from the Client!!
Attempting connection
Connected to: localhost
Got I/O streams

SERVER>>> Connection successful
```

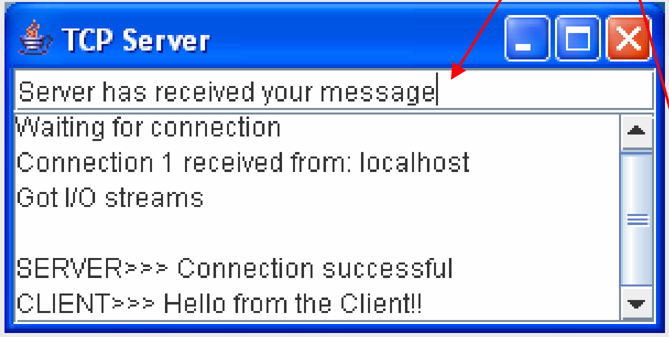
Client sends a message to the server.



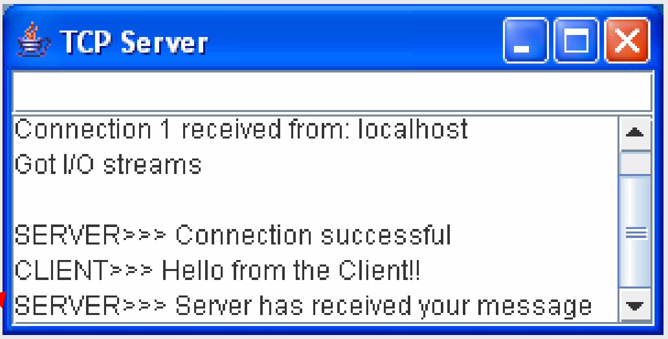
```
TCP Server

Waiting for connection
Connection 1 received from: localhost
Got I/O streams

SERVER>>> Connection successful
CLIENT>>> Hello from the Client!!
```

Server message from the client process.

Server responds to client.

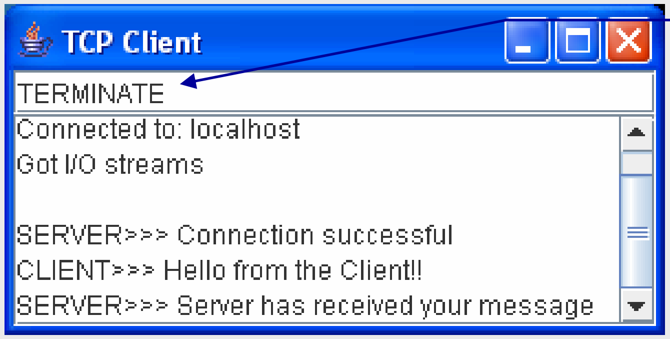


```
TCP Server
Server has received your message
Waiting for connection
Connection 1 received from: localhost
Got I/O streams

SERVER>>> Connection successful
CLIENT>>> Hello from the Client!!
```



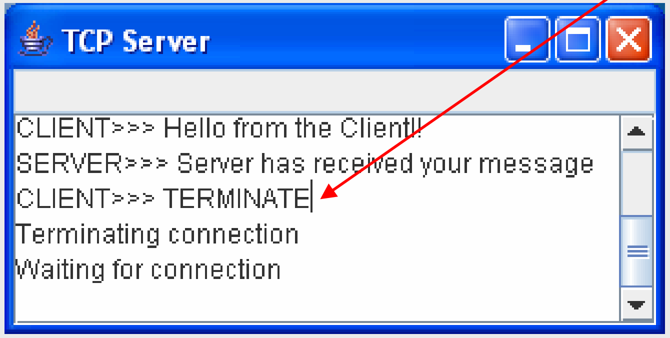
```
TCP Server

Connection 1 received from: localhost
Got I/O streams

SERVER>>> Connection successful
CLIENT>>> Hello from the Client!!
SERVER>>> Server has received your message
```

Sample Screen Shots Illustrating Client/Server Processes (cont.)



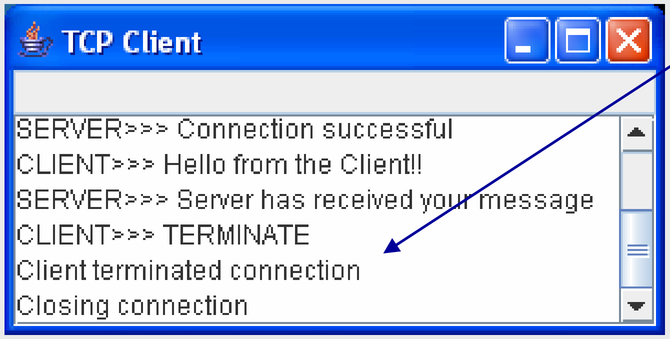
```
TERMINATE
Connected to: localhost
Got I/O streams

SERVER>>> Connection successful
CLIENT>>> Hello from the Client!!
SERVER>>> Server has received your message
```

Client issues message to terminate connection.



```
CLIENT>>> Hello from the Client!!
SERVER>>> Server has received your message
CLIENT>>> TERMINATE
Terminating connection
Waiting for connection
```

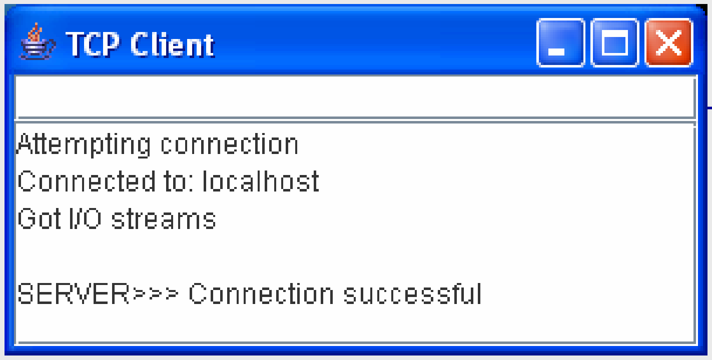
Server receives request from Client to terminate connection. Server responds by terminating connection and then blocking to await a subsequent connection.



```
SERVER>>> Connection successful
CLIENT>>> Hello from the Client!!
SERVER>>> Server has received your message
CLIENT>>> TERMINATE
Client terminated connection
Closing connection
```

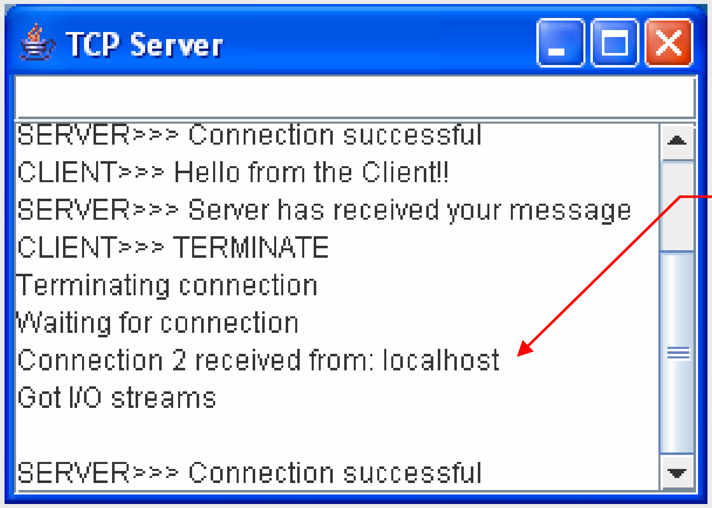
Message from Server that Client terminated connection and that the connection is now closed.

Sample Screen Shots Illustrating Client/Server Processes (cont.)



```
Attempting connection
Connected to: localhost
Got I/O streams

SERVER>>> Connection successful
```

A subsequent connection request from another Client process is accepted by the Server. Server indicates that this is the second connection received from a client.



```
SERVER>>> Connection successful
CLIENT>>> Hello from the Client!!
SERVER>>> Server has received your message
CLIENT>>> TERMINATE
Terminating connection
Waiting for connection
Connection 2 received from: localhost
Got I/O streams

SERVER>>> Connection successful
```

Server accepts a second connection and is now connected to the second client process.