# Client/Server Computing

Sockets



## Peer-to-peer Communication

- Most early Client/Server applications were implemented using low-level, conversational, peer-to-peer protocols
  - such as sockets, NetBios and NamedPipes
- These low level protocols are hard to code and maintain
  - replaced now with protocols providing higher level of abstraction like RPC and ORBs

# Peer-to-peer Communication

- In client-server applications
  - the server provides some services to its client
  - The client asks for a service and get back the result
- The communication that occurs between the client and the server must be reliable
  - no data can be dropped
  - data must arrive on the client side in the same order in which the server sent it

3

# Peer-to-peer Communication

- TCP provides a reliable, point-to-point communication channel that client/server applications on the Internet use to communicate with each other
- To communicate over TCP, a client program and a server program establish a connection to one another
- Each program binds a socket to its end of the connection
- To communicate, the client and the server each reads from and writes to the socket bound to the connection
- Sockets are supported on every operating system

4

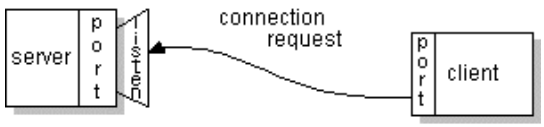
## What's a socket?

- **Definition: A** *socket* is one endpoint of a two-way communication link between two programs running on the network
- A socket is bound to a port number so that the TCP layer can identify the application that data is destined to

5

## What's a socket?

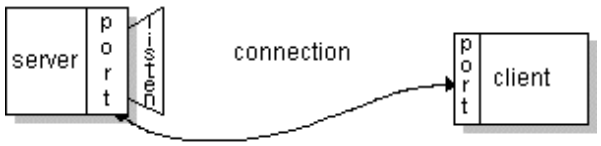
- Normally, a server runs on a specific computer and has a socket that is bound to a specific port number
- The server just waits, listening to the socket for a client to make a connection request.



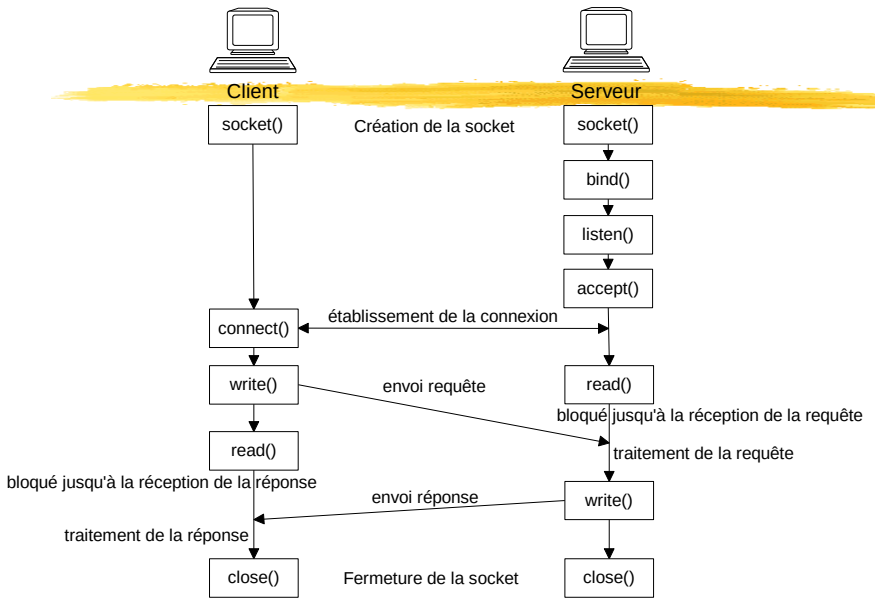

6

# What's a socket?

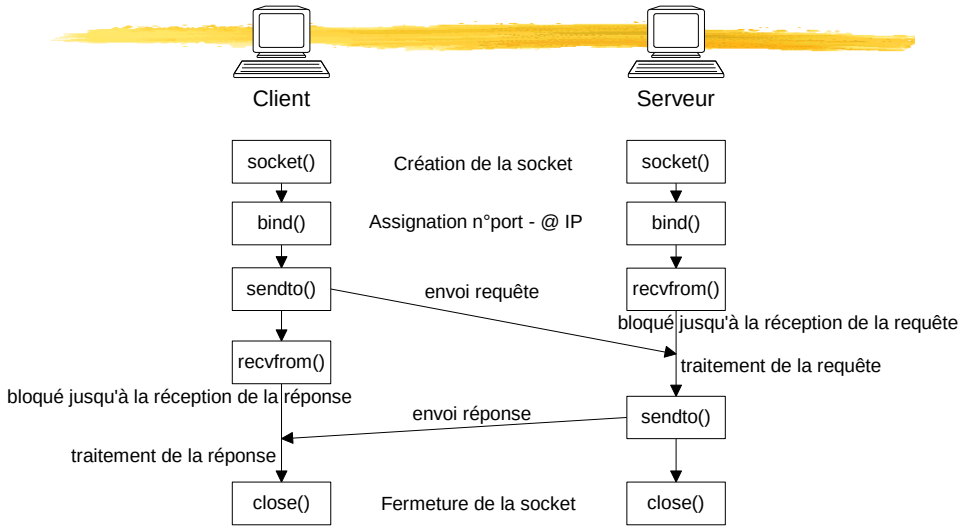
- On the client-side, The client knows the hostname of the machine on which the server is running and the port number to which the server is connected
- To make a connection request, the client tries to rendez vous with the server on the server's machine and port
- If everything goes well, the server accepts the connection




7




8




9

- Socket: telephone
- Bind: assign telephone number to a telephone
- Listen: turn on the ringer so that you can hear the phone call
- Connect: dial a phone number
- Accept: answer the phone
- Read/write: talking

10

# What's a socket?

- Upon acceptance, the server gets a new socket bound to the same port
  - It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client
- On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server
- The client and server can now communicate by writing to or reading from their sockets

11

# L'interface SOCKET

**exemple, TCP**

Série

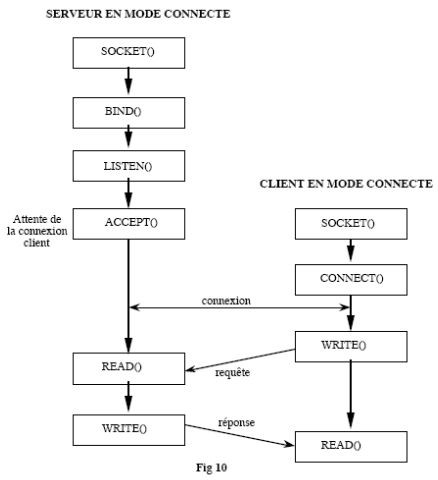
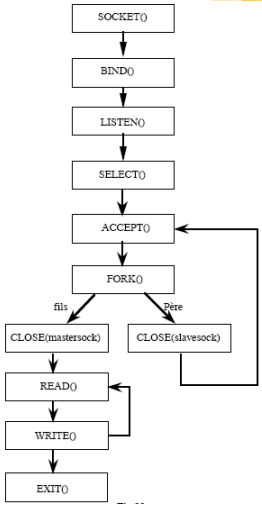



Fig 10

Parallèle

Serveur en mode connecté

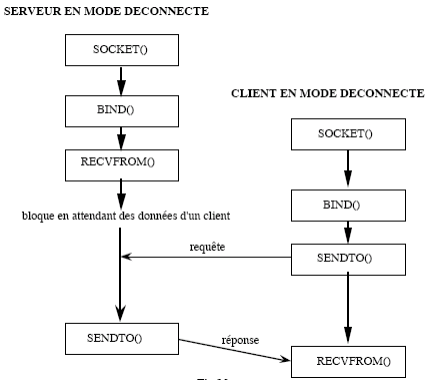



12

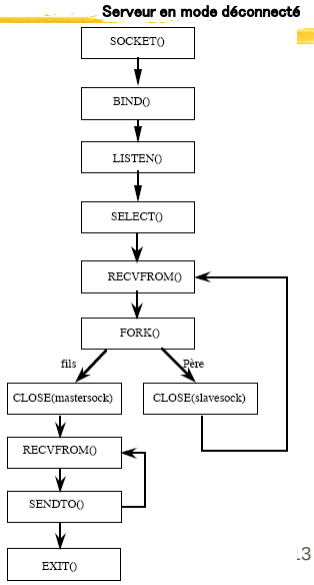
# L'interface SOCKET

**exemple, UDP**

série




Parallèle




.3

# Java and Sockets

- The java.net package in the Java platform provides a class Socket, that implements one side of a two-way connection between your Java program and another program on the network
- The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program

14

## Java and Sockets

- By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion
- java.net includes also the ServerSocket class, which implements a socket that servers can use to listen for and accept connections to clients

15

## Reading from and Writing to a Socket

- Example:
  - The example program implements a client, EchoClient, that connects to the Echo server
  - The Echo server simply receives data from its client and echoes it back
  - EchoClient reads input from the user, sends it to the Echo server, gets a response from the server, and display it, until it reaches the end of input

16

## Reading from and Writing to a Socket

- The basics are always the same
  - Open a socket
  - Open an input stream and output stream to the socket
  - Read from and write to the stream according to the adopted protocol between the client and the server
  - Close the streams
  - Close the socket

17

## Creating a Server in Java (i)

- When the server *first* starts up, it creates a server socket (`java.net.ServerSocket`) on a given *port*:

```
ServerSocket serverSocket = new ServerSocket(port);
```

- Port numbers typically range from 0 to 65535
  - First 1024 port numbers are reserved for privileged services (e.g. HTTP uses port 80, FTP uses port 21)

```
ServerSocket serverSocket = new ServerSocket(8000);
```

- Using a port in use throws a `BindException`

18

# Creating a Server in Java (ii)

- Once a server socket is created, the `accept()` method is called to activate the server socket as a *listener* ready to accept new incoming client connections

```
Socket incomingSocket = serverSocket.accept();
```

- Java supports both TCP and UDP transmission protocols
  - For UDP, use `java.net.DatagramSocket` and `java.net.DatagramPacket` classes

19

# Creating a Client in Java

- A client connects to a listening server via the `Socket()` constructor:

```
Socket clientSocket = new Socket(serverName, port);

Socket s1 = new Socket("java.rpi.edu", 8123);
Socket s2 = new Socket("128.3.127.5", 8000);
Socket s3 = new Socket("127.0.0.1", 9000);
Socket s4 = new Socket("localhost", 9001);
```

- Constructor could throw `UnknownHostException`

20

# InetAddress Class

- The Inetaddress class provides you with a limited interface to DNS for doing both forward and reverse internet address lookups
  - An InetAddress class method corresponds to a DNS request
- No public constructor
- Three static methods:
  - InetAddress getByName(String)
    - Static method used to retrieve the address for the host name passed as the parameter.
  - InetAddress [ ] getAllByName(String)
    - Static method used to retrieve all the addresses for the host name passed as a parameter.
  - InetAddress getLocalHost( )
    - Static method used to retrieve the address for the current, or local, host.

21

# InetAddress Class

- Three additional "getter" methods
  - String getHostName( )
    - Returns the host name.
  - byte[ ] getAddress( )
    - Returns the IP address.
  - String getHostAddress( )
    - Returns the IP address as a string.

22

# InetAddress Examples

```
try
{   InetAddress fullname =
InetAddress.getByName(“bigyellowcat.cs.binghamton.edu");
    InetAddress alias = InetAddress.getByName(“bigyellowcat");
    InetAddress octets = InetAddress.getByName(“128.226.121.44");
    if (fullname.equals(alias) && fullname.equals(octets))
        // All is right with the world! }
catch (UnknownHostException e)
    { // Exception handling here. }
```



# Retrieving the current machine’s address

```
import java.net.*;

public class WhoAmI {

  public static void main (String[] args) {
    try {
      InetAddress a = InetAddress.getLocalHost();
      System.out.println(a.getHostName() + " / " +
                         a.getHostAdress());
    }
    catch (UnknownHostException e) {
      System.out.println
        ("No Access to my Address");
    }
  }
}
```

# Communicating via Sockets (i)

- Once established, a socket contains both an input and an output stream
  - Methods `getInputStream()` and `getOutputStream()` return `InputStream` and `OutputStream` objects (from the `java.io` library)
  - Interesting and useful subclasses of `InputStream` and `OutputStream` include:
    - `DataInputStream` and `DataOutputStream`
    - `ObjectInputStream` and `ObjectOutputStream`

25

# Communicating via Sockets (ii)

- Use `DataInputStream` and `DataOutputStream` to transfer primitive data types between client and server

```
InputStream input = socket.getInputStream();
DataInputStream in = new DataInputStream(input);

OutputStream output = socket.getOutputStream();
DataOutputStream out = new DataOutputStream(output);
```

26

# The EchoClient Example

```
import java.io.*;
import java.net.*;
class EchoClient {

    public static void main(String argv[]) throws Exception
    {
      String Message;
       Socket cs= new Socket("localhost", 6789);
      String M;
      BufferedReader toServer= new BufferedReader(new
      InputStreamReader(System.in));
       Message=toServer.readLine();

       DataOutputStream toS= new
      DataOutputStream(cs.getOutputStream());
      toS.writeBytes(Message+'\n');

      BufferedReader fromServer= new    BufferedReader(new
      InputStreamReader(cs.getInputStream()));
      M=fromServer.readLine();
      System.out.println("from server:"+M);
}
cs.close();
    }
}
```

27

# Writing the Server Side of a Socket

```
import java.io.*;
import java.net.*;
class EchoServer {
  public static void main(String argv[]) throws Exception
    {
     ServerSocket ss= new ServerSocket(6789);
     Socket client= ss.accept();

     BufferedReader fromClient = new BufferedReader(new
       InputStreamReader(client.getInputStream()));

     String Message= fromClient.readLine();
     System.out.println("from client: "+ Message);
     DataOutputStream toClient= new
     DataOutputStream(client.getOutputStream());
     toClient.writeBytes(Message.toUpperCase()+'\n');
        }
    }
```

28

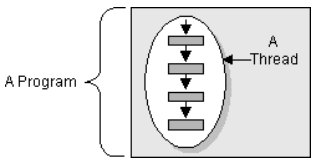
# Running the Programs

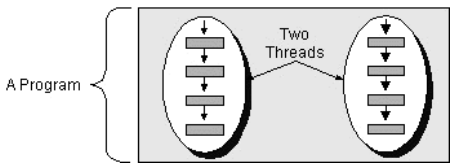
- You must start the server program first
- Next, run the client program

29

# Threads

- Definition: Thread is a single Sequential Flow of Control within a program.
- Other Names: Thread = Execution Context = Lightweight Process
- Thread like a Sequential Program, has
  - A beginning, a sequence, and an end.
  - Has a single point of execution, at any given time




30

# Threads

- Thread is similar to a Real Process, but runs in a program and uses its resources, and program's environment.
- It must have some private resources like Execution Stack and Program Counter.

31

# Multi-Threading

- Motivation for Multi-Threading
  - User-interface
    - User can give requests while Something is running
  - Optimize throughput
    - When some process is stuck in some resource like I/O waiting for a response, others can use the CPU
  - Multi-Processor environment
    - To utilize all the processors (Like in a web server)

32

# Creating Threads In Java

- To create new threads in java we have
  - Runnable interface
  - Thread class
- Every object that will run something in a different Thread, must implement "Runnable" interface (directly or indirectly) .

```
interface Runnable
{
        void run();
}
```

33

# Creating Threads In Java

- Thread class implements Runnable
- Thread.start() method: immediately returns and starts to execute the "run()" method in a different thread.
- It runs either its own run() method or its internal Runnable instance which is passed in the constructor.

34

# Provide a Runnable object

- The Runnable interface defines a single method, run, meant to contain the code executed in the thread
- The Runnable object is passed to the Thread constructor, as the following HelloRunnable example

```
public class HelloRunnable implements Runnable {
public void run() {
    System.out.println("Hello from a thread!"); }
public static void main(String args[]) {
    (new Thread(new HelloRunnable())).start(); } }
```

35

# Subclass Thread

- The Thread class itself implements Runnable
  - its run method does nothing
- An application can subclass Thread, providing its own implementation of run, as in the HelloThread example

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!"); }
    public static void main(String args[]) {
        (new HelloThread()).start(); }
}
```

36

# Supporting Multiple Clients

- Client connection requests are queued at the port, so the server must accept the connections sequentially
- However, the server can service them simultaneously through the use of threads
  - one thread per each client connection. The basic flow of logic in such a server is this:
    - while (true) {
      - accept a connection ;
      - create a thread to deal with the client ;
      - end while
  - The thread reads from and writes to the client connection as necessary.

37

# Serving Multiple Clients

- So far, we've only looked at a server that blocks when it receives an incoming client connection
  - To serve multiple clients, a server delegates each incoming client connection to a dedicated *thread*

```
while (true) {
  Socket socket = serverSocket.accept();
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

38

# Supporting Multiple Clients

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

class TCPServerThread extends Thread{
       static int nb =0;
       public void run(){
                 nb++;
                 System.out.println("I am a thread"+nb);}

       public TCPServerThread(Socket s) throws Exception{

       BufferedReader fromClient = new BufferedReader(
       new InputStreamReader(s.getInputStream()));

       String Message= fromClient.readLine();
       System.out.println("from client: "+ Message);
       DataOutputStream toClient= new DataOutputStream(s.getOutputStream());
       toClient.writeBytes(Message.toUpperCase()+'\n');   }

   public static void main(String argv[]) throws Exception
     {
       ServerSocket ss= new ServerSocket(6789);
       while (true){
       Socket client= ss.accept();
       new TCPServerThread(client).start();}//end of while
         }
     }
```

39

# Datagram vs. TCP

- Clients and servers that communicate via a reliable channel, such as a TCP socket, have a dedicated point-to-point channel between themselves
  - To communicate, they establish a connection, transmit the data, and then close the connection
  - All data sent over the channel is received in the same order in which it was sent. This is guaranteed by the channel.
- In contrast, applications that communicate via datagrams send and receive completely independent packets of information
  - These clients and servers do not have and do not need a dedicated point-to-point channel
  - The delivery of datagrams to their destinations is not guaranteed. Nor is the order of their arrival.

40

# UDP Server

- A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed
- In a UDP server, the server binds a datagram socket to a port and receives data on it
- The java.net package contains three classes to help you write Java programs that use datagrams to send and receive packets over the network
  - DatagramSocket, DatagramPacket, and MulticastSocket
  - An application can send and receive DatagramPackets through a DatagramSocket
  - DatagramPackets can be broadcast to multiple recipients all listening to a MulticastSocket

41

# The Java Datagram Socket API

- In Java, two classes are provided for the datagram socket API:
  - the ***DatagramSocket*** class for the sockets.
  - the ***DatagramPacket*** class for the datagram exchanged.
- A process wishing to send or receive data using this API must instantiate a ***DatagramSocket*** object, or a socket in short.
- Each socket is said to be ***bound*** to a UDP port of the machine on which the process is running.

42

# The Java Datagram Socket API

To send a datagram to another process, a sending process:

- creates an object that represents the datagram itself. This object can be created by instantiating a ***DatagramPacket*** object which carries
  1. the payload data as a reference to a byte array, and
  2. the destination address (the host ID and port number to which the receiver's socket is bound).

- issues a call to a ***send*** method in the ***DatagramSocket*** object, specifying a reference to the ***DatagramPacket*** object as an argument.

43

# The Java Datagram Socket API

- In the receiving process, a ***DatagramSocket*** object must also be instantiated and bound to a local port, the port number must agree with that specified in the datagram packet of the sender.

- To receive datagrams sent to the socket, the receiving process creates a ***DatagramPacket*** object which references a byte array and calls a ***receive*** method in its ***DatagramSocket*** object, specifying as argument a reference to the ***DatagramPacket*** object.

44

# UDP Client

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

public class UDPClient {
   public static void main(String args[]) throws Exception
   {
     BufferedReader inFromUser =
       new BufferedReader(new InputStreamReader(System.in));

     DatagramSocket clientSocket = new DatagramSocket();
     InetAddress IPAddress = InetAddress.getByName("localhost");

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

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

45

# UDP Server

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

class UDPServer {
  public static void main(String args[]) throws Exception
    {
      DatagramSocket serverSocket = new DatagramSocket(9876);
      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

            while(true){
         DatagramPacket receivePacket =
          new DatagramPacket(receiveData, receiveData.length);
          serverSocket.receive(receivePacket);
            String sentence = new String(receivePacket.getData());

         InetAddress IPAddress = receivePacket.getAddress();

         int port = receivePacket.getPort();

                  String capitalizedSentence = sentence.toUpperCase();

         sendData = capitalizedSentence.getBytes();

         DatagramPacket sendPacket =
           new DatagramPacket(sendData, sendData.length, IPAddress,
                       port);

         serverSocket.send(sendPacket);

            }//end of while
    }
}
```

46

# UDP Server - Example

- An example consists of two applications:a Client and a Server
  - The server continuously receives datagram packets over a datagram socket
  - Each datagram packet received by the server indicates a client request for a quotation
  - When the server receives a datagram, it replies by sending a datagram packet that contains a one-line "quote of the moment" back to the client
- The client application in this example is fairly simple
  - It sends a single datagram packet to the server indicating that the client would like to receive a quote of the moment
  - The client then waits for the server to send a datagram packet in response
- Two classes implement the server application
  - QuoteServer and QuoteServerThread
- A single class implements the client application
  - QuoteClient

47

# The QuoteServer.java

- The main method simply creates a new QuoteServerThread object and starts it

```
import java.io.*;
public class QuoteServer {
    public static void main(String[] args) throws IOException {
        new QuoteServerThread().start(); }
}
```

- The QuoteServerThread class implements the main logic of the quote server

48

# The QuoteServerThread.java

- the QuoteServerThread creates a DatagramSocket on port 4445 (arbitrarily chosen)
  - This is the DatagramSocket through which the server communicates with all of its clients

```
public QuoteServerThread() throws IOException { this("QuoteServer"); }
public QuoteServerThread(String name) throws IOException {
        super(name);
        socket = new DatagramSocket(4445);
try { in = new BufferedReader( new FileReader("one-liners.txt")); } catch
(FileNotFoundException e)
        System.err.println("Couldn't open quote file. " + "Serving time
instead."); } }
```

- Creates a socket and prepare to read from a file

49

# The QuoteServerThread.java

- The interesting part of the QuoteServerThread is its run method
  - overrides run in the Thread class and provides the implementation for the thread
- The run method contains a while loop that continues as long as there are more quotes in the file
  - During each iteration of the loop, the thread waits for a DatagramPacket to arrive over the DatagramSocket
  - The packet indicates a request from a client
  - In response to the client's request, the QuoteServerThread gets a quote from the file, puts it in a DatagramPacket and sends it over the DatagramSocket to the client that asked for it

50

# The QuoteServerThread.java

- Let's look first at the section that receives the requests from clients

```
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
```

- Now assume that, the server has received a request from a client for a quote
- Now the server must respond

```
String dString = null;
if (in == null) dString = new Date().toString();
else dString = getNextQuote();
buf = dString.getBytes();
InetAddress address = packet.getAddress();
int port = packet.getPort();
packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);
```

51

# The QuoteClient.java

- The QuoteClient class contains one method, the main method for the client application. The top of the main method declares several local variables for its use:

```
int port;
InetAddress address;
DatagramSocket socket = null;
DatagramPacket packet;
byte[] sendBuf = new byte[256];
```

- The QuoteClient application requires one command-line arguments: the name of the machine on which the QuoteServer is running

```
if (args.length != 1) { System.out.println("Usage: java QuoteClient
    <hostname>"); return; }
```

52

# The QuoteClient.java

- Next, the main method creates a DatagramSocket:
  DatagramSocket socket = new DatagramSocket();
- Next, the QuoteClient program sends a request to the server:

```
byte[] buf = new byte[256];
InetAddress address = InetAddress.getByName(args[0]);
DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 4445);
socket.send(packet)
```

- Next, the client gets a response from the server and displays it:

```
packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
String received = new String(packet.getData(), 0, packet.getLength());
System.out.println("Quote of the Moment: " + received);
```

getData() returns a table of byte

53

# How to implement a protocol

- Consider the Server implements a KnockKnock protocol
  - The server will answer the client according to the protocol

```
KnockKnockProtocol kkp = new KnockKnockProtocol();
while ((inputLine = in.readLine()) != null) {
    outputLine = kkp.processInput(inputLine);
    out.println(outputLine);
    if (outputLine.equals("Bye.")) break; }
```

54