Networking

Sockets in Java

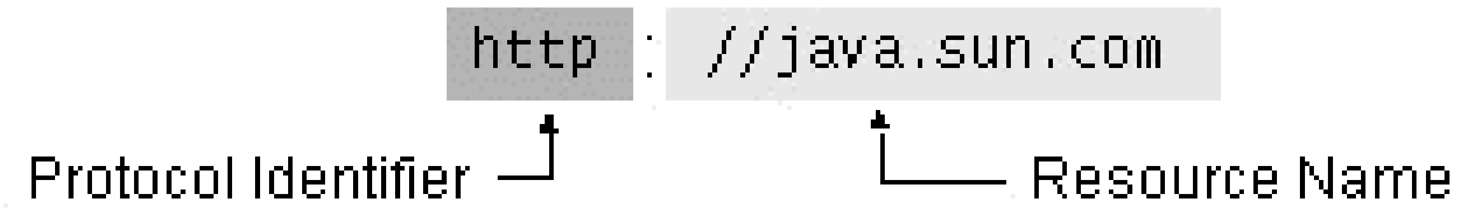
Network Supporting of Java

- Provides the classes for implementing networking applications
- It can be divided into two parts:
  - the **socket classes**, you can communicate with any server on the Internet or implement your own Internet server
  - A number of classes are provided to make it convenient to use **Universal Resource Locators** (URLs) to retrieve data on the Internet

Working with URLs

What Is a URL ?

- **Definition:** URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet



- As in the previous diagram, a URL has two main components:
  - Protocol identifier
  - Resource name

Resource Name

- **Host Name**
  - The name of the machine on which the resource lives.
- **Filename**
  - The pathname to the file on the machine.
- **Port Number**
  - The port number to which to connect (typically optional).
- **Reference**
  - A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

hi-tech college JOHN BRYCE Leading in IT Education a matrix company

Creating a URL

- Absolute URL
  - http://www.gamelan.com
  - `URL gamelan = new URL("http://www.gamelan.com/");`
- Relative URL
  - http://www.gamelan.com/pages/Gamelan.game.html
  - `URL` **`gamelan`** `= new URL("http://www.gamelan.com/pages/");`
  - `URL gamelanGames = new URL(`**`gamelan`**`, "Gamelan.game.html");`
  - **`URL(URL`** ***`baseURL`***`,` **`String`** ***`relativeURL`*****`)`**
- Other URL constructs
  - `URL gamelan = new URL("http", "www.gamelan.com", 80, "pages/Gamelan.network.html");`

MalformedURLException

- Each of the four URL constructors throws a MalformedURLException if the arguments to the constructor refer to a null or unknown protocol.

```
try {
    URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
    . . . //
    exception handler code here . . .
}
```

Parsing a URL

- The URL class provides several methods that let you query URL objects:
  - getProtocol()
    - Returns the protocol identifier component of the URL.
  - getHost()
    - Returns the host name component of the URL.
  - getPort()
    - Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.
  - getFile()
    - Returns the filename component of the URL.
  - getRef()
    - Returns the reference component of the URL
- **Note:** Remember that not all URL addresses contain these components.

Reading Directly from a URL

- After you've successfully created a URL, you can call the URL's `openStream()` method to get a stream from which you can read the contents of the URL.
- The openStream() method returns a java.io.InputStream object

```
import java.net.*;
import java.io.*;
public class URLReader {
  public static void main(String[] args) throws Exception {
     URL yahoo = new URL("http://www.yahoo.com/");
     BufferedReader in = new BufferedReader(
       new InputStreamReader( yahoo.openStream()));
     String inputLine;
     while ((inputLine = in.readLine()) != null)
     System.out.println(inputLine); in.close();
  }
}
```

Setting the Proxy Host

- **UNIX**

```
java -Dhttp.proxyHost=proxyhost [-
 Dhttp.proxyPort=portNumber] URLReader
```

- **DOS shell (Windows 95/NT)**

```
java -Dhttp.proxyHost=proxyhost [-
 Dhttp.proxyPort=portNumber] URLReader
```

Connecting to a URL

- After you've successfully created a URL object, you can call the URL object's `openConnection` method to connect to it

```
try {
      URL yahoo = new URL("http://www.yahoo.com/");    URLConnection
yahooConnection =     yahoo.openConnection();
} catch (MalformedURLException e) {
      // new URL() failed . . .
} catch (IOException e) {
      // openConnection() failed . . .
}
```

- you can use the `URLConnection` object to perform actions such as **reading** from or **writing** to the connection

Reading from a URLConnection

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
  public static void main(String[] args) throws Exception
  {
      URL yahoo = new URL("http://www.yahoo.com/");
      URLConnection yc = yahoo.openConnection();
      BufferedReader in = new BufferedReader(
         new InputStreamReader( yc.getInputStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null)
         System.out.println(inputLine); in.close();
  }
}
```

Writing to a URLConnection

- Many HTML pages contain *forms*-- **text fields** and **other GUI objects** that let you enter ***data to send to the server***.
- After you type in the required information and initiate the query by clicking a button, <u>***your Web browser writes the data to the URL***</u> over the network
- At the other end, a <u>**`cgi-bin` script (usually) *on the server receives the data, processes it***</u>, and then sends you a response, usually in the form of a new HTML page

POST method

- Many cgi-bin scripts use the **POST** METHOD for reading the data from the client
- Thus writing to a URL is often called *posting to a URL*. Server-side scripts use the POST METHOD to read from their standard input.
- **Note:** Some server-side cgi-bin scripts use the GET METHOD to read your data. The POST METHOD is quickly making the GET METHOD obsolete because it's more versatile and has no limitations on the amount of data that can be sent through the connection.

Interaction with cgi-bin scripts

- A Java program can interact with cgi-bin scripts also on the server side by following these steps:

1. Create a URL.
2. Open a connection to the URL.
3. Set output capability on the URLConnection.
4. Get an output stream from the connection. This output stream is connected to the standard input stream of the cgi-bin script on the server.
5. Write to the output stream.
6. Close the output stream.

Writing to a URLConnection - Example

```
import java.io.*;
import java.net.*;
public class Reverse {
  public static void main(String[] args) throws Exception {
       if (args.length != 1) {
              System.err.println("Usage: java Reverse " +
                                     "string_to_reverse");
               System.exit(1);
       }
       String stringToReverse = URLEncoder.encode(args[0]);
       URL url = new URL("http://java.sun.com/cgi-" +
                                 "bin/backwards");
       URLConnection connection = url.openConnection();
       connection.setDoOutput(true);
```

Writing to a URLConnection - Example

```
        PrintWriter out = new PrintWriter(
           connection.getOutputStream());
        out.println("string=" + stringToReverse);
        out.close();

        BufferedReader in = new BufferedReader(
           new InputStreamReader( connection.getInputStream()));
        String inputLine;

        while ((inputLine = in.readLine()) != null)
           System.out.println(inputLine);

        in.close();


    }
  }
```

Sockets

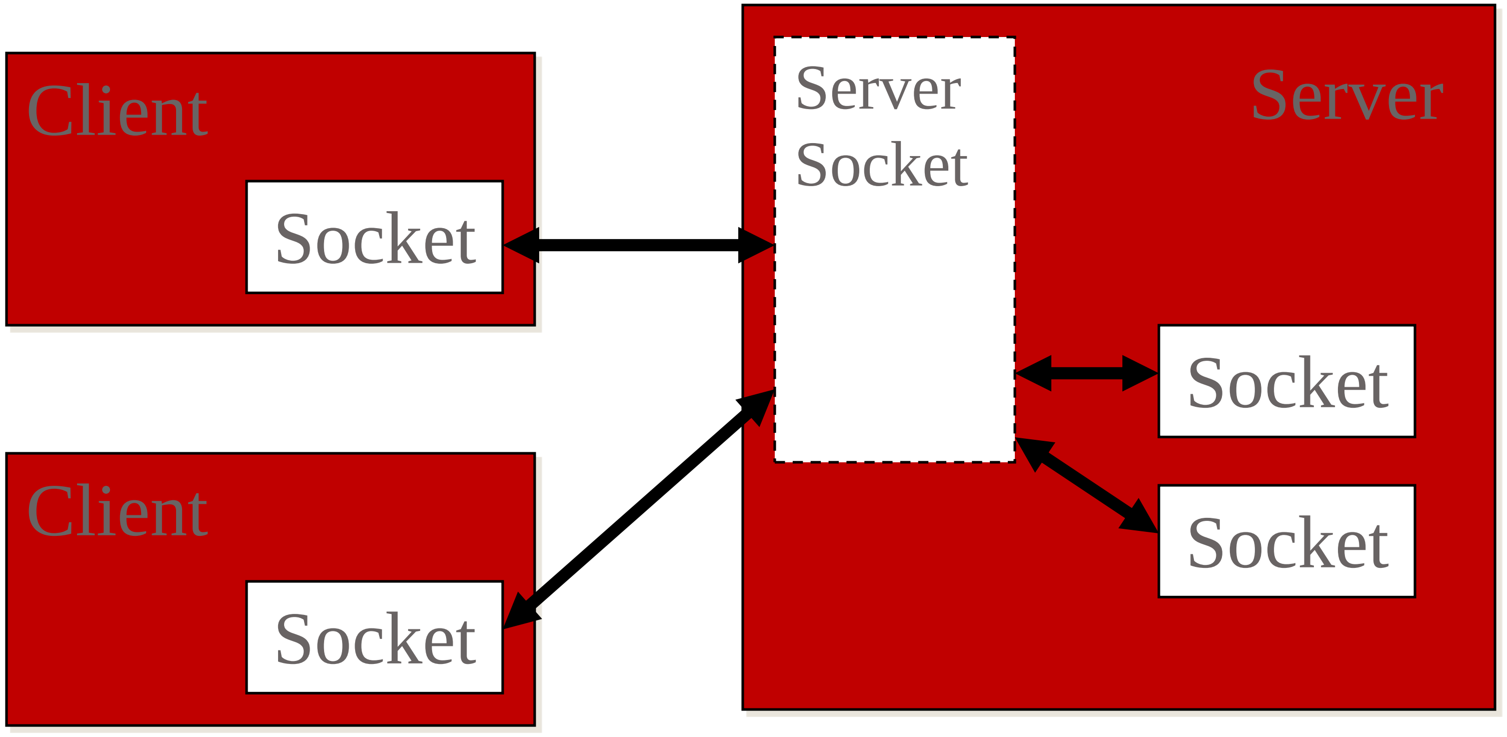
Client-Server Application

- Server
  - provides some service, such as processing database queries or sending out current stock prices
- Client
  - uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor
- The communication that occurs between the client and the server must be **reliable** (TCP)

  That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it

Clients and Servers

- The server application use only one `ServerSocket` object connecting to the clients(with the `accept()` method), but each of client-server connection is implemented by a `Socket` object.

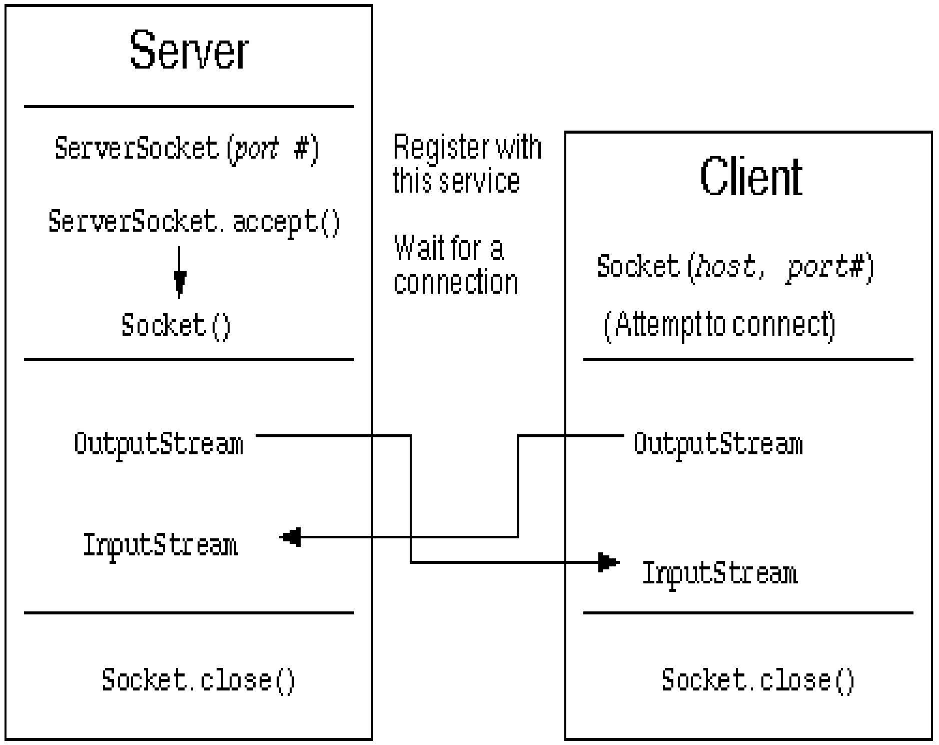


What Is a Socket ?

- A socket is one end-point of a two-way communication link between two programs running on the network.
- Socket classes are used to represent the connection between a client program and a server program.
- The **java.net** package provides two classes--**`Socket`** and **`ServerSocket`**--that implement the client side of the connection and the server side of the connection, respectively

Reading from and Writing to a Socket

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.



Socket I/O

- After a successful connection we can access the streams, which are necessaries to the communication

```
try { Socket server = new Socket(“trantor”, 25);
      InputStream in  = server.getInputStream();
      OutputStream out = server.getOutputStream();
      out.write( 42 );

      PrintStream pout = new PrintStream( out );
      pout.println(“Hello!”);

      Byte back = in.read();
      DataInputStream din = new DataInputStream( in );
      String response = din.readLine();

      server.close();
} catch (IOException e) {}
```

Socket Output

- The server program begins by creating a new ServerSocket object to listen on a specific port (see the statement in bold in the following code segment).

```
try { ServerSocket listener = new ServerSocket( 25 );
      while( !finished ) {
          Socket aClient = listener.accept();
          InputStream in = aClient.getInputStream();
          OutputStream out = aClient.getOutputStream();
          Byte importantByte = in.read();
          DataInputStream din = new DataInputStream( in );
          String request = din.readLine();
          out.write( 43 );
          PrintStream out = new PrintStream( out );
          pout.println( “Viszlát!” );
          aClient.close();
     }
     listener.close();
} catch (IOException e) {}
```

Supporting Multiple Clients

- 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.

Datagrams

What Is a Datagram ?

- **Definition:** A *datagram* is an **independent**, self-contained message sent over the network whose **arrival, arrival time, and content** are **not guaranteed**
- 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.

Writing a Datagram Client and Server

- The example featured in this section 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 QuoteServer Class

- The QuoteServer class, shown here in its entirety, contains a single method: the `main` method for the quote server application. The `main` method simply creates a new `QuoteServerThread` object and starts it:

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

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

The QuoteServerThread Class

- When created, 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.");
  }
}
```

The run method

```
public void run() {

       while (moreQuotes) {
           try {
               byte[] buf = new byte[256];

                   // receive request
               DatagramPacket packet =
                         new DatagramPacket(buf, buf.length);
               socket.receive(packet);

                   // figure out response
               String dString = null;
               if (in == null)
                   dString = new Date().toString();
               else
                   dString = getNextQuote();
               buf = dString.getBytes();
```

The run method

```
                // send the response to the client
                    // at "address" and "port"
                    InetAddress address =
packet.getAddress();
                    int port = packet.getPort();
                    packet =
                         new DatagramPacket(buf,
buf.length,
                                         address,
port);
                    socket.send(packet);
               } catch (IOException e) {
                    e.printStackTrace();
                      moreQuotes = false;
               }
          }
          socket.close();
  }
```

The QuoteClient Class

```
import java.io.*;
import java.net.*;
import java.util.*;
public class QuoteClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
              System.out.println("Usage: java QuoteClient
                                                  <hostname>");
            return;
        }
        // get a datagram socket
        DatagramSocket socket = new DatagramSocket();
        // send request
        byte[] buf = new byte[256];
        InetAddress address = InetAddress.getByName(args[0]);
        DatagramPacket packet =
              new DatagramPacket(buf, buf.length, address, 4445);
        socket.send(packet);
```

The QuoteClient Class

```
            // get response
            packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);

            // display response
            String received = new String(packet.getData());
            System.out.println("Quote of the Moment: " +
      received);

            socket.close();
        }
    }
```

Broadcasting to Multiple Recipients

- **`MulticastSocket`** is used on the client-side to listen for packets that the server broadcasts to multiple clients

```
import java.io.IOException.*;

public class MulticastServer {
    public static void main(String[] args) throws
  IOException {
        new MulticastServerThread().start();
    }
}
public class MulticastServerThread extends
  QuoteServerThread { ...
}
```

MulticastServerThread - run()

```
public void run() {
 while (moreQuotes) {
 try {
      byte[] buf = new byte[256];
      // construct quote
      String dString = null;
      if (in == null)      dString = new Date().toString();
      else   dString = getNextQuote();
      buf = dString.getBytes();
      // send it
      InetAddress group = InetAddress.getByName("230.0.0.1");
      DatagramPacket packet = new DatagramPacket(buf,
             buf.length, group, 4446);
      socket.send(packet);
```

MulticastServerThread - run() [1/2]

```
                // sleep for a while
                try {
                        sleep((long)(Math.random() *
                        FIVE_SECONDS));
                } catch (InterruptedException e) { }
        } catch (IOException e) {
                e.printStackTrace();
                moreQuotes = false;
        }
    }
    socket.close();
  }
```

MulticastClient

```
import java.io.*;
import java.net.*;
import java.util.*;
public class MulticastClient {
  public static void main(String[] args) throws
  IOException {
        MulticastSocket socket = new
  MulticastSocket(4446);
        InetAddress address =
                InetAddress.getByName("230.0.0.1");
        socket.joinGroup(address);
        DatagramPacket packet;
             // get a few quotes
        for (int i = 0; i < 5; i++) {
                byte[] buf = new byte[256];
                packet = new DatagramPacket(buf,
  buf.length);
                socket.receive(packet);
```

MulticastClient

```
    String received = new
      String(packet.getData());
                    System.out.println("Quote of
      the Moment: " +
                 received);
         }
         socket.leaveGroup(address);
         socket.close();
       }
    }
```

Note: Many firewalls and routers are configured not to allow UDP packets. If you have trouble connecting to a service outside your firewall, or if clients have trouble connecting to your service, ask your system administrator if UDP is permitted.

References

- http://java.sun.com
- SUN Educational Services SL-275