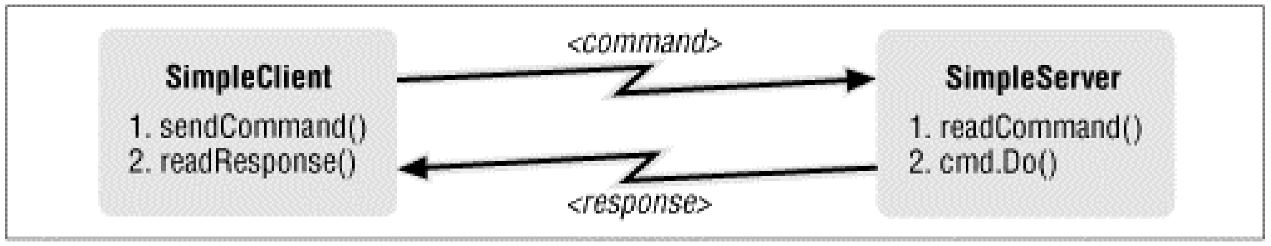
Java

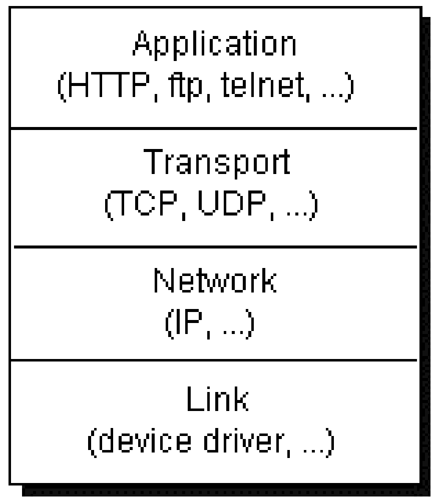
Networking



Presented by Bartosz Sakowicz

Networking Basics

Usually computers running on the Internet communicate to each other using either the Transmission Control Protocol **(TCP)** or the User Datagram Protocol **(UDP):**



When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the java.net package. These classes provide system-independent network communication.

TCP

•When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection.

•TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.

•TCP provides a point-to-point channel for applications that require reliable communications.

•***TCP*** **(*Transmission Control Protocol*) is a connection-based protocol that provides a reliable flow of data between two computers.**

UDP

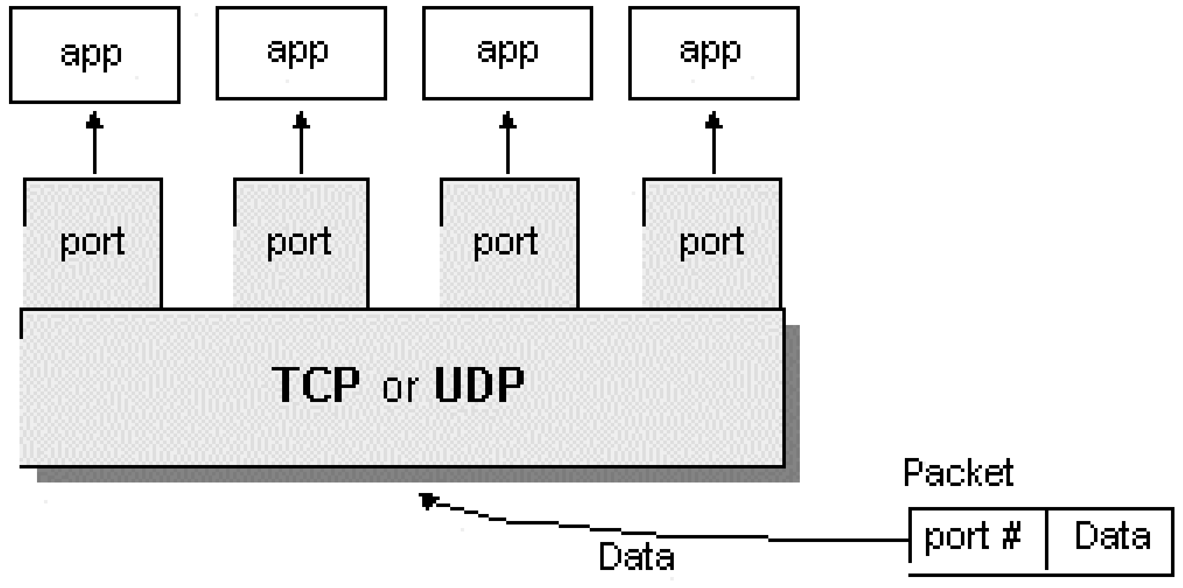
- The UDP protocol provides for communication that is not guaranteed between two applications on the network.
- UDP sends independent packets of data, called **datagrams**, from one application to another.
- Many firewalls and routers have been configured not to allow UDP packets.
- Sometimes reliability provided by TCP is not necessary (eg. Time Server) and you can use UDP which doesn’t have overhead related to establishing connection.

Ports

Generally a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection.

Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined.

Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.



URL

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

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used.

Creating a URL (everything is an object !):

URL DMCS = new URL("http://www.dmcs.p.lodz.pl/");

Relative URL:

URL indexDMCS = new URL(DMCS,"/index.html");

Other constructors:

URL("http", "www.dmcs.p.lodz.pl","/index.html");

URL("http", 80, "www.dmcs.p.lodz.pl","/index.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  . . . }
```

URLs are "write-once" objects. Once you've created a URL object, you cannot change any of its attributes (protocol, host name, filename, or port number).

Reading from URL

```
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); //HTML from yahoo

        in.close();

 } }
```

Connecting to a URL

When you connect to a URL, you are initializing a communication link between your Java program and the URL over the network.

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

Reading from 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(); } }
```

Reading from URL versus reading from URLConnection

- You can use either way to read from a URL.
- Reading from a URLConnection instead of reading directly from a URL might be more useful. This is because you can use the URLConnection object for other tasks (like writing to the URL) at the same time.

Writing to a URLConnection

To write to an URL you have to do following 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 (eg.) script on the server.
5. Write to the output stream.
6. Close the output stream.

Writing to a URLConnection(2)

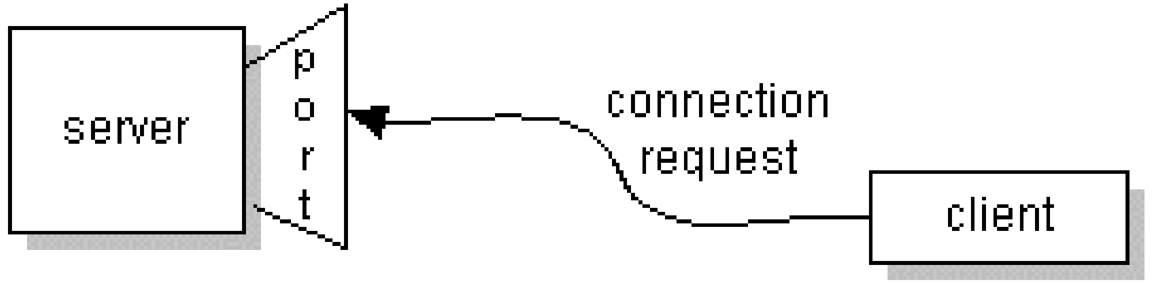
```
...
String stringToReverse = URLEncoder.encode(args[0]);
URL url = new URL("http://java.sun.com/cgi-bin/backwards");
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
PrintWriter out = new PrintWriter( connection.getOutputStream());
out.println("string=" + stringToReverse);
//backward  script waits for data in the above form
out.close();
...
```

Sockets

A **socket** is one endpoint of a two-way communication link between two programs running on the network.

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.

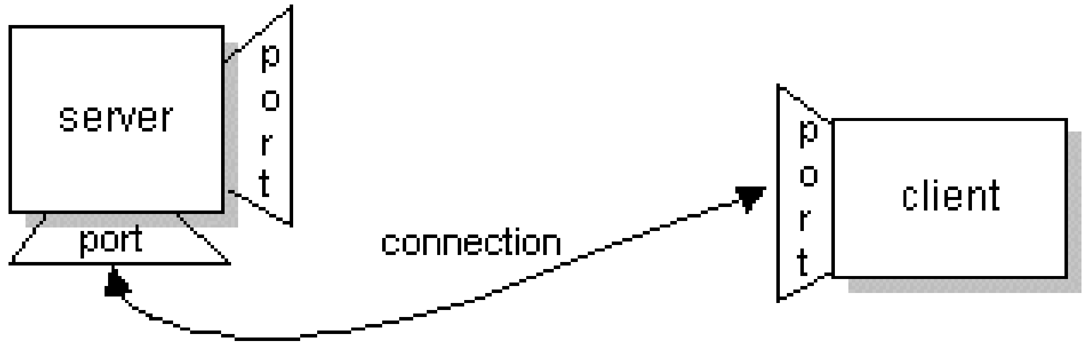
On the client-side: The client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.



Sockets(2)

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) 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.



Reading from and writing to a Socket

To do is you should perform five basic steps:

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.

Reading from and writing to a Socket - example

```
import java.io.*;
import java.net.*; //Socket
public class EchoClient {
public static void main(String[] args)  throws IOException {
    Socket echoSocket = null;
    PrintWriter out = null;
    BufferedReader in = null;
```

Verte→

Reading ... (2)

```
try {
    echoSocket = new Socket(„comp", 7); //Echo Server port
    out = new PrintWriter(echoSocket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(
    echoSocket.getInputStream()));
} catch (UnknownHostException e) {
    System.err.println("Don't know about host: comp.");
    System.exit(1);
 } catch (IOException e) {
    System.err.println("Couldn't get I/O for " + "the connection to:
    comp.");
    System.exit(1);   }
```

Verte→

Reading ... (3)

```
    BufferedReader stdIn = new BufferedReader( new
    InputStreamReader(System.in));

    String userInput;

    while ((userInput = stdIn.readLine()) != null) {
         out.println(userInput);

         System.out.println("echo: " + in.readLine());

    }

    out.close();        in.close();

    stdIn.close();      echoSocket.close();

//The precedence is important. You should first close all streams

// connected to the socket and than socket itself.

} }
```

Reading from URL - socket version

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

String hostname = "http://www.yahoo.com";

int port = 80;

String filename = "/index.html";

Socket s = new Socket(hostname, port);

InputStream sin = s.getInputStream();

BufferedReader fromServer = new BufferedReader(new
InputStreamReader(sin));

OutputStream sout = s.getOutputStream();

PrintWriter toServer = new PrintWriter(new
OutputStreamWriter(sout));
```

verte-->

Reading from URL - socket version(2)

```
toServer.print("GET " + filename + " HTTP/1.0\n\n");
toServer.flush();

 for(String l = null; (l = fromServer.readLine()) != null; )

System.out.println(l);

 toServer.close();

fromServer.close();

s.close();
```

Server Side of Socket

Creating a server socket:

```
try {
    serverSocket = new ServerSocket(4444);
} catch (IOException e) {
    System.out.println("Could not listen on port: 4444");
    System.exit(-1);
}
```

Server Side of Socket(2)

If the server successfully connects to its port, then the ServerSocket object is successfully created and the server continues to the next step - accepting a connection from a client:

```
Socket clientSocket = null;

try {

clientSocket = serverSocket.accept();

} catch (IOException e) {// accept failed}
```

The accept method waits until a client starts up and requests a connection on the host and port of this server. When a connection is requested and successfully established, the accept method returns a new Socket object which is bound to a new port.

Server Side of Socket(3)

After the server successfully establishes a connection with a client, it communicates with the client:

```
PrintWriter out =
    new PrintWriter( clientSocket.getOutputStream(), true);
BufferedReader in =
    new BufferedReader( new InputStreamReader(
clientSocket.getInputStream()));
```

Supporting multiple clients

Server can service clients 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 ;
}
```

Datagrams

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

The java.net package contains classes to help you write Java programs that use datagrams to send and receive packets over the network:

- DatagramSocket
- DatagramPacket

Sending Datagrams

```
...
byte[] buf = new byte[256];
InetAddress address = InetAddress.getByName(„http://...”);
DatagramPacket packet = new DatagramPacket(buf, buf.length,
    address, 4445);
socket.send(packet);
...
```

Receiving and sending Datagrams

```
...
Datagram Socket socket = new DatagramSocket(4445);
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
InetAddress address = packet.getAddress();
int port = packet.getPort();
packet = new DatagramPacket(buf, buf.length, address, port);
  socket.send(packet);
...
```