Java Network Programming

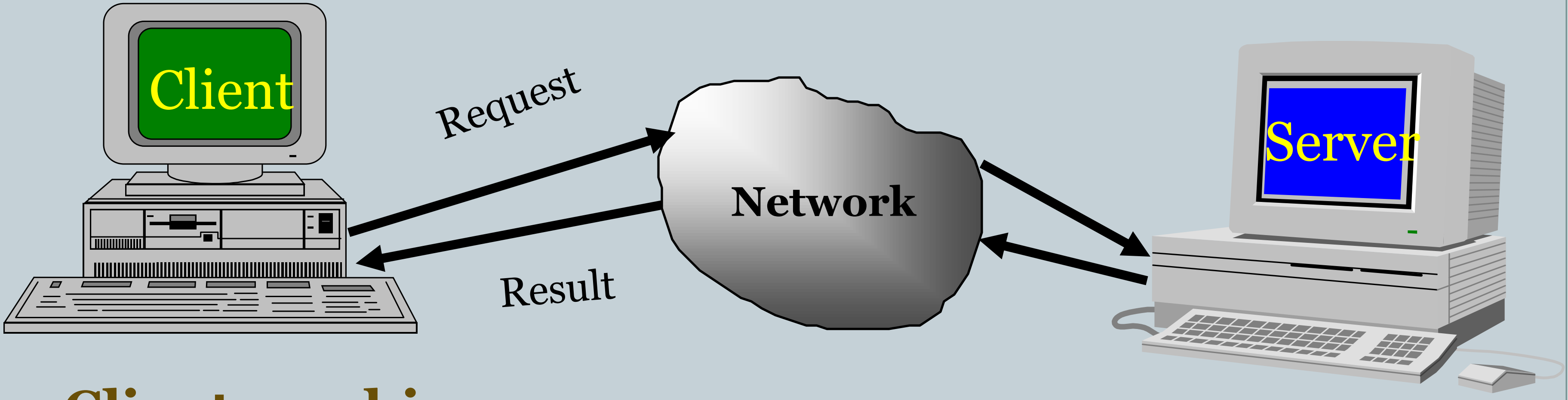
CLIENT/SERVER ARCHITECTURE

Agenda

- Networking Basics
  - TCP, UDP, Ports, DNS, Client-Server Model
- Sockets
- Datagrams
- URL

Elements of Client-Server Computing

- a client, a server, and network

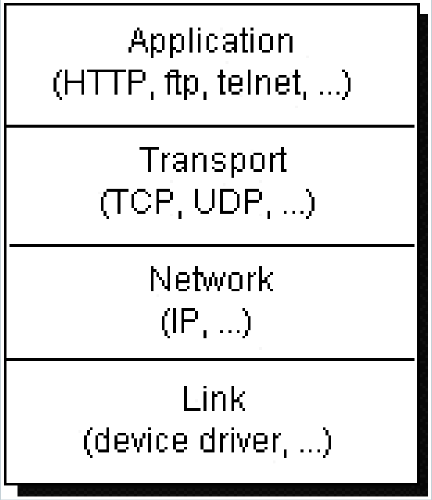


Client machine

Server machine

Networking basics

- Computers running on the Internet communicate with each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).



Application
(HTTP, ftp, telnet, ...)

Transport
(TCP, UDP, ...)

Network
(IP, ...)

Link
(device driver, ...)

Networking Basics

- **Internet protocol (IP)** addresses
  - Every host on Internet has a unique IP address

    `143.89.40.46, 203.184.197.198`

    `203.184.197.196, 203.184.197.197, 127.0.0.1`
  - More convenient to refer is to use hostname string

    `google.com, gmail.com, localhost`
  - One hostname can correspond to multiple internet addresses:
    - www.yahoo.com:

      66.218.70.49; 66.218.70.50; 66.218.71.80; 66.218.71.84; …

DNS-Domain Name System

- The **Domain Name system** (DNS) maps these names to numbers.
- Most importantly, it serves as the "phone book" for the Internet by translating human-readable computer hostnames, e.g. *www.example.com*, into the IP addresses, e.g. *208.77.188.166*, that networking equipment needs to deliver information.

Understanding Ports

- Ports
  - Many different services can be running on the host
  - A **port** identifies a service within a host

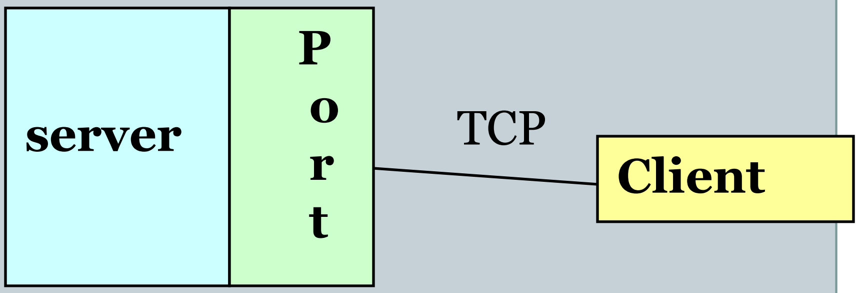
- IP address + port number = "phone number“ for service

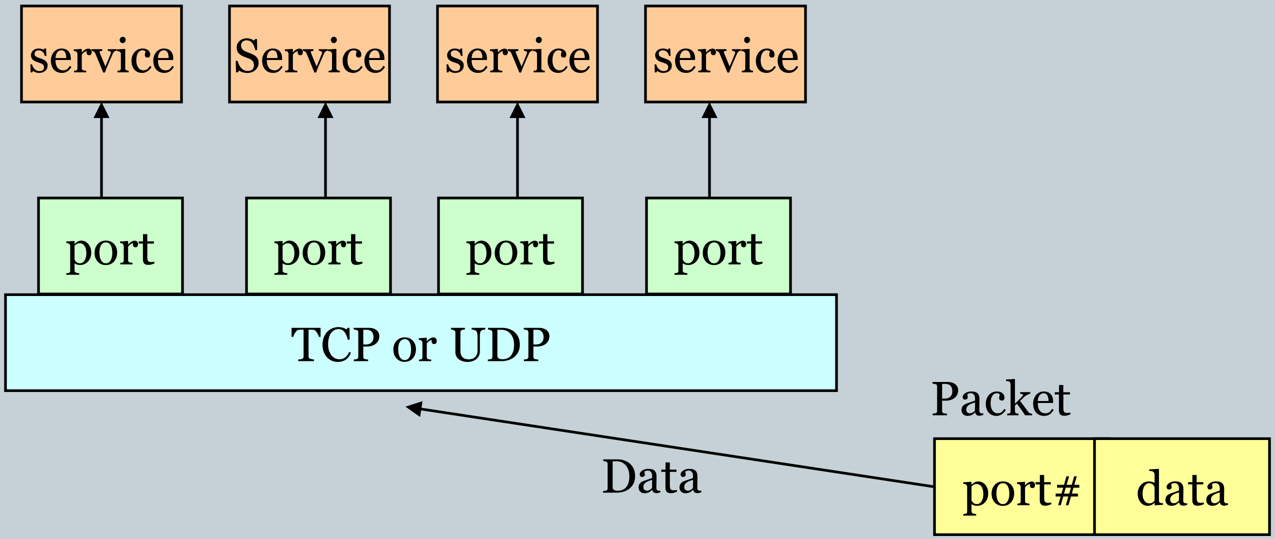
Understanding Ports

- Port is represented by a positive (16-bit) integer value
- Some ports have been reserved to support common/well known services:
  - ftp 21
  - telnet 23
  - smtp 25
  - http 80
- User level process/services generally use port number value >= 1024, since 0-1023 ports are reserved and known as *Well Known Ports*.

Understanding Ports

- The TCP and UDP protocols use *ports* to map incoming data to a particular *process* running on a computer.





Use of Ports

- Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined.
  - The computer is identified by its 32-bit IP address, which is used by Internet Protocol(IP) to deliver data to the right computer on the network.
  - Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.

Types of Communication

- There are 2 types of communication:
  - Connection-oriented communication
  - Connection-less communication

Transmission Control Protocol

- A connection-based protocol that provides a reliable flow of data between two computers.
- Provides a point-to-point channel for applications that require reliable communications.
  - The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel
- 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.

User Datagram Protocol

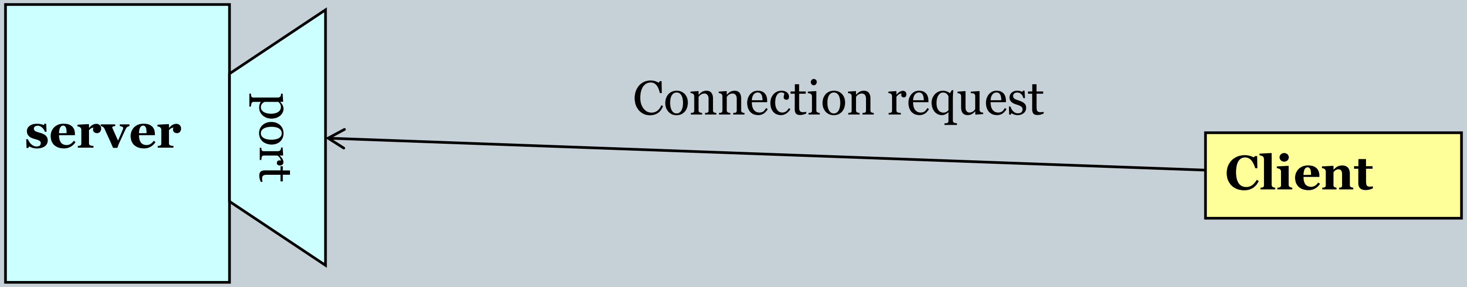
- A protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival.
- UDP is not connection-based like TCP and is not reliable:
  - Sender does not wait for acknowledgements
  - Arrival order is not guaranteed
  - Arrival is not guaranteed
- Used when speed is essential, even in cost of reliability
  - e.g. games etc.

Sockets

- A socket is an endpoint of a two-way communication link between two programs running on the network.
- Network communication using Sockets is very much similar to performing file I/O
  - In fact, socket handle is treated like file handle.
  - The streams used in file I/O operation are also applicable to socket-based I/O

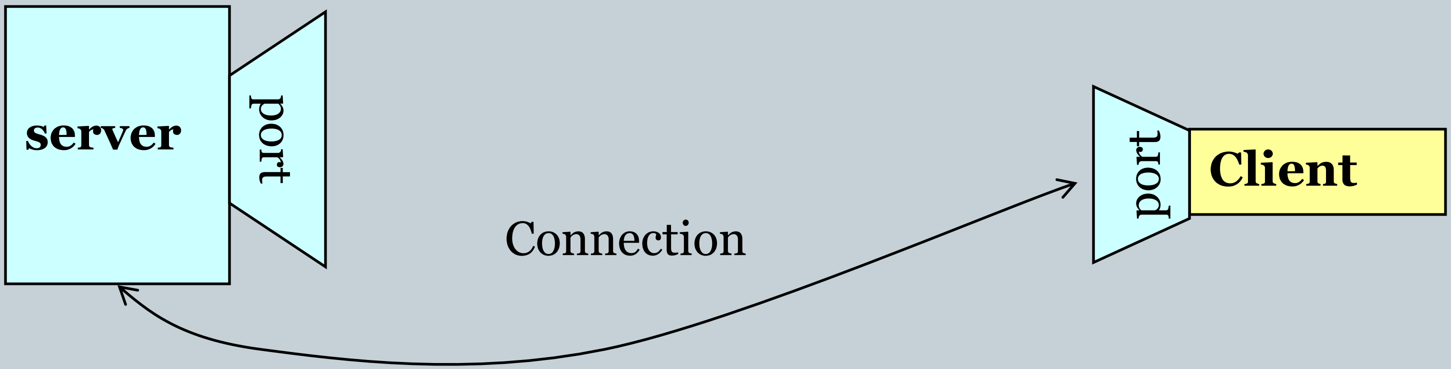
Socket Communication

- A server (program) runs on a specific computer and has a socket that is bound to a specific port.
- The server waits and listens to the socket for a client to make a connection request.



Socket Communication

- If everything goes well, the server accepts the connection.
- Upon acceptance, the server gets a new socket bounds to a different port.
  - It needs a new socket so that it can continue to listen to the original socket for connection requests while serving the connected client.



Networking Classes

- Through the classes in java.net, Java programs can use TCP or UDP to communicate over the Internet.
  - The `URL`, `URLConnection`, `Socket`, and `ServerSocket` classes all use TCP to communicate over the network.
  - The `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` classes are used by UDP.

TCP/IP in Java

- Accessing TCP/IP from Java is straightforward. The main functionality is in the following classes:
  - `java.net.InetAddress` : Represents an IP address (either IPv4 or IPv6) and has methods for performing DNS lookup.
  - `java.net.Socket` : Represents a TCP socket.
  - `java.net.ServerSocket` : Represents a server socket which is capable of waiting for requests from clients.

InetAddress

- The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address.
- We interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address.
- The InetAddress class hides the number inside.
- Serves three main purposes:
  - Encapsulates an address
  - Performs name lookup (converting a host name into an IP address)
  - Performs reverse lookup (converting the address into a host name)

Factory Methods in InetAddress class

- static InetAddress getLocalHost( )

  throws UnknownHostException

  - *Returns the InetAddress object that represents the local host.*
- static InetAddress getByName(String hostName)

  throws UnknownHostException

  - *Returns the InetAddress for a host name passed to it.*
- static InetAddress[ ] getAllByName(String hostName)

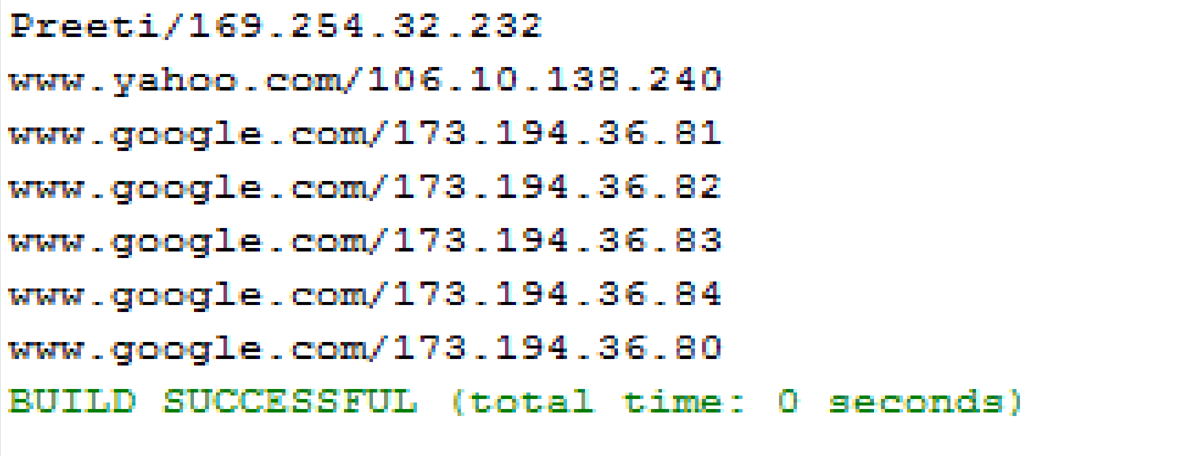
  throws UnknownHostException

  - *Returns an array of InetAddress that represent all the names that a passes hostName resolves to.*
- *UnknownHostException* is thrown if DNS system can not find the IP address for specific host.

Example:

```
class InetAddressTest
{
        public static void main(String args[])
                throws UnknownHostException
        {
                InetAddress Address = InetAddress.getLocalHost();
                System.out.println(Address);
                Address =
                InetAddress.getByName("www.yahoo.com");
                System.out.println(Address);
                InetAddress SW[] =
                InetAddress.getAllByName("www.google.com");
                for (int i=0; i<SW.length; i++)
                        System.out.println(SW[i]);
        }
}
```

Example



```
Preeti/169.254.32.232
www.yahoo.com/106.10.138.240
www.google.com/173.194.36.81
www.google.com/173.194.36.82
www.google.com/173.194.36.83
www.google.com/173.194.36.84
www.google.com/173.194.36.80
BUILD SUCCESSFUL (total time: 0 seconds)
```

Socket Programming

CLIENT/SERVER ARCHITECTURE

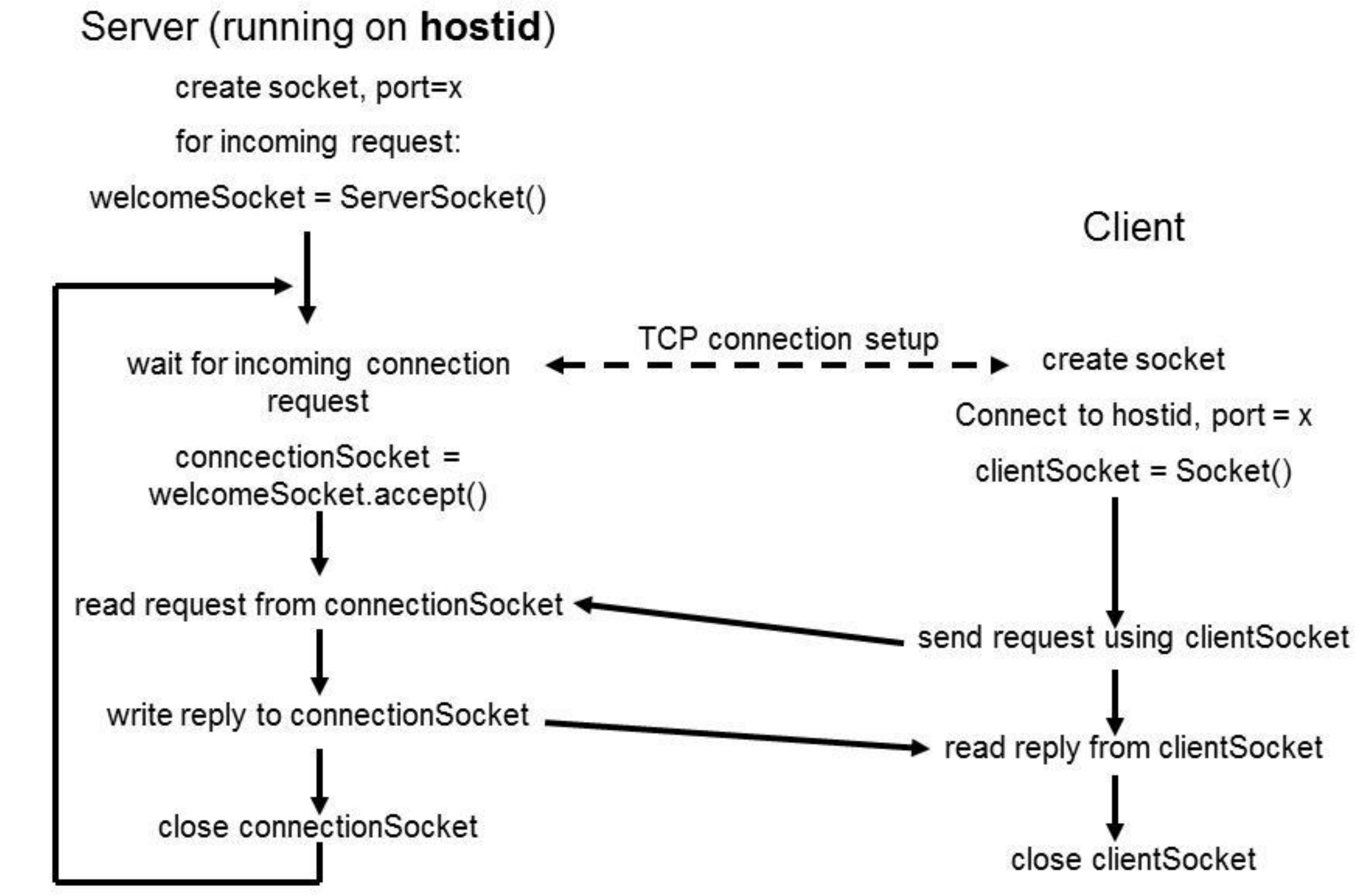
Socket Classes

- A socket is bound to a port number so that the TCP layer can identify the application that data destined to be sent.
- Java.net package provides two classes:
  - Socket – for implementing a client
  - ServerSocket – for implementing a server

Two types of TCP Sockets

- `java.net.Socket` is used by clients who wish to establish a connection to a (remote) server
  - A client is a piece of software (usually on a different machine) which makes use of some service
- `java.net.ServerSocket` is used by servers so that they can accept incoming TCP/IP connections
  - A server is a piece of software which *advertises* and then provides some service on request

Client-Server Interaction via TCP



Server (running on **hostid**)

create socket, port=x

for incoming request:

welcomeSocket = ServerSocket()

wait for incoming connection request

conncectionSocket = welcomeSocket.accept()

read request from connectionSocket

write reply to connectionSocket

close connectionSocket

TCP connection setup

Client

create socket

Connect to hostid, port = x

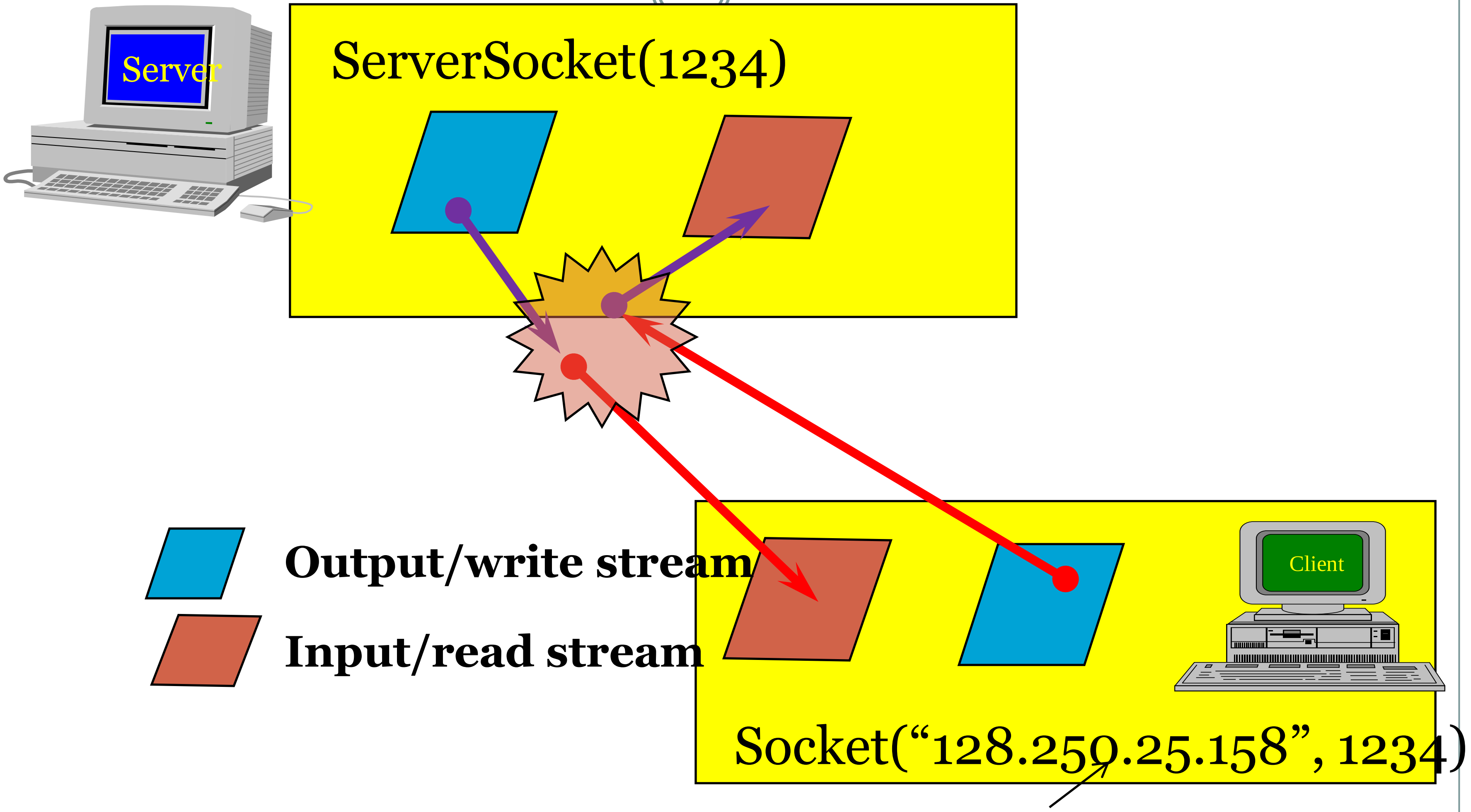
clientSocket = Socket()

send request using clientSocket

read reply from clientSocket

close clientSocket

Java Sockets



ServerSocket(1234)

🟦 Output/write stream

🟥 Input/read stream

Socket(“128.250.25.158”, 1234)

It can be host_name like “books.google.com”

ServerSocket

- The **ServerSocket** class is used to create socket for server that listen for either local or remote client programs to connect to them on published port.
- A server socket waits for requests to come over the network. It performs some operation based on that request, and then possibly returns a result to the client.
- When a client connects to a server socket, a TCP connection is made, and a (normal) socket is created for each end point.

Constructors

- ServerSocket (int port)

  throws BindException, IOException

  - *Creates server socket on the specified port with a queue length of 50*

- ServerSocket (int port, int maxQueue)

  throws BindException, IOException

  - *Creates server socket on the specified port with a maximum queue length of maxQueue*

- ServerSocket (int port, int maxQ, InetAddress localAddress) *throws IOException*

  - *Creates a server socket on the specified port with a maximum queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds.*

Some useful methods

- Socket accept()
  - Block waiting for a client to attempt to establish a connection.
- void close()
  - Called by the server when it is shutting down to ensure that any resources are deallocated

Accepting Connections

- Usually, the `accept()` method is executed within an infinite loop
  - i.e., `while(true){...}`
- The accept method returns a new socket (with a new port) for the new channel. It blocks until connection is made.
- Syntax:
  - Socket accept() throws IOException

Implementing a Server

- Open the Server Socket:

```
ServerSocket server;
DataOutputStream os;
DataInputStream is;
server = new ServerSocket( PORT );
```

- Wait for the Client Request:

```
Socket client = server.accept();
```

- Create I/O streams for communicating to the client

```
is = new DataInputStream(client.getInputStream() );
os = new DataOutputStream(client.getOutputStream());
```

- Perform communication with client

  Receive from client: `String line = is.readLine();`

  Send to client: `os.writeBytes("Hello\n");`

- Close sockets:

```
client.close();
```

Client Sockets

- Java wraps OS sockets (over TCP) by the objects of class java.net.Socket

  Socket(String *remoteHost*, int *remotePort*)

- Creates a TCP socket and connects it to the remote host on the remote port (hand shake)
- Write and read using streams:
  - InputStream getInputStream()
    - Returns the InputStream associated with invoking socket
  - OutputStream getOutputStream()
    - Returns the OutputStream associated with invoking socket

Constructors

- Socket(String *remoteHost*, int *remotePort*)
  - Creates a socket connecting the local host to the named host and port; can throw an **UnknownHostException** if the named host is not found.
- Socket(InetAddress ip, int *remotePort*)
  - Creates a socket using a preexisting InetAddress object and a port;

Instance Methods

- InetAddress getInetAddress( )
  - Returns the InetAddress associated with the Socket object.
  - It returns null if socket is not connected.
- int getPort( )
  - Returns the remote port to which invoking Socket object is connected.
- int getLocalPort( )
  - Returns the local port to which invoking Socket object is connected.
- void close( )
  - This method is used to close the connection created between Client and Server.

Implementing a Client

1. Create a Socket Object:

```
client = new Socket( server, port_id );
```

2. Create I/O streams for communicating with the server:

```
istream = new BufferedReader(new
    InputStreamReader(client.getInputStream()));
ostream = new PrintWriter(client.getOutputStream() );
```

3. Perform I/O or communication with the server:

- Receive data from the server:

```
String line = istream.readLine();
```

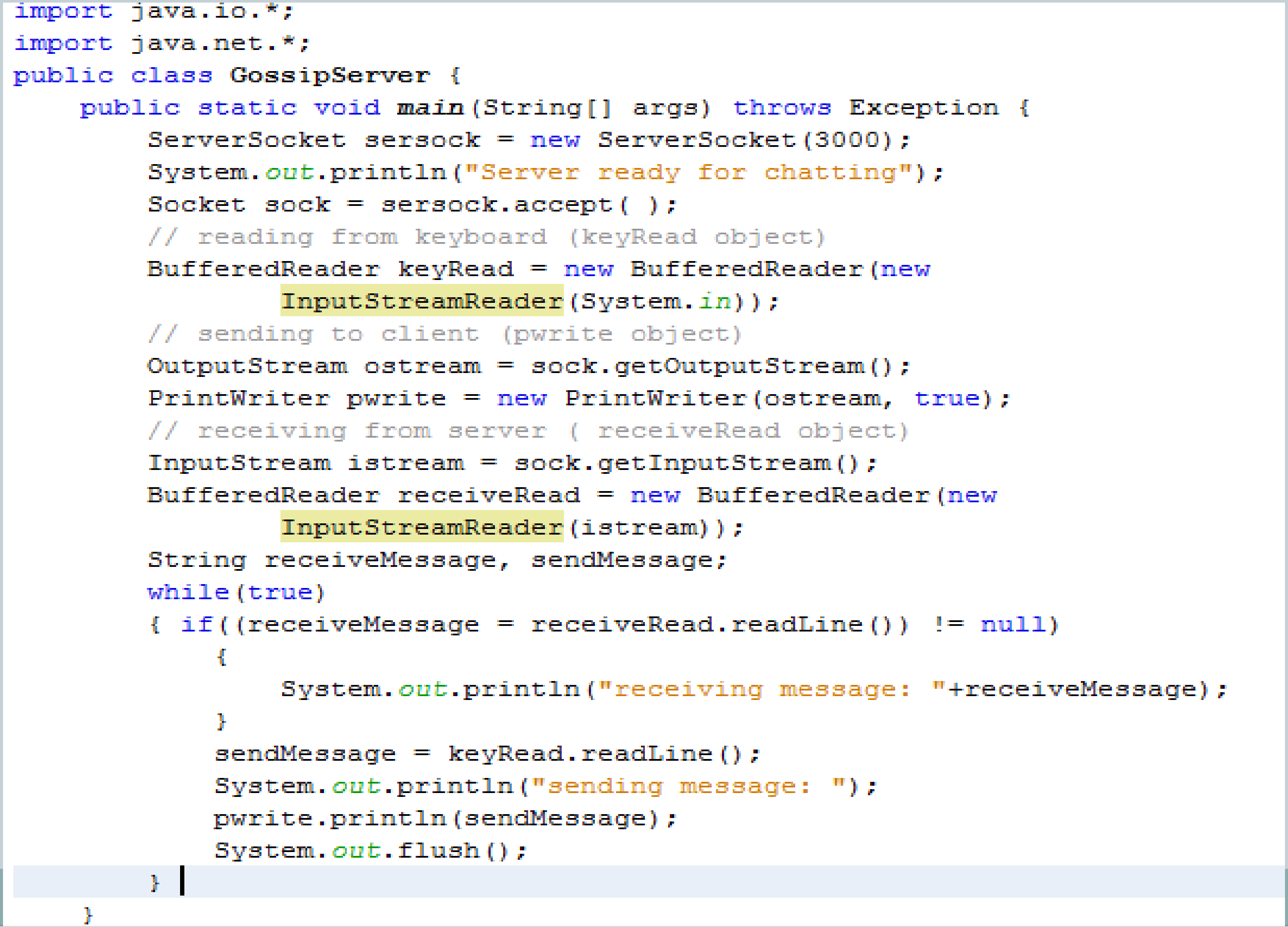
- Send data to the server:

```
ostream.println("Hello\n");
```

4. Close the socket when done:

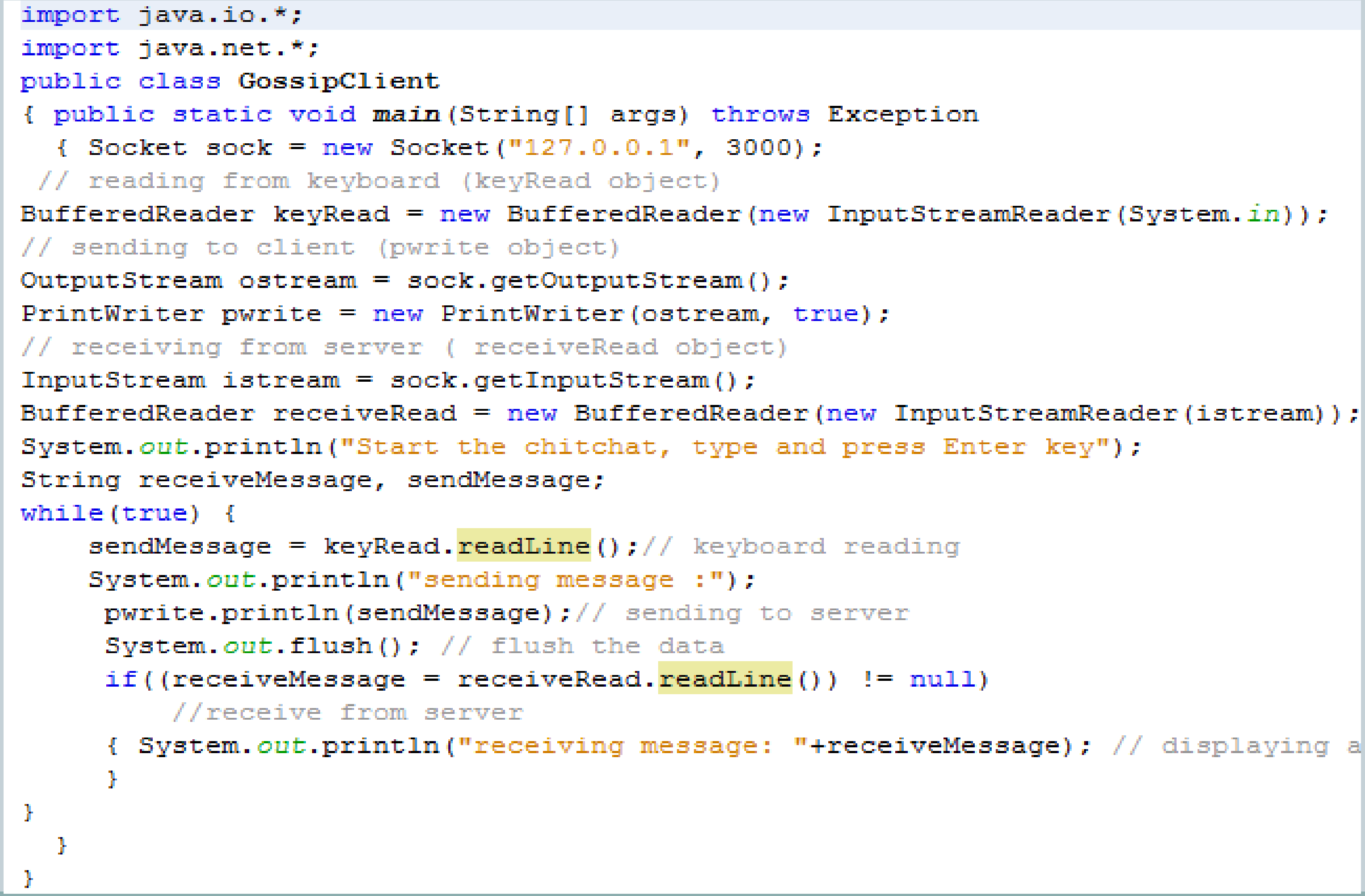
```
client.close();
```

Chatting Program(Server side)



```java
import java.io.*;
import java.net.*;
public class GossipServer {
    public static void main(String[] args) throws Exception {
        ServerSocket sersock = new ServerSocket(3000);
        System.out.println("Server ready for chatting");
        Socket sock = sersock.accept( );
        // reading from keyboard (keyRead object)
        BufferedReader keyRead = new BufferedReader(new
                InputStreamReader(System.in));
        // sending to client (pwrite object)
        OutputStream ostream = sock.getOutputStream();
        PrintWriter pwrite = new PrintWriter(ostream, true);
        // receiving from server ( receiveRead object)
        InputStream istream = sock.getInputStream();
        BufferedReader receiveRead = new BufferedReader(new
                InputStreamReader(istream));
        String receiveMessage, sendMessage;
        while(true)
        { if((receiveMessage = receiveRead.readLine()) != null)
            {
                System.out.println("receiving message: "+receiveMessage);
            }
            sendMessage = keyRead.readLine();
            System.out.println("sending message: ");
            pwrite.println(sendMessage);
            System.out.flush();
        }
    }
```

Chatting Program(Client side)



```java
import java.io.*;
import java.net.*;
public class GossipClient
{ public static void main(String[] args) throws Exception
  { Socket sock = new Socket("127.0.0.1", 3000);
 // reading from keyboard (keyRead object)
BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
// sending to client (pwrite object)
OutputStream ostream = sock.getOutputStream();
PrintWriter pwrite = new PrintWriter(ostream, true);
// receiving from server ( receiveRead object)
InputStream istream = sock.getInputStream();
BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));
System.out.println("Start the chitchat, type and press Enter key");
String receiveMessage, sendMessage;
while(true) {
    sendMessage = keyRead.readLine();// keyboard reading
    System.out.println("sending message :");
     pwrite.println(sendMessage);// sending to server
     System.out.flush(); // flush the data
     if((receiveMessage = receiveRead.readLine()) != null)
         //receive from server
     { System.out.println("receiving message: "+receiveMessage); // displaying a
     }
}
  }
}
```

Datagram

SOCKET PROGRAMMING WITH UDP

Datagrams

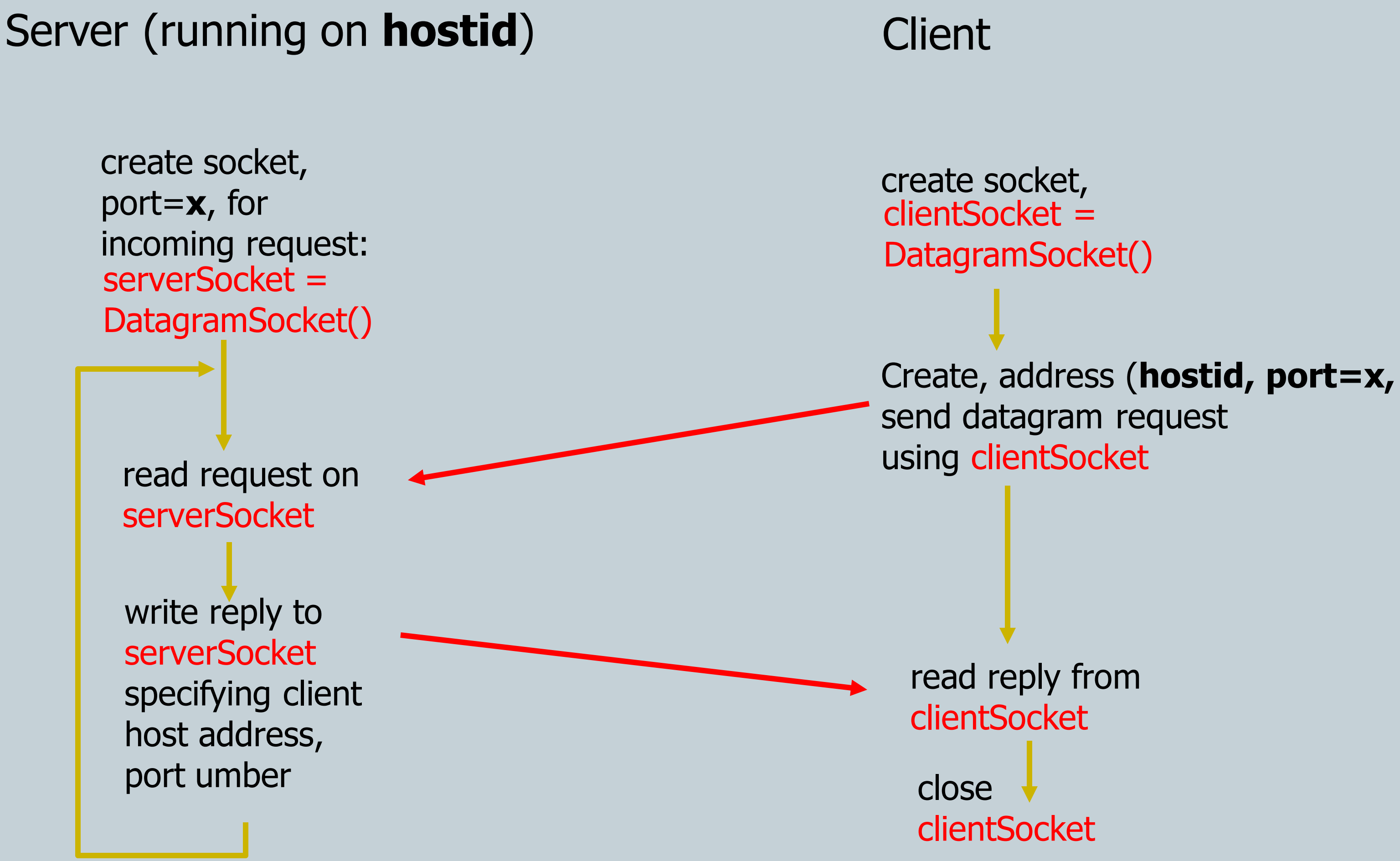
- A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.
- The java.net package contains classes to use datagrams to send and receive packets over the network: DatagramSocket and DatagramPacket

Socket programming with UDP

- UDP
  - Connectionless and unreliable service.
  - There isn’t an initial handshaking phase.
  - Doesn’t have a pipe.
  - Transmitted data may be received out of order, or lost

- Socket Programming with UDP
  - No need for a socket.
  - No streams are attached to the sockets.
  - The sending hosts creates “packets” by attaching the IP destination address and port number to each batch of bytes.

Client/Server Socket Interaction:UDP



Server (running on **hostid**)

Client

create socket,
port=**x**, for
incoming request:
serverSocket =
DatagramSocket()

read request on
serverSocket

write reply to
serverSocket
specifying client
host address,
port umber

create socket,
clientSocket =
DatagramSocket()

Create, address (**hostid, port=x,**
send datagram request
using clientSocket

read reply from
clientSocket

close
clientSocket

DatagramSocket

- public DatagramSocket( ) throws SocketException
  - Allocate any avaliable port number for creating a socket on local host(used for receiving datagram)
- public DatagramSocket(int port) throws SocketException
  - Use the specified port number for creating a socket on local host(used for receiving datagram)

DatagramPacket

- public DatagramPacket(byte[] buffer, int length)
  - This constructor specifies a buffer that will receive data, and the size of a packet.
  - Example:

```
      byte[] buffer = new byte[8192];
      DatagramPacket dp = new
                 DatagramPacket(buffer, buffer.length);
```

- public DatagramPacket(byte[] buffer, int offset, int length)
  - The second form allows to specify an offset into the buffer at which data will be stored.

DatagramPacket

- public DatagramPacket(byte[] data, int length, InetAddress destination, int port)
  - This form specifies a target address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent.
- public DatagramPacket(byte[] data, int offset, int length, InetAddress destination, int port)
  - This form allows to transmits packets beginning at the specified offset into the data.

Sending and Receiving Packets

- public void send(DatagramPacket dp) throws IOException
  - Sends the full datagram out onto the network
- public void receive(DatagramPacket dp) throws IOException
  - Waits until a datagram fills in emptyPacket with the message

Several methods defined by DatagramPacket

- InetAddress getAddress( )
  - Returns the **InetAddress** of the source.
- int getPort( )
  - Returns the port number.
- byte[ ] getData( )
  - Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
- int getLength( )
  - Returns the length of the data contained in the byte array that would be returned from the **getData( ) method.**
- int getOffset()
  - Returns the starting index of the data.

Several methods defined by DatagramPacket

- void setAddress(InetAddress *ipAddress)*
  - Sets the address to which a packet will be sent. The address is specified by *ipAddress*.
- void setData(byte[] *data*)
  - Sets the data part of a packet to *data*, the offset to zero, length to number of bytes in *data*.
- void setData(byte[] *data*, int *index*, int *size*)
  - Sets the data to *data*, the offset to *index*, and the length to *size*.
- void setLength(int *size*)
  - Sets the length of the packet to *size*.
- void setPort(int *port*)
  - Sets the port to specified *port number*.

Example: DatagramSender

- This example sends datagrams to a specific host (anywhere on the Internet)
- The steps are as follows:
  - Create a new DatagramPacket
  - Put some data which constitutes your message in the new DatagramPacket
  - Set a destination address and port so that the network knows where to deliver the datagram
  - Create a socket with a *dynamically allocated* port number
  - Send the packet through the socket onto the network

Example: DatagramSender

```
byte[] data = “This is the message”.getBytes();
DatagramPacket packet =
      new DatagramPacket(data, data.length);

// Create an address
InetAddress destAddress =
      InetAddress.getByName(“fred.domain.com”);
packet.setAddress(destAddress);
packet.setPort(9876);

DatagramSocket socket = new DatagramSocket();
socket.send(packet);
```

Example:DatagramReceiever

- The steps for sending the data:
  - Create an empty DatagramPacket (and allocate a buffer for the incoming data)
  - Create a DatagramSocket on an *agreed* socket number to provide access to arrivals
  - Use the socket to receive the datagram (the thread will block until a new datagram arrrives)
  - Extract the data bytes which make up the message

Example:DatagramReceiever

```
// Create an empty packet with some buffer space
byte[] data = new byte[1500];
DatagramPacket packet =
      new DatagramPacket(data, data.length);

DatagramSocket socket = new DatagramSocket(9876);

// This call will block until a datagram arrives
socket.receive(packet);

// Convert the bytes back into a String and print
String message =
   new String(packet.getData());
System.out.println("message is " + message);
System.out.println("from " + packet.getAddress());
```

UDP Server.java

```
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());
```

UDP Server.java

```
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);

    }
  }
}
```

UDP Client.java

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

class UDPClient {
    public static void main(String args[]) throws Exception
    {

      BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

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

      String sentence = br.readLine();

      sendData = sentence.getBytes();
```

UDP Client.java

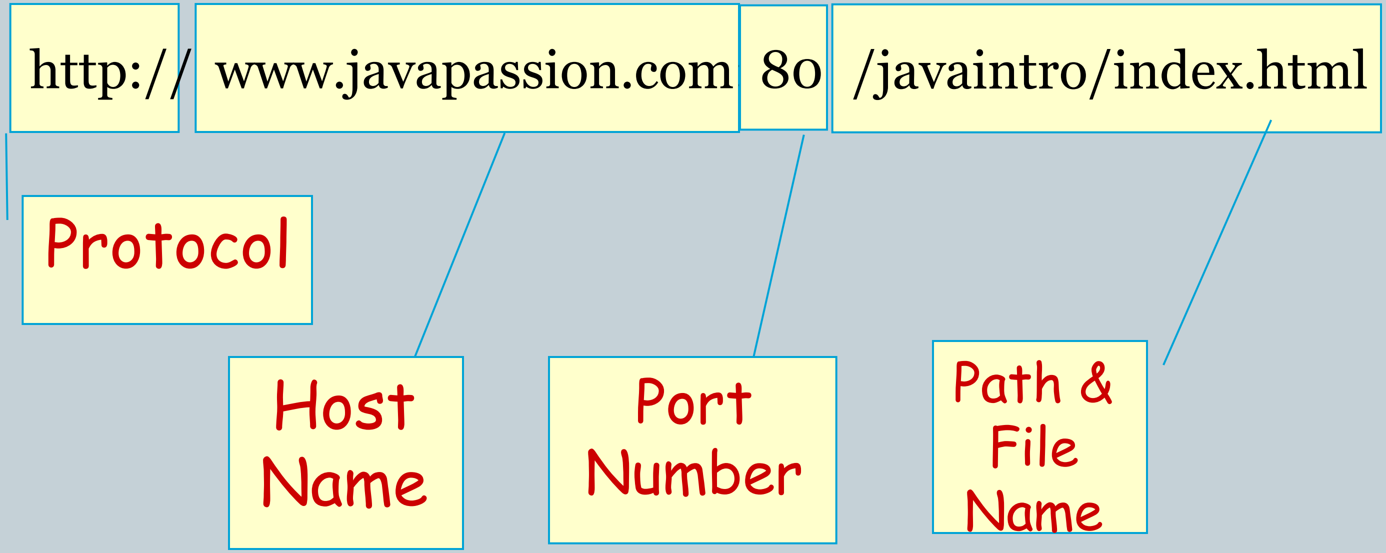
```
DatagramPacket sendPacket =
      new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

 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();

   }
}
```

URL

UNIFORM RESOURCE LOCATOR

URL - *Uniform Resource Locator*

- URL is a reference (an address) to a resource on the Internet.
  - A resource can be a file, a database query and more.
- URL provides form to uniquely identify or address information on the internet.



http:// www.javapassion.com 80 /javaintro/index.html

Protocol

Host Name

Port Number

Path & File Name

Class URL

- Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.

Constructors

- URL(String urlSpecifier)
  - Allows to create a URL for the specified context
- URL(URL urlObj, String urlSpecifier)
  - Allows to use an existing URL as a reference context and then create a new URL from the specified context.
- URL(String protName, String hostName, int port, String path)
- URL(String protName, String hostName, String path)
  - Above two constructors allow to break up the URL into its component parts.
- Each can throw an exception of **MalformedURLException**

Example

```
class URLDemo
{
   public static void main(String args[])
      throws MalformedURLException
   {
      URL hp = new URL("http://content-

ind.cricinfo.com/ci/content/current/story/news.html");
      System.out.println("Protocol: " + hp.getProtocol());
      System.out.println("Port: " + hp.getPort());
      System.out.println("Host: " + hp.getHost());
      System.out.println("File: " + hp.getFile());
      System.out.println("Ext:" + hp.toExternalForm());
   }
}
```

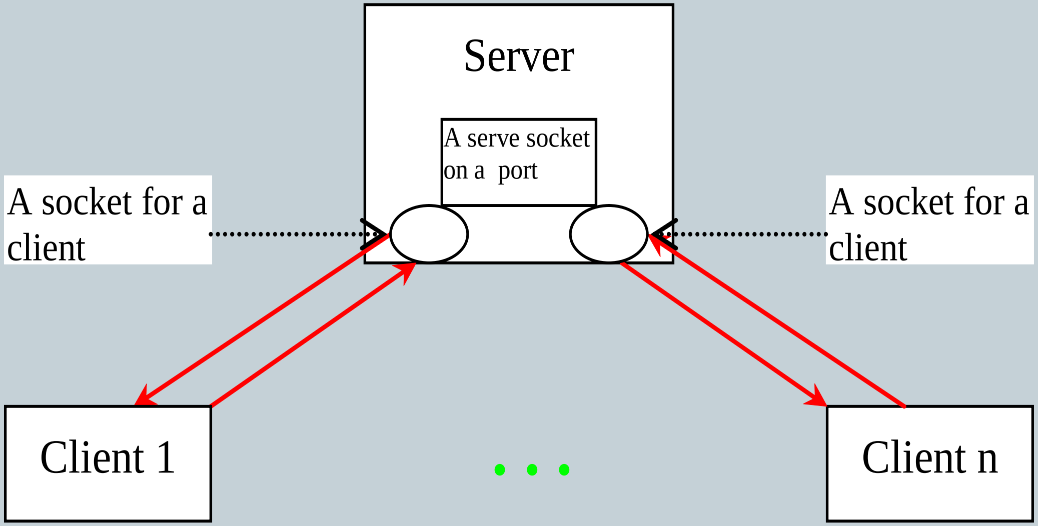
Output

```
Protocol: http
Port: -1
Host: content-ind.cricinfo.com
File: /ci/content/current/story/news.html
Ext:http://content-ind.cricinfo.com/ci/content/current/story/news.html
```

MultiThreading

SERVING MULTIPLE CLIENTS

Example: Serving Multiple Clients



Server

A serve socket on a port

A socket for a client

A socket for a client

Client 1

. . .

Client n

Note: Start the server first, then start multiple clients.

Serving Multiple Clients

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to connect to it. You can use threads to handle the server's multiple clients simultaneously. Simply create a thread for each connection. Here is how the server handles the establishment of a connection:

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

The server socket can have many connections. Each iteration of the while loop creates a new connection. Whenever a connection is established, a new thread is created to handle communication between the server and the new client; and this allows multiple connections to run at the same time.