Network Programming

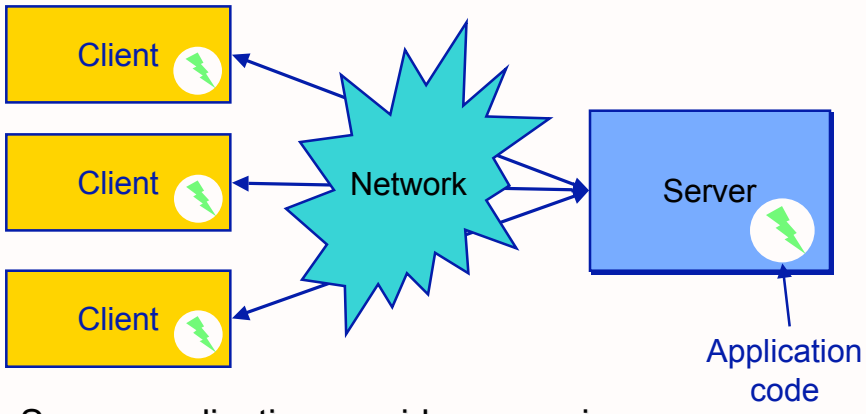
Sara Sprenkle

July 27, 2006

Network Programming and Java

- A computer network is an interconnected collection of autonomous computers
  - each computer is independent
  - some method of communication exists between them
- Java abstracts all of the details of the underlying network and how the operating system interacts with it
  - Hidden and encapsulated in the java.net package
  - Makes network programming easier

Distributed Programming



- Server application provides a service
- Client program(s) communicates with server application

Network Addresses

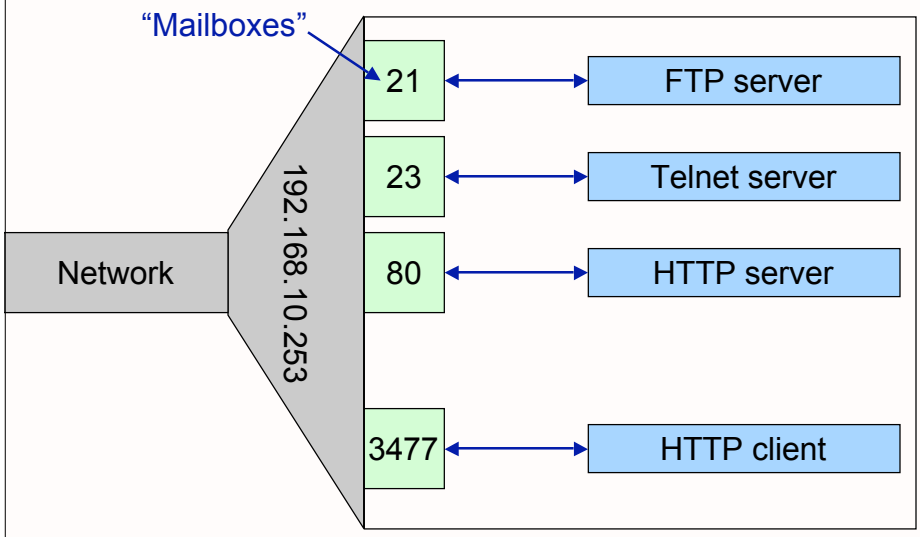
- A computer on a network has an address.
  - address is used to uniquely identify the computer (also known as a host) on the network
- The most common address system in use today is the Internet Protocol (IPv4) addressing system
  - a 32-bit address, typically written as a "dotted-quad": four numbers, 0 through 254, separated by dots, e.g.,

192.168.10.253

Ports

- Each host on the network has a set of ports
  - Ports are like mailboxes: the address specifies the host, the port specifies the application on the host
  - Ports range from 1 to 65537
- These ports allow multiple applications to use the same network interface/address to communicate over the network.
- For example
  - a web server will communicate on the network using port 80
  - an FTP server on the same host will have the same address but use port 21

A Machine's Ports



Well-Known Ports

- Port numbers < 1024 are well-known ports
  - Well-known ports are assigned to application servers
  - Port 80 always has an HTTP server
  - Port 21 always has an FTP server
- Client listens on another port (above 1024) to receive responses from a server
- No technical reason servers must conform to these standards
  - A convention so that clients will always know where the web server is, where the FTP server is, …
  - Can have an HTTP server at a port > 1024

Sockets

- A socket is an abstraction of an endpoint of a two-way communications link
- An application creates a socket that is bound to a remote address and remote port.
  - port on the host (client) could be random
  - a "connection" is created between the client using this port and the specified remote address at the remote port

Services provided by networks

- Connection-oriented
- Connection-less

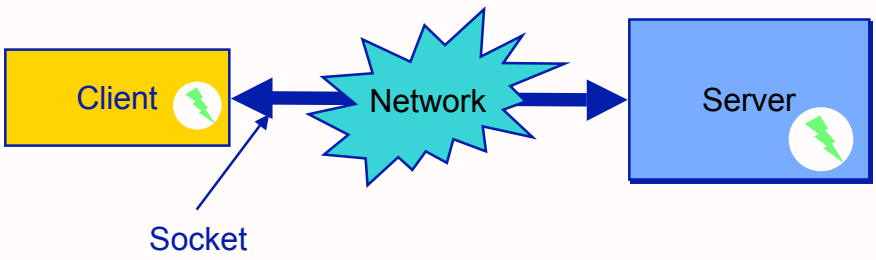
Connection-Oriented Service

- A connection-oriented service is like the telephone system
  - acts like a pipe
  - sender pushes object bits into the pipe and then come out of the receiver in the same condition as they were pushed in
  - pipe is connected to one port on the sender and one port on the receiver
- Implemented in Java using stream sockets

Stream Sockets

- After a stream socket is open on both the server and the client and the sockets connect to each other, a pipe connects endpoints and provides a reliable byte stream
- Transmission Control Protocol (TCP)
  - most popular protocol that implements a stream, or connection-oriented, service
  - Java uses to implement stream sockets
  - Reliable service
    - when something is sent from one end to the other, it arrives in order, in the same state, and is not lost or duplicated in the network

Stream Sockets



- Stream Socket: communicates using TCP
  - Hides details of TCP from programmer

Connectionless Service

- A connectionless service is like the postal system
  - One side sends messages to the other side
  - Each message is independent
  - Can lose messages in the network, duplicate messages, corrupt data during transport
  - An unreliable service
    - although most of the time this bad stuff does not happen -- much more reliable underlying network
  - One side creates a message and sends it to the other side

Datagram Sockets

- User Datagram Protocol (UDP)
  - Popular protocol that Java uses to implement datagram sockets
  - Unreliable: All sorts of things can happen to the messages during transport in the network (although most of the time, they get there just fine).
- No connection between these sockets
  - A socket is opened to another socket, but no connection is actually made
  - When a message is passed to the socket, it is sent over the network to the other socket. Most of the time it gets there.

Example Java Client Program

- Connect to a server (another host on the network)
- Open a stream to a certain port
- Display what the server sends


```
public class SocketTest {
   public static void main(String[] args) {
      try {
         Socket s = new Socket(
            "time-A.timefreq.bldrdoc.gov", 13);
         BufferedReader in = new BufferedReader(
            new InputStreamReader(s.getInputStream()));

         boolean more = true;
         while (more) {
            String line = in.readLine();
            if (line == null) more = false;
            else              System.out.println(line);
         }
      } catch (IOException exp) {
         System.out.println("Error:" + exp);
      }
   }
}
```

Reading from a Socket

```
Socket s = new Socket(
      "time-A.timefreq.bldrdoc.gov", 13);

BufferedReader in = new BufferedReader(new
      InputStreamReader(s.getInputStream()));
```

- The first line creates a socket that connects to the host with the specified name at port 13 on that host
- getInputStream() is called on the socket to get a byte stream that reads from the socket
- An InputStreamReader wraps the byte stream and a BufferedReader wraps the InputStreamReader
- The BufferedReader reads all characters sent by the server using readLine() and displays each line to System.out.

Network I/O and Exceptions

- All of the networking code in this example is inside of a try block
- A number of things can go wrong with network communications
  - a power failure knocking out an intermediate router or switch
  - a misconfiguration,
  - someone tripping over a cable
- If any of these errors are detected, an IOException is generated, so any program performing such functionality should handle such exceptions

Host Names and IP Addresses

- A name is provided to the socket constructor
  - not an IP address
  - called a host name
- Java uses the Domain Name Service (DNS) to resolve the host name into an IP address
  - connect to the host using the IP address
- Usually, you will not work directly with IP addresses
  - You can connect a socket using a host's IP address

Host Names and IP Addresses

- InetAddress's static method, getByName().
- For example,

```
InetAddress addr = InetAddress.getByName(
     "www.udel.edu");
```

will return an InetAddress object that encapsulates the sequence of four bytes

128.175.13.63

Multiple IP Addresses per Host

- A host can have more than one IP address
  - facilitate load-balancing.
  - For example, www.cnn.com currently corresponds to 8 different IP addresses
    - one can be picked at random whenever the host is accessed (usually just the first)
- To determine all of the IP addresses of a specific host, call getAllByName()…

```
InetAddress[] addresses = InetAddress.getAllByName(
      "www.cnn.com");
```


The Loopback Address and localhost

- To get information about the machine the program is running on, the hostname **localhost** always represents the local host
- Hostname corresponds to the IP address 127.0.0.1, which is known as the loopback address.
  - a special IP address that means "the computer connected right here"

Determining the Local Address

- If the program calls getByName() with localhost, the returned IP address is 127.0.0.1
- To get the actual IP address of the host, call getLocalHost()
  - returns the actual IP address of the host on the network
- For example…

```
InetAddress address =
     InetAddress.getLocalHost();
```

InetAddressPractice.java

A Bi-Directional Client

- Our simple client program connects to a server and displays what the server sent back
  - After the server finishes, the client disconnects
- Often, the client wants to send data to the server as well as receive data from the server
  - Sockets are bi-directional
  - Need to open an output stream on the socket

This test program opens both input and output streams on the same socket – to both read from and write to the server.

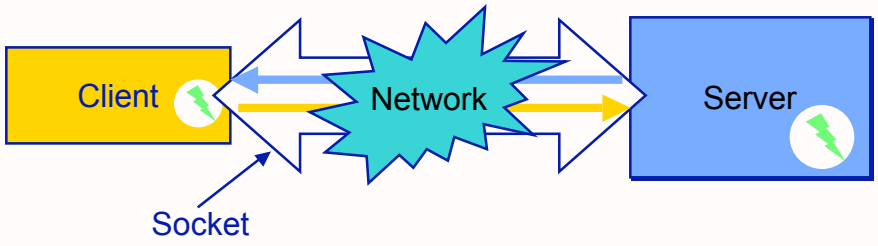
```
public class BidirSocketTest {
   public static void main(String[] args) {
      try {
         Socket s = new Socket(
            "time-A.timefreq.bldrdoc.gov", 13);
         BufferedReader in = new BufferedReader(
            new InputStreamReader(s.getInputStream()));
         PrintWriter out = new PrintWriter(
            s.getOutputStream(), true);     // auto-flush
         // read from in (input – received from server)
         // write to out (output – send to server)

   } catch (IOException exp) {
         System.out.println("Error:" + exp);
      }
   }
}
```


Clients and Servers

- When we open a connection, it is made to a host at a certain address to a certain port.
- For this to work, the server on the remote host must be listening to that port and wait for a client to connect to that port
  - the server obtains a socket that is an abstraction of its end of the stream, connected to the connecting client

The Socket Abstraction



- Each end of socket has input/output

The ServerSocket Class

- Server programs (programs that listen to a port for a connection request) are implemented using the **ServerSocket** class.
- A ServerSocket object is created by specifying the port number to listen for connections on…

```
ServerSocket svr1 = new ServerSocket(1998);
```

  - creates a server socket on port 1998
    - not a well-known port number because it is > 1024
  - Server object listens for connection requests on this port

Accepting a Connection

- The server program can wait for a client request to connect on that port by calling accept()
  - blocking method that waits indefinitely until a client attempts to connect to the port
  - When client connects, accept() returns a Socket object, which is how the server communicates with the client…

```
// will block until a client connects
Socket incoming = svr1.accept();
```


Example: An Echo Server

- Create a simple server that will wait for a client to connect
- When a client connects, the server will read a line from the client and then return a line identical to what it has received.
- Known as an echo server because it echoes back what it receives from the client
- As an added twist, echo server will echo back what it receives in all capital letters

```
public class CapsEchoServer {
   public static void main(String[] args) {
      try {
         ServerSocket svr1 = new ServerSocket(1998);
         Socket incoming = svr1.accept();
         BufferedReader in = new BufferedReader(new
            InputStreamReader(incoming.getInputStream());
         PrintWriter out = new PrintWriter(
            incoming.getOutputStream(), true);
         out.println("CAPS Echo Server. Type BYE to exit");
         boolean done = false;
         while (!done) {
            String line = in.readLine();
            if (line == null) done = true;
            else if (line.trim().equals("BYE")) done = true;
            else out.println("Echo:" + line.trim().toUpperCase());
         }
         incoming.close();
      } catch (IOException exp) {
         System.out.println("Error:" + exp);
      }
}}
```


Example: An Echo Server

- Purpose of a ServerSocket object is to wait for connections
- When a client connects, it generates a new Socket object, which is the server's endpoint of the connection, and returns the from the call to accept()
- Suppose we would like to allow multiple clients to connect to the echo server at the same time

Servers and Multiple Clients

- Servers should handle multiple concurrent clients
  - If a server only allowed one client to connect at any given time, any client can monopolize the service by remaining connected to the server for a long time
- Who does this sound like a job for?

Servers and Multiple Clients

- After the server returns from accept() with the Socket object, the server can start a new thread to handle the connection between the server and this client
- The main server program can go back and call accept() again, waiting for a new client to connect

A Multithreaded Server

```
while (true)
{
      Socket incoming = svr1.accept();
      Thread clientThread =
           new ThreadedEchoHandler(incoming);
      clientThread.start();
}
```

- User-defined **ThreadedEchoHandler** class derives from Thread
- the client communication loop is its run() method…


```
class ThreadedEchoHandler extends Thread {
   ThreadedEchoHandler(Socket incoming)
   { this.incoming = incoming; }

   public void run() {
      try {
         BufferedReader in = new BufferedReader(
            new InputStreamReader(incoming.getInputStream()));
         PrintWriter out = new PrintWriter(
            incoming.getOutputStream());
         boolean done = false;
         while (!done) {
            String line = in.readLine();
            if (line == null) done = true;
            else if (line.trim().equals("BYE")) done = true;
            else out.println("Echo:" + line.trim().toUpper();
         }
         incoming.close(); } catch (IOException exp)
            { System.out.println("Error:" + exp); }
   }
   Socket incoming;
}
```

A Multithreaded Server

- Each connection starts a new thread
  - multiple clients can connect to the server at the same time
- As soon as a client connects, accept() returns a Socket that encapsulates this new connection
  - socket is passed into a new thread to handle the connection
  - The thread is then started and deals with the connection from then on
- The main thread goes back to waiting for a new connection

Multithreaded Server Issues

- Any problems with having a thread handle each incoming request?
  - Performance

Multithreaded Server Issues

- For each request, must create a thread
  - Overhead in creating threads
- What happens if receive too many client requests and have to start/fork too many threads?
  - Machine runs out of memory
  - Machine gets bogged down
  - Threads can't make progress

Multi-threaded Server Solutions

- Solution: Limit the number of incoming connections/threads available
- new ServerSocket(int port, int backlog)
  - The maximum length of the queue
  - After <backlog> requests, additional requests are refused
- Create a thread pool
  - Create available threads at startup
  - Get one of these threads when to handle requests
  - See java.util.concurrent.Executors

Other Types of Network Streams

- So far, we have connected BufferedReaders and PrintWriters to our socket's input and output streams
  - To receive and send text from the streams
  - Reader/Writer classes handle text
- Suppose we wanted to exchange typed data over a network socket
  - wrap a DataInputStream object and a DataOutputStream object to the socket

Typed Data Network Streams

- For a client program
  - Create a Socket object
  - Obtain the input and output streams associated with that socket
  - Attach DataInputStream and DataOutputStream objects

```
Socket skt1 = new Socket("localhost",1998);
DataInputStream in =
   new DataInputStream(skt1.getInputStream());
DataOutputStream out =
   new DataOutputStream(skt1.getOutputStream());
doubleData1 = in.readDouble();
out.writeDouble(doubleData2);
```

Object Network Streams

- We can follow a similar approach if we want to transfer objects over the network
  - the objects must be Serializable

```
Socket skt1 = new Socket("localhost",1998);
ObjectInputStream in =
   new ObjectInputStream(skt1.getInputStream());
ObjectOutputStream out =
   new ObjectOutputStream(skt1.getOutputStream());
chicken1 = (Chicken)in.readObject();
out.writeObject(chicken2);
```


Socket Timeouts

- In a real-life situation, reading from a socket indefinitely is a bad idea
  - the network could go down, causing the program to wait on the socket forever.
- Java supports a timeout value
  - If the program has been waiting for the socket for the specified timeout interval, a InterruptedIOException is generated
  - Timeout value is set by calling setSoTimeout() on the socket

Socket Timeouts

```
Socket sckt1 = new Socket(. . . );
sckt1.setSoTimeout(10000);  // 10 second timeout

try {
      String line;
      while ((line = in.readLine()) != null)
            { process received data }
}
catch (InterruptedException)
{
      System.out.println(
            "The socket timeout has been reached.");
}
```


Socket Timeout Limitations

- You need to have a Socket object established to call setSoTimeout(), but the Socket constructor will block until the socket is initially connected
- Creating a timeout on the socket constructor was added in Java 2 1.3
  - If we didn't have this ability, the best solution would be to attempt to construct the socket in a separate thread and then wait for that thread to either complete or timeout

Socket Construction Timeouts

- Construct a new class **SocketOpener**
- a static method openSocket()
  - Attempts to open a stream socket to a specified host and port number for up to a specified timeout interval
  - Returns a socket or a null Socket reference


```
class SocketOpener implements Runnable {
      private String host; private int port;
      private Socket socket;
      public static Socket openSocket(String host,
         int port, int timeout) {
            SocketOpener o = new SocketOpener(host, port);
                  Thread t = new Thread(o);
            t.start();
            try { t.join(timeout); }
            catch (InterruptedException exp) { }
            return o.getSocket(); }
      public SocketOpener(String host, int port) {
            socket = null;
            this.host = host; this.port = port; }
      public void run() {
            try {
                  socket = new Socket(host, port);
            } catch (InterruptedException exp) { } }
      public Socket getSocket()
            { return socket; }
}
```

The SocketOpener Class

- When openSocket() is called, the SocketOpener class starts a new thread that attempts to open the socket
  - new thread calls the blocking constructor for the socket
- The main thread calls join() on the new thread causing it to wait
  - the new thread to complete, returning the newly created socket
  - the join timeout to expire, returning null for the socket reference

The SocketOpener Class

- User-defined SocketOpener class successfully implements a timeout-enabled socket creation mechanism
  - Provides the ability to timeout a socket construction
    - Not natively possible in the Java language
- This class can be very useful if a connection to the server may not be successfully established

Higher Level Network Programming

- Socket-level programming is quite powerful because your application directly controls what is sent and directly receives what is received
  - not very convenient to transfer network files using a well-known protocol such as HTTP
  - your application would need to implement the HTTP protocol (i.e., generate headers and correctly format the HTTP packets)
- Java has URL-based network communications

URL: Uniform Resource Locator

- URLs have two main components
  - the protocol name
  - the resource name
- To URL for the file named index.html on the host java.sun.com and that HTTP should be used to retrieve it

```
http://java.sun.com/index.html
```

- To retrieve the same file using FTP

```
ftp://java.sun.com/index.html
```

URL Resources

- The format of the resource field in a URL is dependent on the protocol being used, but most (including HTTP) include the following components
  - host name the resource is located on
  - filename (full path to the resource on the host)
  - port number to connect to (typically optional)
    - HTTP default: 80
  - a reference (such as a tag in an HTML file)

Creating a URL Object

- Java abstracts the URL in the **URL** class
- To create an object representing a URL, pass the string of the URL to the constructor

```
URL javaPage = new URL("http://java.sun.com");
URL file2get = new URL(
     "ftp://stimpy.eecis.udel.edu/file1.txt");
URL file2put = new URL(
     "ftp://stimpy.eecis.udel.edu/file2.txt");
```

Relative URLs

- Relative URLs are URL objects constructed relative to another URL object
- Suppose your program needs to create URL objects for these two network resources

```
http://www.cis.udel.edu/file1
http://www.cis.udel.edu/file2
```

- Create a common URL and then relative URLs for differences…

```
URL baseURL = new URL("http://www.cis.udel.edu");
URL file1URL = new URL(baseURL, "file1");
URL file2URL = new URL(baseURL, "file2");
```


Relative URLs

- Relative URLs are very useful when a certain resource has a reference contained in it
  - Example: an anchor in an HTTP document
- Suppose the index.html file at java.sun.com has an anchor named DOWNLOADS in it
  - To construct the appropriate URL

```
URL javaURL = new URL("http://java.sun.com");
URL indexURL = new URL(javaURL, "index.html");
URL downloadsURL = new URL(
      indexURL, "#DOWNLOADS");
```

URLs with Specific Ports

- Possible to construct a URL for a specific port number
- Suppose a host has two web servers, one on the traditional port 80 and another on port 8080
  - To retrieve the file index.html from the port 8080 web server

```
URL newURL = new URL("http",
  "128.4.133.74", 8080, "/index.html");
```

- Constructs the URL:

```
http://128.4.133.74:8080/index.html
```


URLs and Exceptions

- Creating a URL object can throw a **MalformedURLException** if any of the constructor arguments are null or refer to an unknown protocol
- URL construction must be placed inside a try/catch block

```
try {
      URL myURL = new URL ( . . . );
}
catch (MalformedURLException exp) {
      handle the malformed URL exception here
}
```

URL Getter/Accessor Methods

- URL class's accessor methods allow all of the information about the resource it represents to be obtained
  - getProtocol() returns the URL's protocol
  - getHost() returns the URL's host
  - getFile() returns the URL's filename
  - getPort() returns the URL's port
  - getRef() returns the URL's reference


```
URL complexURL = new URL(
      "http://128.4.133.74"
      + ":8080/CPM/grader.html#BOTTOM");

complexURL.getProtocol();
// returns "http"
complexURL.getHost();
// returns "128.4.133.74"
complexURL.getFile();
// returns "/CPM/grader.html"
complexURL.getPort();
// returns 8080
complexURL.getRef();
// returns "BOTTOM"
```

Reading Directly from a URL

- After a URL object has been created, the program can read the resource represented by the URL
  - Call openStream() to obtain an InputStreamReader object
- We can construct a URL object to index.html at www.yahoo.com
  - open an InputStreamReader on that resource (file)
  - attach a BufferedReader to that reader
  - read the index.html file
  - copy everything that is read to the standard output stream (the screen)

This program will display the contents of the file index.html located at www.yahoo.com to the default output stream.

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

public class URLReader {
      public static void main(String[] args) {
            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();
      }
}
```

Not showing the try/catch

URLReader.java

URL vs. Socket Programming

- Could write program without using the URL classes
  - parse the URL string
  - lookup the hostname
  - open a socket
  - connect to the host using the appropriate port
  - generate appropriate commands to go to the HTTP server on the host
  - open a receive stream
  - process the incoming data

URL vs. Socket Programming

- Details (specifically those of the HTTP protocol) are all handled by the URL class
  - encapsulates all of these socket-level details and programmer doesn't have to write them (or mess them up!)
- URL class knows about the HTTP and FTP protocols
  - Same approach could be used to retrieve files using the FTP protocol as well
- Also a java.net.HttpUrlConnection class

Datagram Communications

- Let's look at connectionless communications using datagrams
  - specifies the creation of individual messages, called datagrams, which are transmitted one at a time, from one host to the other

Datagram Communications

- Two primary classes deal with datagram communications
  - **DatagramPacket** and **DatagramSocket**
- No server socket class
  - No connections in datagram communications
  - packets are simply transmitted from one host to another
- To transmit a datagram, the program should first construct a DatagramPacket object and then deliver it to a DatagramSocket

Constructing a Datagram

- Call the constructor for DatagramPacket, passing it four pieces of data:
  - a byte array of data to be transmitted
  - the number of bytes of data
  - the internet address of the host to send the data to
  - the port number to send the data to on the receiving host

Constructing a Datagram

- To construct a datagram packet with
  - data contained in a byte array named byte_array
  - of 50 bytes
  - transmitted to a host with hostname "localhost" at port number 1998

```
InetAddress client_addr = InetAddress.getByName(
     "localhost");
DatagramPacket DGtobeSent = new DatagramPacket(
     byte_array,
     50,
     client_addr,
     1998);
```

Constructing a Datagram Socket

- Call the constructor by passing it the port number

```
DatagramSocket sck1 = new DatagramSocket(1998);
```

- A datagram socket can also be constructed without passing it a port number
  - allows the system to pick a random port number…

```
DatagramSocket sck1 = new DatagramSocket();
```


Sending a Datagram Packet

- After a datagram socket has been constructed, datagram packets can be sent using this socket.
- Call send() on the datagram socket with the datagram packet that is to be sent…

```
// send the datagram packet using
// the datagram socket
sck1.send(DGtobeSent);
```

Receiving a Datagram Packet

- Datagram socket can also receive a datagram packet
- Construct a DatagramPacket object
- Call receive() on the datagram socket
  - receives a datagram packet directed to the port associated with the socket
  - copies the received packet into the specified DatagramPacket object

Receiving a Datagram Packet

```
// open a socket
DatagramSocket sck1 = new DatagramSocket(1998);

// setup the packet
byte buffer[] = new byte[ 1000 ];

DatagramPacket received = new DatagramPacket(
     buffer, buffer.length);

// wait for a packet to arrive
sck1.receive(received);

// now, process the packet
```

Example: An Echo Server

- Write a program that will listen for datagram packets on port 1998 and then echo the packets back to the host they originated from


```
DatagramSocket sck1 = new DatagramSocket(1998);
while(true) {
      try {
        // receive a datagram packet
        byte buffer[] = new byte[500];
        DatagramPacket received = new DatagramPacket(
            buffer, buffer.length);
        sck1.receive(received);

        // create an echo packet and sent it
        DatagramPacket tobeSent = new DatagramPacket(
            received.getData(),
            received.getLength(),
            received.getAddress(),
            received.getPort());
        sck1.send(tobeSent);
      } catch (IOException exp1) {
        System.out.println("Error:" + exp1);
      }
}
```

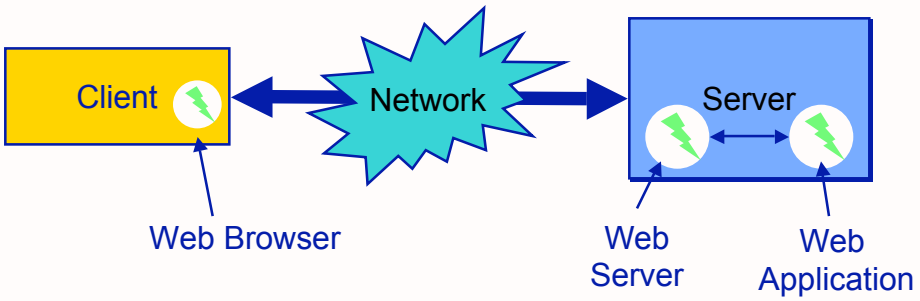
More General URL Connections

- Reading URLs is very simple but becomes more complex when we want to do both reading and writing to a URL
- Why would we want to write to a URL?

Dynamic Web Pages

- Returned document depends on parameters passed from user
- A user fills out a form to send data to a web server
  - User sees a number of text boxes, combo boxes, and other UI components on a web page
  - When the user clicks the "Submit" button, the contents of the text fields and the settings of the combo, check, and radio buttons, are sent to the server to be processed by a program
- The script or application on the server that should process this data is indicated in the ACTION attribute of the HTML FORM tag

Dynamic Web Pages



- Web browser: makes requests, renders responses
- Web Server: handles static requests
- Web Application: handles dynamic requests

Server-side Processing

- Many alternatives for server-side processing
  - CGIs (Common Gateway Interface), written in C or Perl
  - PHP (Hypertest Preprocessor)
  - ASPs (Active Server Page)
  - JSPs/Servlets (more next week)
- Same handling process
  - When the server receives the form data, it launches the appropriate server-side script/program to handle the data
  - Script processes the form data and produces another HTML page that the web server sends back to the browser
  - Response page can contain new information (usually based on the form data received) or could simply be an acknowledgement

Data / Parameters

- Data / parameters are always passed to scripts in a name and value pair (similar to a hashtable)
  - userid = user
  - passwd = pswd
- Let's look at the two different methods of transmitting these parameters to the server

The GET Method

- Attaches the parameters to the end of the URL
  - Parameter list starts with a '?'
  - Individual parameters are separated by a '&'
- **URL encoding**: Need to encode the parameters so that requests work correctly in the HTTP protocol
- To encode the parameters
  - Replace any spaces with a '+'
  - Replace all non-alphanumeric characters with a '%' followed by the 2-digit hexadecimal ASCII code of the character

The GET Method

- To encode the parameter bookname equal to “Mastering C++” and the parameter location equal to “Newark DE” and pass these parameters to script.pl at server.udel.edu

```
http://server.udel.edu/script1.pl?bookname=
 Mastering+C%2b%2b&location=Newark+DE
```

- Parameter list starts with a ‘?’
- Individual parameters are separated by a ‘&’
- Handled by your browser automatically

Java Programming and GET

- Construct a URL object that represents the script and encoded the parameters in the URL
- Connect to that URL and read from it as before
- For example, to code the previous example and display the returned page…

This program uses the GET method and then reads what the script returns and displays it to the standard output stream.

```
public static void main(String[] args)
{
      URL script = new URL(
            "http://server.udel.edu/script1.pl");
      URL sendToScript = new URL(
            script,
            "?bookname=" + "Mastering+C%2d%2d"
            + "&location=" + "Newark+DE");

      BufferedReader in = new BufferedReader(
            new InputStreamReader(
            sendToScript.openStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
      in.close();
}
```


The POST Method

- Opens an output stream to the URL connection and writes name/value pairs to the stream
  - a connection is established to the resource represented by the URL (the script)
  - the name/value pairs are sent to the script through an output stream, which transmits the parameters to the script

Opening a URLConnection

- When a URL object is created, no connection is made
  - When the openStream() method of the URL class is called, a connection is made to the resource represented by the URL and then an InputStream is constructed and returned
  - Allows the client to receive information from the resource (the web page or script)

Opening a URLConnection

- If more than an InputStream is required, a connection must first be manually established
  - Call openConnection() of the URL class
  - Returns a **URLConnection** class object, which is an abstraction of an open, established, connection
- To open a connection to the example script

```
URL script = new URL (
      "http://server.udel.edu/script1.pl");
URLConnection connection =script.openConnection();
```

Getting an InputStream

- Get the InputStream using getInputStream() on the URLConnection object
  - URLConnection represents an open connection
- Equivalent code:

```
URL script = new URL(
      "http://server.udel.edu/script1.pl");
InputStream in = script.openStream();


URL script = new URL(
      "http://server.udel.edu/script1.pl");
URLConnection openConn = script.openConnection();
InputStream in = openConn.getInputStream();
```


Getting an Output Stream

- Need to construct a URLConnection object when we need an OutputStream
  - used to allow the client to send data to the server, as in the case of server-side scripts
- Construct the URLConnection object (which connects to the resource) and then call getOutputStream()
  - returns an OutputStream which allows the client program to send data to the server-side script

The POST Method

- The POST method sends CGI parameters to the script using this approach
- The POSTed data must be URL encoded and be separated using the '&' character.
- Suppose script2.pl is also on server.udel.edu and it is the same as script1.pl, except that it receives parameters via POST not GET
- We can modify our example program to do this…

This program uses the POST method and then reads what the script returns and displays it to the standard output stream.

```
public static void main(String[] args)
{
      URL script = new URL(
            "http://server.udel.edu/script2.pl");
      URLConnection openConn = script.openConnection();
      PrintWriter out = new PrintWriter(
            openConn.getOutputStream());
      out.print("bookname=" + "Mastering+C%2d%2d" + "&");
      out.print("location=" + "Newark+DE" + "\n");
      out.close();
      BufferedReader in = new BufferedReader(
            new InputStreamReader(
            openConn.getInputStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
      in.close();
}
```

URL Encoding with **URLEncoder**

- You/ your program does not have to manually replace spaces and non-alphanumeric characters
- The **URLEncoder** class has a static method encode()
  - takes a String
  - returns a URL encoded String
- We can modify our program to interpret the first command-line argument as the bookname and the second command-line argument as the location, instead of using hard-coded fields

This program now sends the first two command-line arguments, correctly URL encoded, as the parameters to the script.

```
public static void main(String[] args)
{
   URL script = new URL(
      "http://server.udel.edu/script2.pl");
   URLConnection openConn = script.openConnection();
   PrintWriter out = new PrintWriter(
      openConn.getOutputStream());
   out.print("bookname=" + URLEncoder.encode(args[0]) + "&");
   out.print("location=" + URLEncoder.encode(args[1]) + "\n");
   out.close();
   BufferedReader in = new BufferedReader(
      new InputStreamReader(
      openConn.getInputStream()));
   String inputLine;
   while ((inputLine = in.readLine()) != null)
      System.out.println(inputLine);
   in.close();
}
```

A Web Server

- Receives GET/POST requests from users
- Processes requests
  - Given to appropriate application to handle
    - PHP, ASP, Java Servlet Container, …
  - Handles static requests by sending document to requestor
    - Or appropriate error message if the file does not exist or the file does not have appropriate read permissions

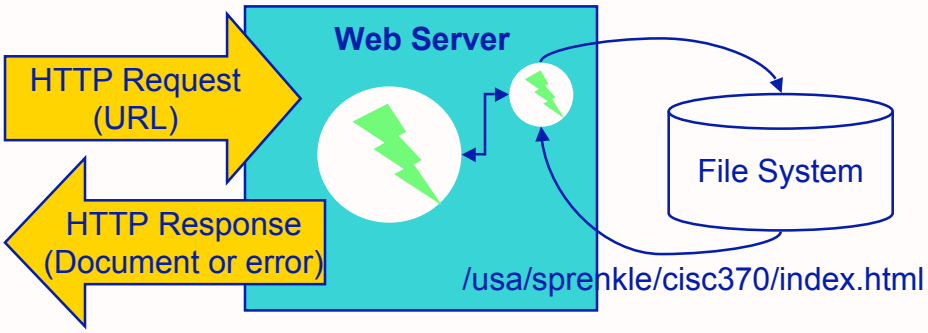
A Web Server: Handling Requests

- Has one thread per client to handle request
  - Limit on number of threads, as discussed
- Serves files from some directory
  - My web-accessible files are in /usa/sprenkle/public_html
  - But users access with resource name ~sprenkle
  - Server maintains mapping from ~sprenkle to appropriate location

Web Server

- At a high-level

Request: GET /~sprenkle/cisc370



Your Multithreaded Web Server

- Basically implement what's on the previous slide!