Outline of Topics

- Networking Concepts
  - Internet Addresses and ports
  - Packets and TCP/IP
  - Sockets
- Java Classes
  - InetAddress
  - Socket and ServerSocket
  - URL and URLConnection
  - DatagramSocket and DatagramPacket
- Threading and networking
- Firewall Issues
Internet Addresses

- Every computer represented by (at least one) internet address (IP address)
  - Can use quad bytes or easier-to-remember domain names
    - 131.94.126.81 (32 bit quad-bytes)
    - ocelot.aul.fiu.edu (domain name)
  - Can use special name and address for local machine
    - 127.0.0.1
    - localhost
  - Machines can have multiple names/addresses
  - Some addresses represent multiple machines; useful for load-balancing

Ports

- Abstraction to allow multiple programs access to single internet connection
- Ports are numbered 0 to 65,535
- Ports 0 to 1,023 are reserved and often restricted
- Well known-ports:
  - 7 echo
  - 13 time of day
  - 21 ftp
  - 23 telnet
  - 80 http
Socket

- Can communicate with another machine by opening a socket.
- If the other machine is listening on the port, you can establish a two-way connection. Listening at other end is done with a server-side socket.
- Example: connect to www.cs.harvard.edu at port 13 and get time of day
  - Can do this in a telnet window
  - Can do this using browser and URL http://www.cs.harvard.edu:13/

TCP/IP vs UDP

- **UDP**
  - Packets are sent from one machine to another
  - Packets may take different routes
  - Packets may arrive in random order or not at all
  - You must set up a protocol that numbers packets, and sends out a header. Receiver can request retransmission of missing packets, and reassemble
  - Faster but more work, good if loss of packets is tolerable (Real Audio)

- **TCP/IP**
  - Does all the dirty work. Looks like a smooth stream.
The Java Socket Class

- Defined in java.net package
- Create a socket by specifying the machine and port to connect to.
- Once you have a socket, can use `getInputStream` and `getOutputStream` to read from and write into the socket.
  - Returns `InputStream` and `OutputStream`
  - Actual types are invisible `SocketInputStream` and `SocketOutputStream`
  - Should always turn off buffering on `PrintWriter`
- Old I/O rules apply once you have streams

Getting the Time of Day

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

class Time {
    public static void main(String[] args) {
        try {
            Socket t = new Socket("www.cs.harvard.edu", 13);
            InputStream in = t.getInputStream();
            InputStreamReader rin = new InputStreamReader(in);
            BufferedReader bin = new BufferedReader(rin);

            String str;
            while((str = bin.readLine()) != null)
                System.out.println(str);
        }
        catch( IOException e ) {
            System.out.println(e);
        }
        // should close socket in finally block
    }
}
```
**InetAddress**

- Abstracts the idea of an IP address
  - Some constructors require `InetAddress` instead of a string
  - Using `InetAddress` avoids repeated lookup of same IP address; can save time
- **InetAddress** is a factory class; no constructors
  - Use `getByName` static method
  - Use `getLocalHost` (gives real info, not localhost)
    - Can then use instance methods `getHostName`, `getHostAddress`

**ServerSocket**

- Used to listen on a *local* port
- Constructor specifies port to listen on
- Cannot listen on a port that is already being listened on
- After creation, call `accept`.
  - Blocks until a connection comes in
  - When connection occurs, accept returns a `Socket`
  - At that point, you can get a pair of streams and communicate with the client
  - Should close inside finally block when done
import java.io.*;
import java.net.*;

class EchoServer {
    public static void main(String[] args) {
        Socket sock = null;
        try {
            ServerSocket ss = new ServerSocket(3737); // Use port 3737
            sock = ss.accept();
            InputStreamReader in = new InputStreamReader(sock.getInputStream());
            BufferedReader is = new BufferedReader(in);
            PrintWriter os = new PrintWriter(sock.getOutputStream(), true);

            os.println("Welcome to the EchoServer!");
            os.println("Enter "+"***"+" to exit");

            String str;
            while ((str = is.readLine()) != null && !str.trim().equals("***"))
                os.println(str);
        } catch (IOException e) { /* Could write to a log file */ }
        finally { /* close stuff here */ }
    }
}

Problem With Previous Example

● Can only handle one connection
● Once accept is called nobody is listening on the port any more
  – future connect attempts will fail
  – could use a loop to call accept again after connection is processed
    ● will allow connections indefinitely
    ● will only allow one at a time, however
● Solution: use threads!
Threads And Networking

- **main thread**
  - creates the `ServerSocket`
  - has a tight loop listening for a connection
  - when a connection comes in, main thread spawns a background thread
    - passes the socket to the background thread
    - background thread processes the connection
    - main thread resumes listening for a connection
    - multiple simultaneous connections possible, subject to system limits
    - main thread runs indefinitely
    - main thread could keep a shared list of all background threads it has spawned: chatroom possibilities!

---

A Better Echo Server: Main

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

public class BetterEchoServer
{
    public static void main( String [] args )
    {
        ServerSocket ss = null;
        try {
            ss = new ServerSocket( 3737 );
            while( true )
            {
                Socket sock = ss.accept();
                Thread t = new EchoHandler( sock );
                t.start();
            }
        }
        catch( IOException e ) { /* Could write to a log file */ }
        finally { /* close stuff here */ }
    }
}
```
The Background Handler Thread

class EchoHandler extends Thread {
    private Socket sock;

    public EchoHandler( Socket incoming ) {
        sock = incoming;
    }

    public void run() {
        try {
            InputStreamReader in = new InputStreamReader( sock.getInputStream() );
            BufferedReader is = new BufferedReader( in );
            PrintWriter os = new PrintWriter( incoming.getOutputStream(), true );

            os.println( "Welcome to the EchoServer!" );
            os.println( "Enter " + "***" + " to exit" );

            String str;
            while( ( str = is.readLine() ) != null && !str.trim().equals( "***" ) )
                os.println( str );
        } catch( IOException e ) { /* Could write to a log file */ }
        finally { /* close stuff here */ }
    }
}

Datagrams

● Models UDP transmission
● Create a DatagramSocket, and use it to send or receive a DatagramPacket.
● DatagramPacket contains
  − The InetAddress of the other party
    ● Either you set it to send initially
    ● receive will fill it in, so you reuse the packet to reply
  − The port of the other party
  − The bytes to send (including how many)
  − Basically a C++-style struct with a bunch of sets and gets.
Connecting to Real Echo Server

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

class EchoClient {
    public static void main(String[] args) {
        DatagramSocket sock = null;
        String oneLine = null;
        try {
            InetSocketAddress remoteIP = InetAddress.getByName("www.cs.harvard.edu");
            BufferedReader bin = new BufferedReader(new InputStreamReader(System.in));
            sock = new DatagramSocket(); sock.setSoTimeout(5000);
            System.out.println("you> ");
            while((oneLine = bin.readLine()) != null) {
                byte[] msg = oneLine.getBytes();
                sock.send(new DatagramPacket(msg, msg.length, remoteIP, 7));
                byte[] reply = new byte[msg.length];
                DatagramPacket replyPack = new DatagramPacket(reply, reply.length);
                sock.receive(replyPack);
                System.out.println("echo> " + new String(reply) + "\you> ");
            }
        } catch (IOException e) { /* Print some messages */ }
        finally { /* Close the socket */ }
    }
}
```

Uniform Resource Locators (URLs)

- **Represent web resources**
  - good idea to have / at end of directory URLs!!!!
  - `file:/.dir/foo.txt`
  - `https://www.itn.net/

- **Consists of**
  - protocol (http, file, ftp, https, etc.)
  - IP address
  - port (optional; defaults exist)
  - resource
Java Classes

- **URL**
  - abstracts the notion of a URL
  - supports http, ftp, file
  - https ok in browser if you download JSSE

- **URLConnection**
  - abstract class abstracts the notion of a connection
  - can optionally set request headers
  - then make connection
  - then optionally get returned header info
  - then access resource with both input stream and outputstream (for instance, to post forms)
  - can define your own protocols

Getting A Text-Based Web Page

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

class GetWebPage {
    public static void main(String[] args) {
        try {
            URL url = new URL(args[0]);
            URLConnection urlconn = url.openConnection();
            
            String str = null;
            if (urlconn.getContent() instanceof InputStream) {
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(urlconn.getInputStream()));
                
                while ((str = in.readLine()) != null) {
                    System.out.println(str);
                }
            } catch (IOException e) {
                System.out.println(e);
            }
        }
    }
}
```
Firewalls and Security

- Most corporate environments will not allow you to directly open sockets to sites outside the corporate network (or maybe even inside)
- Usually access is controlled by a proxy server
  - You open connection to the proxy server
  - If proxy likes you, it forwards the request on your behalf
  - Typically proxy server will allow only http requests to acceptable hosts and will deny most others
    - Proxy servers are more suspicious than Java VMs!

http Connecting Through The Proxy

- Need undocumented magic for http
  - After you have a URLConnection object, set some header data before making actual connection via `connect` or `getInputStream`.

```java
URLConnection conn = url.openConnection();
System.getProperties().put( "proxySet", "true" );
System.getProperties().put( "proxyHost", PROXY_HOST );
System.getProperties().put( "proxyPort", PROXY_PORT );

// If proxy server requires authentication, add next three lines.
// username and password will be what proxy verifies.
String proxyAuth = "Basic " + new sun.misc.BASE64Encoder().encode( "username:password".getBytes() );
conn.setRequestProperty( "Proxy-Authorization", proxyAuth );
```
Password-Protected Web Pages

- Can access password protected pages by including an additional property in the header

```java
// name and pwd will be what web page verifies.
String webAuth = "Basic " + new sun.misc.BASE64Encoder().encode("name:pwd".getBytes());
conn.setRequestProperty("Authorization", webAuth);
```

ftp Connections

- ftp protocol is supported in Java (you get an invisible FtpURLConnection)
- Password protected pages accessed via standard ftp URL (works in browsers too)

  ftp://username:password@ftp.imdb.com/

- If you need to tunnel through proxy servers, use same idea as http, but with replacements:

```java
System.getProperties().put("ftpProxySet", "true");
System.getProperties().put("ftpProxyHost", PROXY_HOST);
System.getProperties().put("ftpProxyPort", PROXY_PORT);
```
Secure Connections

- **https** will work if you are inside a browser
  - browser’s VM has anHttpsURLConnection implementation
  - complications if the applet the browser is running was loaded from the network
- Needs Java 1.2 or later (thus can use `setProperty`)
- Then, add to your code:
  ```java
  System.setProperty( "java.protocol.handler.pkgs", 
                      "com.sun.net.ssl.internal.www.protocol" );
  Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider( ) );
  ```

Summary

- Networking in Java is basically easy
  - `ServerSocket` and `Socket` for TCP/IP connections
  - `DatagramSocket` and `DatagramPacket` for UDP connections
  - `URL` and `URLConnection` for basic protocols
- In most corporate environments, the only reliable way to communicate is through an http request that tunnels through a proxy server.