Java Networking with Sockets

Online Java IV Lecture 7

What is a socket?

- ✓ **Stream connecting processes running in different address spaces**
  - ❑ Can be across a network or on the same machine.
- ✓ **We say *create a socket connection between machine A and machine B*.**
  - ❑ This means, roughly, create input and output streams for sending data between programs running simultaneously on each machine.
- ✓ **The programs can then *talk to each other.***
- ✓ **In Java this is lowest-level form of communication from application developer's view**

Sockets, cont.

- **Sockets represent a low-level abstraction for application communication.**
  - Programmer is aware of a stream that connects two computers.
  - Programmer fully responsible for managing and interpreting flow of bytes between computers

- **Higher-level techniques**
  - message passing systems (MPI, SOAP, JMS),
  - extensions to web servers (ASP, JSP, servelets, etc),
  - distributed objects (CORBA, RMI), web services, etc.

More about sockets in Java

- ✓ **One of the good things about Java**
- ✓ **Supported in the standard language (j2sdk)**
- ✓ **Distinction between high and low-level blurred somewhat by ability to wrap streams (e.g. ObjectOutputStream)**
- ✓ **Still, socket programming differs from other distributed programming in its low-level nature.**

Why is this paradigm useful?

- ✓ **Shared resources (web servers, ftp servers, mail servers)**
- ✓ **Online auctions, exchanges, etc.**
- ✓ **Data locality**
- ✓ **Localize computing power**
- ✓ **Crash protection**
- ✓ **Software maintainability**

Conceptual overview of basic client-server program

- ✓ **Write a program that dials up another program at a specified IP address running on a specified port. Call this program the client.**
- ✓ **Second program – server – accepts connection and establishes input/output stream to client.**
- ✓ **When server accepts, client can establish input/ouput stream to server**
- ✓ **Client makes request of server by sending data. Server sends replies to client. Protocol must be defined so client/server understand can interpret messages.**

Conceptual overview of basic peer-to-peer program

- ✓ **Two processes running on specific port of specific machine.**
- ✓ **Either process can dial up the other process.**
- ✓ **When connection is established, applications talk at a peer level, each making requests of each other, rather than one making requests and the other serving up those requests.**
- ✓ **Will see many examples soon.**

Socket Machinery in Java

Java classes for direct socket programming

- ✓ **Good news: This is very simple in Java**
- ✓ **Really only 3 additional classes are needed**
- ✓ java.net.InetAddress
- ✓ java.net.Socket
- ✓ java.net.ServerSocket

Important class, cont.

- **java.net.InetAddress**
  - static InetAddress getByName(String name)
    - given a hostname name, return the InetAddress object representing that name (basically encapsulates name and IP associated with name);
  - static InetAddress[] getAllByName(String name)
    - same as above but for case where many ip's mapped to single name (try www.microsoft.com, e.g.).
  - static InetAddress getLocalHost()
    - get InetAddress object associated with local host.
  - static InetAddress getByAddress(byte[] addr)
    - get InetAddress object associated with address addr

Most important classes/methods

- **java.net.Socket**
  - Socket(InetAddress addr, int port);
    - create a Socket connection to address addr on port port
  - InputStream getInputStream();
    - returns an instance of InputStream for getting info from the implicit Socket object
  - OutputStream getOutputStream();
    - returns an instance of OutputStream for sending info to implicit Socket object.
  - close();
    - close connection to implicit socket object, cleaning up resources.

Important classes, cont.

- **java.net.ServerSocket**
  - ServerSocket(int port);
    - enables program to listen for connections on port port
  - Socket accept();
    - blocks until connection is requested via Socket request from some other process. When connection is established, an instance of Socket is returned for establishing communication streams.

Error Handling

- ✓ **Very important to ensure that server is robust and will not crash.**
- ✓ **Important Exceptions:**
  - ❑ InterruptedIOException
  - ❑ ConnectException
- ✓ **Be sure to close your sockets either after a crash or upon expected completion.**

Examples

- ✓ **Best way to learn this is to study several canonical examples**
- ✓ **See many simple course examples in lec07_code.zip**
  - ❑ Examples presented in the slides
  - ❑ Available as executable applications you should download and experiment with

Messages/Protocols

What is a message?

- ✓ **Technically, a structured piece of info sent from one agent to another.**
- ✓ **Can be thought of as a set of commands with arguments that each agent understands and knows how to act upon.**
- ✓ **Groupings of such commands are commonly referred to as a "protocol".**
- ✓ **HTTP, FTP, etc. are all protocols (get, put, ...)**

Why write our own message-passing system?

- ✓ **Existing protocols might be totally inappropriate for the needs of an application.**
- ✓ **An existing protocol may work but be too inefficient (e.g. SOAP).**
- ✓ **In general, can fine-tune the protocol exactly to your application to minimize memory and bandwidth overhead.**

What about distributed objects?

- ✓ **Can be overkill when communication needs are simple.**
- ✓ **Can be inefficient when transaction throughput is critical.**
- ✓ **Rapid implementation takes precedence of sophisitication/flexibility**
- ✓ **special network protocols need to be avoided (behind a firewall, etc.)**
- ✓ **CORBA, RMI, etc. not available**

Architecting a Message Passing System

- **Crucial point:**
  - Isolate communication details from application details.
    - Your stand-alone objects should be well-defined and unaware of the message passing environment.
  - Provide a structured way to link messages to method calls on these objects.

- **Asynchronous vs. Synchronous Message Handling**
  - Synchronous: each agent waits for response after sending their message, then does work – “handshaking”.
  - Asynchronous (typical) – work needs to be done after sending message. Doesn’t know when reply will come, but single thread busy so can’t process message.

PORT NUMBERS

- **The Internet Assigned Numbers Authority**
  - http://www.iana.org/assignments/port-numbers
  - From the version noted *(last updated 2008-12-10)*
- **The port numbers are divided into three ranges:**
  - the Well Known Ports,
  - the Registered Ports
  - the Dynamic and/or Private Ports.
- **The Well Known Ports are those from 0 through 1023.**
  - DCCP Well Known ports SHOULD NOT be used without IANA registration. The registration procedure is defined in [RFC4340], Section 19.9.
- **The Registered Ports are those from 1024 through 49151**
  - DCCP Registered ports SHOULD NOT be used without IANA registration. The registration procedure is defined in [RFC4340], Section 19.9.
- **The Dynamic and/or Private Ports are those from 49152 through 65535**

Getting Information From the Net

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

public class ParseURL
{
   public static void main(String[] args) throws Exception
   {
      URL aURL = new URL("http://java.sun.com:80/docs/books/"
                         + "tutorial/index.html#DOWNLOADING");

      System.out.println("protocol = " + aURL.getProtocol());
      System.out.println("host = " + aURL.getHost());
      System.out.println("filename = " + aURL.getFile());
      System.out.println("port = " + aURL.getPort());
      System.out.println("ref = " + aURL.getRef());
      System.out.println("contents = " + aURL.getContent());

   }
}
/* ************
protocol = http
host = java.sun.com
filename = /docs/books/tutorial/index.html
port = 80
ref = DOWNLOADING
********** */
```

Reading from the Net

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

public class URLReader
{
   public static void main(String[] args) throws Exception
   {
      URL myJava = new URL("http://www.ucsd.edu");

      BufferedReader in = new BufferedReader(
                              new InputStreamReader(
                                  myJava.openStream()));

      String inputLine;

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

      in.close();
      System.in.read();
   }
}
```

Establishing a Connection

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

public class URLConnectionReader
{
    public static void main(String[] args) throws Exception
    {
        URL myJava = new URL("http://java.sun.com:80/docs/books/"
                        + "tutorial/index.html#DOWNLOADING");
        URLConnection theConnection = myJava.openConnection();
        BufferedReader in = new BufferedReader(
                                    new InputStreamReader(
                                        theConnection.getInputStream()));
        String inputLine;

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

        System.in.read();
    }
}
```

InetAddress

```
import java.net.*;

public class Resolver {
  public static void main( String args[] ) {
    InetAddress ipAddr;

    try {
      ipAddr = InetAddress.getByName( args[0] );
      System.out.print( "IP address = " + ipAddr + "\n " );
    }
    catch ( UnknownHostException ex ){
      System.out.println( "Unknown host " );
    }
  }
}
```

```
V:\UCSD\Java\On-line Java IV\Lesson 7>java Resolver yahoo.com
IP address = yahoo.com/206.190.60.37
```

Sending and Receiving Data Packets

```
public class QuoteClient {
    public static void main(String[] args) throws IOException {

        if (args.length != 1) {
             System.out.println("Usage: java QuoteClient <hostname>");
             return;
        }
        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);

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

         // display response
         String received = new String(packet.getData(), 0);
         System.out.println("Quote of the Moment: " + received);

         socket.close();
    }
}
```

Constructs a datagram socket and binds it to any available port on the local host machine.

Sends a datagram packet from this socket. The DatagramPacket includes the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.

Connectionless Server

- ✓ **The server must be started before the client attempts to make a connection**

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

Server Running as a Separate Thread (1 of 3)

```
public class QuoteServerThread extends Thread {

    protected DatagramSocket socket = null;
    protected BufferedReader in = null;
    protected boolean moreQuotes = true;

    public QuoteServerThread() throws IOException {
        this("QuoteServerThread");
    }

    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("Could not open quote file.");
            System.err.println("Serving time instead.");
        }
    }
```

Server Running as a Separate Thread (2 of 3)

```
public void run() {
        while (moreQuotes) {
            try {
                byte[] buf = new byte[256];
                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();
                     // 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();
    }
```

Server Running as a Separate Thread (3 of 3)

```
protected String getNextQuote() {
        String returnValue = null;
        try {
            if ((returnValue = in.readLine()) == null) {
               in.close();
               moreQuotes = false;
               returnValue = "No more quotes. Goodbye.";
            }
        } catch (IOException e) {
            returnValue = "IOException occurred in server.";
        }
        return returnValue;
    }
}
```

Create Our Own Protocol (client 1 of 2)

```
public class KnockKnockClient {
    public static void main(String[] args) throws IOException {

        Socket kkSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;
       String laptopName = "BLULAD770100";

        try {
            kkSocket = new Socket(laptopName, 4444);
            out = new PrintWriter(kkSocket.getOutputStream(), true);
            in = new BufferedReader(
                        new InputStreamReader(kkSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: "
                                                    + laptopName + ".");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: “
                                                    + laptopName + ".");
            System.exit(1);
        }
```

Creates a stream socket and connects it to the specified port number on the named host.

Create Our Own Protocol (client 2 of 2)

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

        while ((fromServer = in.readLine()) != null) {
            System.out.println("Server: " + fromServer);
            if (fromServer.equals("Bye."))
                break;

            fromUser = stdIn.readLine();
           if (fromUser != null) {
                System.out.println("Client: " + fromUser);
                out.println(fromUser);
            }
        }
        out.close();
        in.close();
        stdIn.close();
        kkSocket.close();
    }
}
```

Once the streams are created, reading and writing from and to a socket is just like reading and writing from and to a file

Create Our Own Protocol (Server 1 of 2)

```
public class KnockKnockServer {
    public static void main(String[] args) throws IOException {

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

        Socket clientSocket = null;
        try {
            clientSocket = serverSocket.accept();
        } catch (IOException e) {
            System.err.println("Accept failed.");
            System.exit(1);
        }
```

A server socket waits for requests to come in over the network. It performs some operation based on that request, and possibly returns a result

Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made. A new Socket s is created

Create Our Own Protocol (Server 2 of 2)

```
        PrintWriter out =
                    new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                clientSocket.getInputStream()));
        String inputLine, outputLine;
        KnockKnockProtocol kkp = new KnockKnockProtocol();

        outputLine = kkp.processInput(null);
        out.println(outputLine);

        while ((inputLine = in.readLine()) != null) {
             outputLine = kkp.processInput(inputLine);
             out.println(outputLine);
             if (outputLine.equals("Bye."))
                break;
        }
        out.close();
        in.close();
        clientSocket.close();
        serverSocket.close();
    }
```

Our Protocol (1 of 4)

A good place to use an Enum

```
public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;

    private static final int NUMJOKES = 5;

    private int state = WAITING;
    private int currentJoke = 0;

    private String[] clues = { "Turnip", "Little Old Lady", "Atch",
                                                    "Who", "Who" };
    private String[] answers = { "Turnip the heat, it's cold in here!",
                                 "I didn't know you could yodel!",
                                 "Bless you!",
                                 "Is there an owl in here?",
                                 "Is there an echo in here?" };
```

Our Protocol (2 of 4)

```
public String processInput(String theInput) {
        String theOutput = null;

        if (state == WAITING) {
            theOutput = "Knock! Knock!";
            state = SENTKNOCKKNOCK;
        } else if (state == SENTKNOCKKNOCK) {
            if (theInput.equalsIgnoreCase("Who's there?")) {
                theOutput = clues[currentJoke];
                state = SENTCLUE;
            } else {
                theOutput = "You're supposed to say \"Who's there?\"! " +
                            "Try again. Knock! Knock!";
            }
```

Our Protocol (3 of 4)

```
} else if (state == SENTCLUE) {
       if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
           theOutput = answers[currentJoke] + " Want another? (y/n)";
           state = ANOTHER;
       } else {
           theOutput = "You're supposed to say \"" +
            clues[currentJoke] +
                   " who?\"" +
                   "! Try again. Knock! Knock!";
           state = SENTKNOCKKNOCK;
       }
```

Our Protocol (1 of 4)

```
} else if (state == ANOTHER) {
            if (theInput.equalsIgnoreCase("y")) {
                theOutput = "Knock! Knock!";
                if (currentJoke == (NUMJOKES - 1))
                    currentJoke = 0;
                else
                    currentJoke++;
                state = SENTKNOCKKNOCK;
            } else {
                theOutput = "Bye.";
                state = WAITING;
            }
        }
        return theOutput;
    }
}
```

Connecting to Multiple Clients

Uses the same client and the same protocol as the single threaded server

```
public class KKMultiServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = null;
        boolean listening = true;
        System.out.println("Server starting...");

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

        while (listening)
              new KKMultiServerThread(serverSocket.accept()).start();

        serverSocket.close();
    }
}
```

accept() blocks until a client connects. Every time a client connects to the server, a new thread is created

Connecting to Multiple Clients

```
public class KKMultiServerThread extends Thread {
    private Socket socket = null;

    public KKMultiServerThread(Socket socket) {
        super("KKMultiServerThread");
        this.socket = socket;
    }
```

Connecting to Multiple Clients

```
public void run() {
  try {
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
                                  new InputStreamReader(
                                  socket.getInputStream()));
    String inputLine, outputLine;
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);
    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye"))
            break;
    }
    out.close();
    in.close();
    socket.close();
  } catch (IOException e) {
      e.printStackTrace();
  }
  }
}
```

Summary

- ✓ **Java provides a rich set of classes for networking**
- ✓ **Explore the sample code presented here and in the textbook**