Socket programming
Building web applications in Java

CS242 Computer Networks
Department of Computer Science
Wellesley College

Socket programming
Let's turn some of our protocols into applications.

There are two types of network applications:
1. Implementations of a protocol standard (RFC); and
2. Proprietary network applications (do not conform to any existing RFCs).*

Java version.

*Take care not to use one of the well-known ports.

Recall that ...

- A socket is a door between an application-layer process and the transport layer protocol.
- The choice of socket depends on which transport layer protocol best suits the applications needs.

TCP connection sockets

- The client process initiates a TCP connection to a server which is listening for visitors on a welcoming ServerSocket.
- When the server hears a client knock, the welcoming socket invokes its accept() method.
- A new door, the Connection socket, is created solely for the client's use.

From the application's perspective, this connection is a direct virtual pipe between client and servers sockets.
Before going on, we need the concept of a ... stream, a sequence of characters that flow into (an input stream) or out of (an output stream) a process.

The java.io package contains classes to help us manage streams*.

---

We build a proprietary client-server application (TCP)

1. Client reads line from standard input (keyboard), sends to server via socket.
2. Server reads line from it connection socket.
3. Server converts line to uppercase, ....
4. ... and sends the modified line out its connection socket to client.
5. Client reads, prints modified line from its socket and prints the line on its standard output.

---

Client/server socket interaction: TCP

Server (running on hostid)
- create socket, port=x
- wait for incoming connection request
- read request from connectionSocket
- write reply to connectionSocket
- close connectionSocket

Client
- create socket, connect to hostid, port=x
- send request using clientSocket
- read reply from clientSocket
- close clientSocket

---

Java client (TCP)

- import java.io.*;
- import java.net.*;
- class TCPClient {
  public static void main(String argv[]) throws Exception {
    String sentence; String modifiedSentence;
    String hostid, port=x
    Socket clientSocket = new Socket(hostid, port);
    DataOutputStream toServer = new DataOutputStream(clientSocket.getOutputStream());
    BufferedReader fromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    String inputLine = fromServer.readLine();
    String uppercaseLine = inputLine.toUpperCase();
    toServer.write(uppercaseLine.getBytes());
    toServer.close();
    fromServer.close();
  }
}

*http://java.sun.com/j2se/1.4.2/docs/api
The client creates three streams and a socket

- Stream number 1 obtains input from the user.
- Stream number 2 flows out to TCP socket to server.
- Stream number 3 flows in from TCP socket to server.

Java client (TCP)

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

class TCPClient {
    public static void main(String argv[]) throws Exception {
        // Create input stream
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        // Create client socket, connect to server
        Socket clientSocket = new Socket("hostname", 6789);
        // Create output stream attached to socket
        DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
        // Send line to server
        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + '\n');
        // Read line from server
        modifiedSentence = inFromServer.readLine();
        System.out.println("FROM SERVER: " + modifiedSentence);
        // Close socket
        clientSocket.close();
    }
}
```

On the server side, ...

- The server creates a welcoming socket and waits for customers.
- When someone knocks on the door, the server creates a new socket especially for that someone.
Java server (TCP)

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

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789);
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                    new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
            clientSentence = inFromClient.readLine();
            capitalizedSentence = clientSentence.toUpperCase() + '
';
            DataOutputStream  outToClient =
                    new DataOutputStream(connectionSocket.getOutputStream());
            outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

Example concluded: Java server (TCP)

```
Create welcoming socket at port 6789
Wait, on welcoming socket for contact by client
Create input stream, attached to socket
DataOutputStream  outToClient =
new DataOutputStream(connectionSocket.getOutputStream());
Read in line from socket
clientSentence = inFromClient.readLine();
Write out line to socket
capitalizedSentence = clientSentence.toUpperCase() + 'n';
outToClient.writeBytes(capitalizedSentence);
```
Client/server socket interaction: UDP

Server (running on hostid)
- create socket, portx, for incoming request:
  `serverSocket = new DatagramSocket();`
- read request from serverSocket
- write reply to serverSocket specifying client host address, port number

Client
- create socket, clientSocket = DatagramSocket()
- Create, address (hostid, portx)
- send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket

Java client (UDP)

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

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser = 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 = inFromUser.readLine();
        sendData = sentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.send(sendPacket);
        clientSocket.receive(receivePacket);
        System.out.println("Received from server: " + receivePacket.getData().toString());
    }
}
```

Java client (UDP)

Instead of feeding bytes to a stream attached to the socket, UDP will push individual sockets through the Datagram object.

Translate hostname to IP address using DNS
- `InetAddress IPAddress = InetAddress.getByName("hostname");`
Java client (UDP) concludes

Create datagram with data-to-send, length, IP addr, port

Send datagram to server

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

Recieve datagram from server

DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);

String modifiedSentence = new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();

Java server (UDP)

No streams: Socket accepts packets from process and delivers packets to the process

Create datagram socket at port 9876

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);
    InetAddress IPAddress = receivePacket.getAddress();
    int port = receivePacket.getPort();
    String sentence = new String(receivePacket.getData());
    String capitalizedSentence = sentence.toUpperCase();
    sendData = capitalizedSentence.getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
    serverSocket.send(sendPacket);
}

Java server (UDP) concludes

ChildSocket.sendToUpperCase();
Send datagram to client

DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);

End of while loop, loop back and wait for another datagram
Out for another spin?
- Hope we’re still logged into my account on thrush and connected to cs242.
- We type:
  ```java
  > java UDPClient &
  > java UDPServer &
  > fg %1
  ...
  ```
- Kill the background server when you’re done:
  ```bash
type > kill %2
  ```

Goals for Wellscape server*
- Handles only one HTTP request.
- Accepts and parses the HTTP request.
- Gets the requested file from the server's file system.
- Creates an HTTP response message consisting of the requested file preceded by header lines.
- Sends the response directly to the client.

*Non ministrari, sed ministrare.

Implementing a Web server
```java
import java.io.*;
import java.net.*;
import java.util.*;

class WebServer {
    public static void main(String argv[]) throws Exception {
        // 1. Create a listenSocket and listen.
        // 2. When someone knocks on the door, create a dedicated connection socket.
        // 3. Read and parse the request.
        // 4. Serve up the files.
    }
}
```

Steps 1 & 2. Creating the sockets
**Steps 1 and 2. The code**

```java
class WebServer {
    public static void main(String argv[]) throws Exception {
        ServerSocket listenSocket = new ServerSocket(6789);
        Socket connectionSocket = listenSocket.accept();
        BufferedReader inFromClient =
            new BufferedReader(new InputStreamReader(
                connectionSocket.getInputStream()));
        DataOutputStream outToClient =
            new DataOutputStream(
                connectionSocket.getOutputStream());
    }
}
```

**Step 3. Read and parse request**
- What kind of a message are we likely to get?
- Something like 
  ```
  GET /~rshull/index.html HTTP/1.0
  ```

**Step 3. Read & parse GET /~rshull/index.html HTTP/1.0**
- Read first line of HTTP request
  ```
  String requestMessageLine = inFromClient.readLine();
  ```
- Parse line to extract filename
  ```
  StringTokenizer tokenizedLine =
      new StringTokenizer(requestMessageLine);
  ```
  ```
  if (tokenizedLine.nextToken().equals("GET")) {
    String fileName = tokenizedLine.nextToken();
    if (fileName.startsWith("/") == true )
      fileName = fileName.substring(1);
  }
  ```
- Remove the slash if present

**Step 4. Prepare to send file . . .**
- Create a file object
  ```
  File file = new File(fileName);
  int numOfBytes = (int) file.length();
  ```
- Attach input stream to file name
  ```
  FileInputStream inFile =
    new FileInputStream(fileName);
  ```
- Determine size and read from file stream into byte array
  ```
  byte[] fileInBytes = new byte[numOfBytes];
  inFile.read(fileInBytes)
  ```
HTTP response message

status line (protocol status code status phrase)

HTTP/1.0 200 OK
Connection: close
Date: Sat, 17 Sep 2011 12:00:15 GMT
Server: Wellscape/1.3.0 (Unix)
Last-Modified: Sun, 7 Aug 2011 ...
Content-Length: 6821
Content-Type: text/html

data data data data data ...

Some HTTP response status codes

200 OK
request succeeded, requested object in this message

301 Moved Permanently
requested object moved, new location specified in this message (Location)

400 Bad Request
request message not understood by server

404 Not Found
requested document not found on this server

505 HTTP Version Not Supported

... and serve it to the client

Client expects us to follow HTTP protocol

Tell 'em file type ...

... and size ...
... and ship it off

outToClient.writeBytes("HTTP/1.0 200 Document Follows\r\n");
If (fileName.endsWith(".jpg"))
outToClient.writeBytes("Content-Type: image/jpeg\r\n");
If (fileName.endsWith(".gif"))
outToClient.writeBytes("Content-Type: image/gif\r\n");
outToClient.writeBytes("Content-Length: " +
numOfBytes + \r\n");
outToClient.writeBytes("\r\n");
outToClient.write(fileInBytes, 0, numOfBytes);

Close socket and go home

If (tokenizedLine.nextToken().equals("GET")){
String fileName = tokenizedLine.nextToken();
// ... all the stuff on the previous two slides
connectionSocket.close();
}
If it's not a GET, we don't handle it
else System.out.println("Bad Request Message");
}

Tie up loose ends
Can we do better?

- Serve more than one request before quitting.
- Listen for additional requests while serving old ones.
- Increase number of file types we can handle.
- Bullet proof the code.