Socket Programming in Java

Required reading:
Kurose 2.7

CSE 3214, Winter 2016
Instructor: N. Vlajic
Socket Programming

**Socket** – a local-host, application created, OS controlled interface (a “door”) into which application process can send/receive messages to/from another application process

- also, a door between application process and end-to-end transport protocols (TCP or UDP)

- each TCP/UDP socket is uniquely identified with 2 pieces of information
  1. name or address of the host (IP address)
  2. identifier of the given process in the destination host (port number)
Socket Programming (cont.)

- Controlled by application developer
- Controlled by operating system

Process
- Socket
- TCP with buffers, variables

Host or server

Internet

Process
- Socket
- TCP with buffers, variables

Host or server

Controlled by application developer
Controlled by operating system
Socket Programming – development of client/server application(s) that communicate using sockets

- developer has control of everything on application side but has little control of transport side of socket
- only control on transport-layer side is
  1. choice of transport protocol (TCP or UDP)
  2. control over a few transport-layer parameters e.g. max buffer and max segment size

Socket programming refers to programming at the application level/layer!

TCP vs. UDP in Socket Programming – to decide which transport-layer protocol, i.e. which type of socket, our application should use, we need to understand how TCP and UDP differ in terms of

- reliability
- timing
- overhead
TCP vs. UDP Reliability

- UDP - there is no guarantee that the sent datagrams will be received by the receiving socket
- TCP - it is guaranteed that the sent packets will be received in exactly the same order in which they were sent

TCP vs. UDP Timing

- UDP - does not include a congestion-control mechanism, so a sending process can pump data into network at any rate it pleases (although not all the data may make it to the receiving socket)
- TCP - TCP congestion control mechanism throttles a sending process when the network is congested – TCP guarantees that data will eventually arrive at the receiving process, but there is no limit on how long it may take

TCP vs. UDP Overhead

- UDP - every time a datagram is passed into the socket, the local and receiving socket address need to be passed along with it (processing overhead)
- TCP - a connection must be established before communications between the pair of sockets start (connection setup time overhead)
TCP vs. UDP in Socket Programming (cont.)

TCP is useful when **indefinite amount of data need to be transferred ‘in order’ and reliably**
- otherwise, we end up with jumbled files or invalid information
  - examples: HTTP, ftp, telnet, …

UDP is useful when **data transfer should not be slowed down by extra overhead of reliable TCP connection**
  - examples: real-time applications
  - e.g. consider a **clock server** that sends the current time to its client – if the client misses a packet, it doesn't make sense to resend it because the time will be incorrect when the client receives it on the second try

In socket programming we pick transport-layer protocol that has services that best match the needs of our application.
Client-Server Model – most common form of network communication in the Internet whose purpose is to enable/provide various types of service to users

- **CLIENT**: process that initiates communication, requests service, and receives response
  - although request-response part can be repeated several times, whole process is finite and eventually comes to an end

- **SERVER**: process that passively waits to be contacted and subsequently provides service to clients
  - runs infinitely
  - can be *iterative* or *concurrent*

Client needs to know server’s address and port initially, but not vice versa.
An iterative server can process only one request at a time – it receives a request, processes it, and sends the response to the requestor before handling another request. The servers that use UDP (i.e. connectionless servers) are normally iterative.
Example [ iterative vs. concurrent servers ]

A concurrent server can process **many requests at the same time**.

The servers that use TCP (i.e. connection-oriented servers) are normally concurrent.
Principles of Client-Server Communication with UDP

- **CLIENT**:  
  1. create socket  
  2. repeat:  
     a. send  
     b. receive  
     c. process  
  3. destroy  

- **SERVER**:  
  1. create socket  
  2. repeat:  
     a. receive  
     b. process  
     c. send  

- all clients use the same socket to communicate with server  
- clients and server exchange packets (datagrams)  
- no handshaking  
- sender explicitly attaches IP address and port of destination to each packet  
- server must extract IP and port of sender from received packet to be able to send its response back
Principles of Client-Server Communication with UDP (cont.)

Each server serves many clients but handles one request at a time.

- **Clients**
  - Create a socket
  - Repeat as needed
  - Send the request
  - Receive response when it arrives
  - Destroy the socket

- **Server**
  - Create a socket
  - Bind the socket
  - Repeat infinitely
  - Receive a request when it arrives
  - Process the request
  - Send the result
Principles of Client-Server Communication with TCP

- **CLIENT:**
  1. create socket
  2. connect
  3. repeat:
     - 3.a) write
     - 3.b) read
     - 3.c) process
  4. destroy

- **SERVER:**
  1. create socket
  2. listen
  3. repeat
     - 3.a) create new socket
     - 3.b) repeat:
         - 3.b.1) read
         - 3.b.2) process
     - 3.b.3) write
  3.c) destroy socket
A parent server creates many children; each child server serves only one client.

Client
- Create a socket
- Connect to server

Connection setup

Server
- Create a socket
- Bind the socket
- Listen for a client
- Repeat infinitely
  - Create a child to server client

Parent server

Child server

Repeat as needed
- Write bytes
- Read bytes when they arrive
- Destroy the socket

Repeat as needed
- Read bytes when they arrive
- Process
- Write bytes
- Destroy the temporary socket

Create a temporary socket
Java Socket Programming

Advantages of Socket Programming in Java

• **applications are more neatly and cleanly written in Java than in C or C++**
  - there are fewer lines of code and each line can be explained to novice programmer without much difficulty

• **Java keeps all socket transport-layer complexity “under the cover”**
  - developer can focus on application rather than worrying about how network and transport layer operate

• **Java does not rely on native code** ⇒ programs can communicate over network (the Internet) in platform-independent fashion

Disadvantages of Socket Programming in Java

• **Java does not expose the full range of socket possibilities to developer**
int set_up_socket(u_short port) {
    char myname[MAXHOSTNAME+1];
    int s;
    struct sockaddr_in sa;
    struct hostent *he;
    bzero(&sa,sizeof(struct sockaddr_in)); /* clear the address */
    gethostname(myname,MAXHOSTNAME); /* establish identity */
    he= gethostbyname(myname); /* get our address */
    if (he == NULL) /* if addr not found... */
        return(-1);
    sa.sin_family= he->h_addrtype; /* host address */
    sa.sin_port= htons(port); /* port number */
    if ((s= socket(AF_INET,SOCK_STREAM,0)) <0)        /* finally, create socket */
        return(-1);
    if (bind(s, &sa, sizeof(sa), 0) < 0) {
        close(s);
        return(-1); /* bind address to socket */
    }
    listen(s, 3); /* max queued connections */
    return(s);
}

Example [ Java vs. C socket programming ]
ServerSocket servsock = new ServerSocket(port, backlog, bindAddr);
### java.net package

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InetAddress class</td>
<td>represents IP address – implements Serializable (2 subclasses InetAddress, InetAddress – final classes)</td>
</tr>
<tr>
<td>ServerSocket class</td>
<td>passive TCP (server) socket – used on server side to wait for client connection requests</td>
</tr>
<tr>
<td>Socket class</td>
<td>active TCP socket – can be used as communication end point both on client and server side</td>
</tr>
<tr>
<td>DatagramSocket class</td>
<td>connectionless (UDP) socket – used for sending and receiving datagrams (packets that are individually addressed and routed)</td>
</tr>
<tr>
<td>DatagramPacket class</td>
<td>datagram packet – in addition to data also contains IP address and port information – used in UDP!</td>
</tr>
<tr>
<td>MulticastSocket class</td>
<td>subclass of DatagramSocket – can be used for sending and receiving packets to/from multiple users</td>
</tr>
</tbody>
</table>

https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html
```java
package com.javacodegeeks.examples;
import java.net.Inet4Address;
import java.net.UnknownHostException;

public class SimpleInet4AddressExample {

    public static void main(String[] args) {
        String url = "javacodegeeks.com";

        try {
            Inet4Address address = (Inet4Address) Inet4Address.getByName(url);
            System.out.println("The IP of "+url+" is "+address.getHostAddress());
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

**getByAddress**

```java
public static InetAddress getByAddress(String host,
                                        byte[] addr)
    throws UnknownHostException
```

Creates an InetAddress based on the provided host name and IP address. No name service is checked for the validity of the address.

The host name can either be a machine name, such as "java.sun.com", or a textual representation of its IP address.

No validity checking is done on the host name either.
**Example**  [ Java socket programming – unicast communication ]

Use the following simple client/server application to demonstrate socket programming for both TCP and UDP:

1) A client reads a line from its standard input (keyboard) and sends line out through its socket to the server.

2) The server reads a line from its connection socket.

3) The server converts the line to upper case.

4) The server sends the modified line out through its socket to the client.

5) The client reads the modified line from its socket and prints the line on its standard output (monitor).
**Java Socket Programming with UDP**

machine = `jun07.cs.yorku.ca`

**Client**
- create socket
  - `clientSocket = new DatagramSocket()`
- create, address and send datagram using
  - `clientSocket`
- read reply from
  - `clientSocket`
- close
  - `clientSocket`

**Server (running)**
- create socket for incoming datagrams
  - `serverSocket = new DatagramSocket()`
- read datagram from
  - `serverSocket`
- write reply to
  - `serverSocket`
  - specifying client's host address & port number

machine = `blue.cs.yorku.ca`
import java.io.*;
import java.net.*;

class UDPClent {

public static void main (String argv[]) throws Exception {

    BufferedReader inFromUser = new BufferedReader (new InputStreamReader(System.in));
    byte[] sendData = new byte[1204];
    byte[] receiveData = new byte[1204];
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("blue.cs.yorku.ca");

    sendData = sentence.getBytes();

    String sentence = inFromUser.readLine();
    sendData = sentence.getBytes();

    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 1234);
    clientSocket.send(sendPacket);

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

    System.out.println(receivedData);

}}
DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress, 7777);

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.trim());

clientSocket.close();
server on blue.cs.yorku.ca ...

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

class UDPServer {
    public static void main (String argv[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(7777);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while(true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
            .......... ...
        }
    }
}
```

Why do we have to specify port number in this case?!
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase() + '\n';

sendData = capitalizedSentence.getBytes();

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

serverSocket.send(sendPacket);

} }

} }

} }
Java Socket Programming with TCP

Client

create socket, connect to server
clientSocket = new Socket()

send request using clientSocket

read reply from clientSocket

close clientSocket

TCP connection setup

Server (running)

create socket for incoming requests
welcomeSocket = new ServerSocket()

wait for incoming requests

create new socket
connectionSocket = welcomeSocket.accept()

read request from connectionSocket

write reply to connectionSocket

close connectionSocket

close clientSocket
import java.io.*;
import java.net.**;

class TCPClient {

    public static void main (String argv[]) throws Exception {

        String sentence;
        String modifiedSentence;

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

        Socket clientSocket = new Socket("blue.cs.yorku.ca", 5555);

        DataOutputStream outToServer =
                new DataOutputStream(clientSocket.getOutputStream());

        BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

    }
}
sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + '\n');
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();
server on blue.cs.yorku.ca ...

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

class TCPServer {

    public static void main (String argv[]) throws Exception {

        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(5555);

        while(true) {

            Socket connectionSocket = welcomeSocket.accept();

            BufferedReader inFromClient = new BufferedReader (new InputStreamReader(connectionSocket.getInputStream()));

            create welcoming socket at port 5555
            wait for contact-request by clients
            once a request arrives, allocate new socket
            create & attach input stream to new socket

            String clientSentence;
            String capitalizedSentence;

            ServerSocket welcomeSocket = new ServerSocket(5555);

            while(true) {

                Socket connectionSocket = welcomeSocket.accept();

                BufferedReader inFromClient = new BufferedReader (new InputStreamReader(connectionSocket.getInputStream()));

                outToClient

                inFromClient

                TCP Layer

                TCP socket

                dedicated socket

                server on blue.cs.yorku.ca …
create & attach output stream to new socket

DataOutputStream outToClient =
new DataOutputStream(connectionSocket.getOutputStream());

read from socket

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

write to socket

outToClient.writeBytes(capitalizedSentence);

end of while loop - wait for another client to connect

NOTE: This version of TCP Server is NOT actually serve clients concurrently,
but it can be easily modified (with threads) to do so.