Tutorial

Socket Programming in Java

Fall 2016

Based on powerpoint slides provided by Kurose & Ross 6th edition

McGill

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- two types of transport service via socket API:
  - unreliable datagram
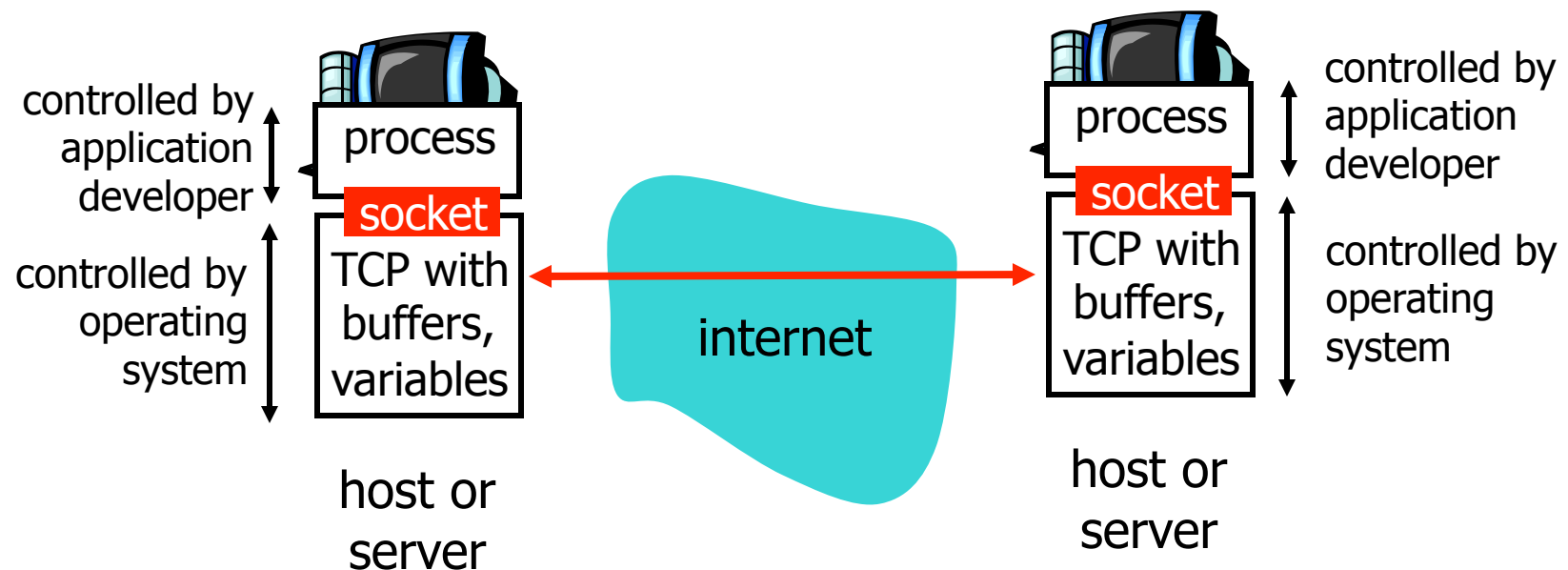 - reliable, byte stream-oriented

socket

a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of bytes from one process to another

controlled by application developer

process

socket

controlled by operating system

TCP with buffers, variables

internet

host or server

controlled by application developer

process

socket

TCP with buffers, variables

controlled by operating system

host or server

Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
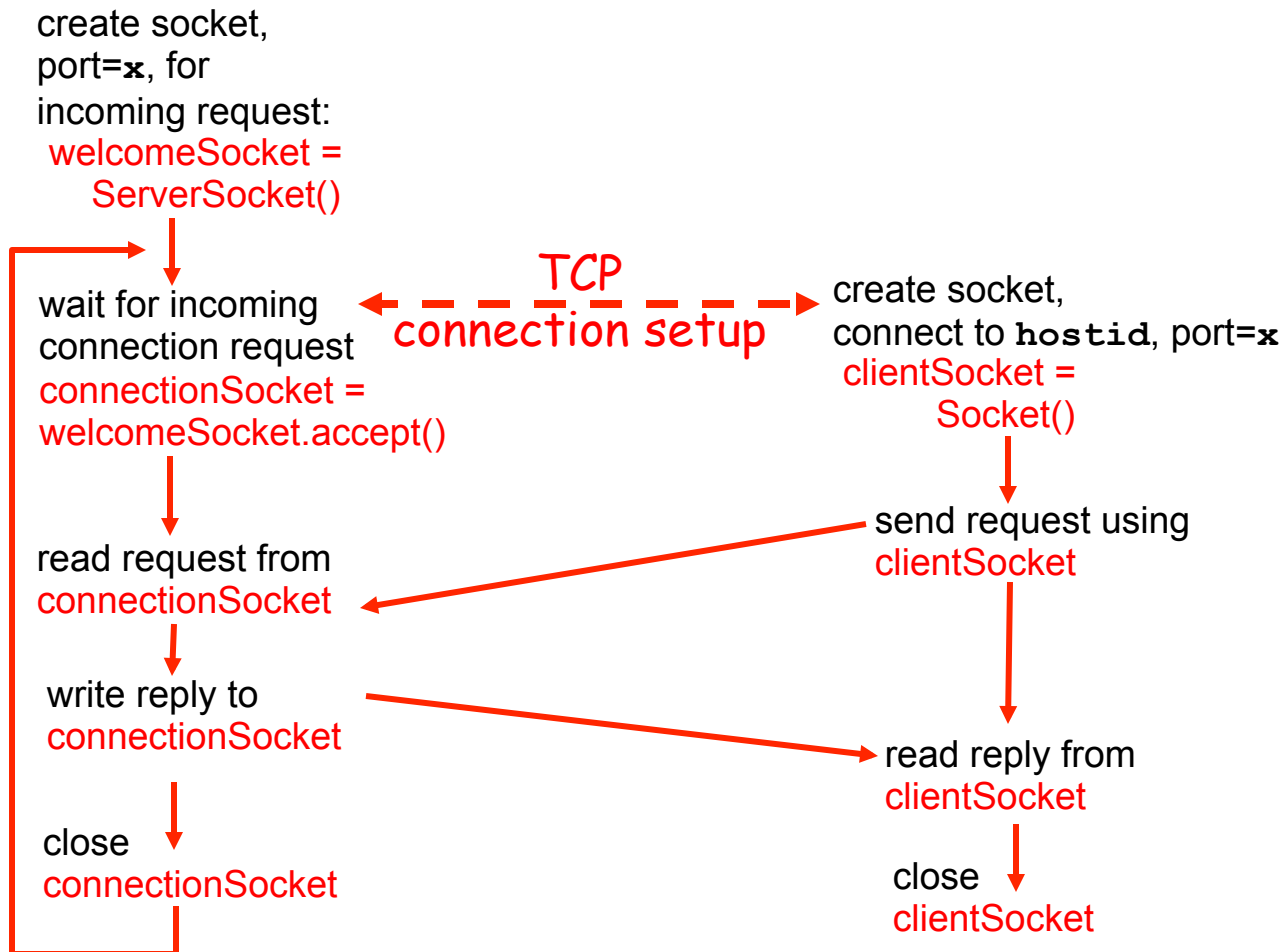 - source port numbers used to distinguish clients (more in Chap 3 of Kurose & Ross)

application viewpoint

TCP provides reliable, in-order transfer of bytes between client and server

Client/server socket interaction: TCP
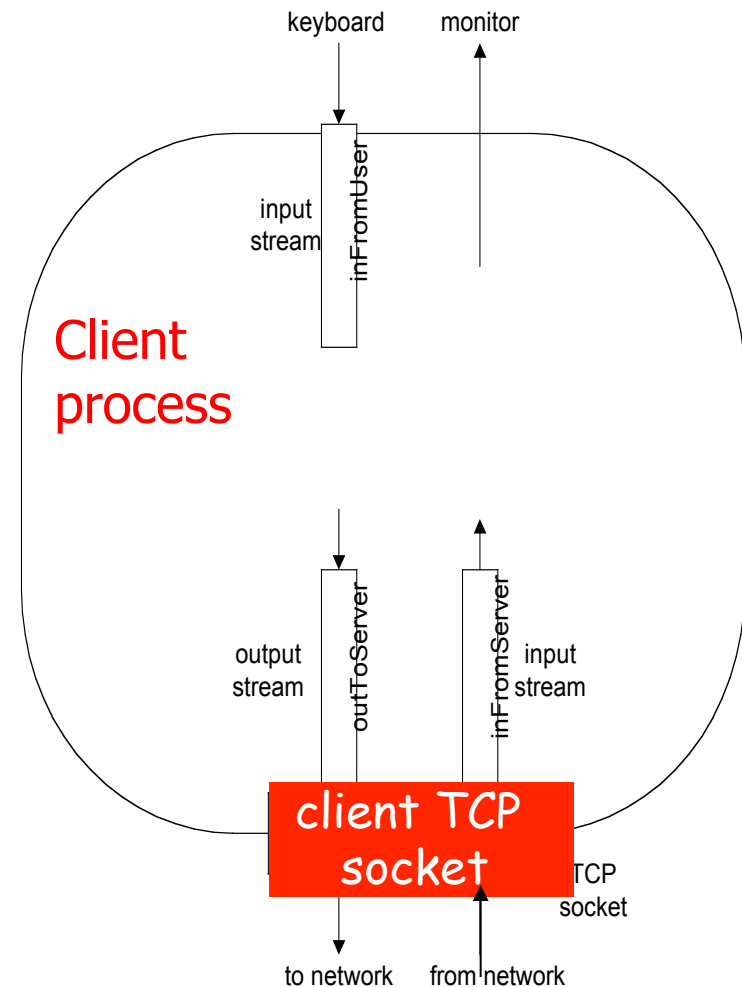
Server (running on **hostid**)   Client

create socket,
port=**x**, for
incoming request:
 welcomeSocket =
    ServerSocket()

wait for incoming   *TCP*   create socket,
connection request   *connection setup* connect to **hostid**, port=**x**
connectionSocket =   clientSocket =
welcomeSocket.accept()   Socket()

                                           send request using
read request from   clientSocket
connectionSocket

write reply to
connectionSocket   read reply from
                                             clientSocket

close   close
connectionSocket   clientSocket

Stream jargon

- A stream is a sequence of data that flows into or out of a process.

- An input stream is attached to some input source for the process, e.g., keyboard or socket.

- An output stream is attached to an output source, e.g., monitor or socket.

  (See Java documentation for more)

keyboard   monitor

input
stream   inFromUser

Client
process

output   outToServer   inFromServer   input
stream   stream

client TCP
socket   TCP
                                 socket

to network   from network

Socket programming with TCP

Example client-server app:

1) client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (**inFromServer** stream)

Example: Java client (TCP)

```java
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("hostname", 6789);

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

Create input stream →

Create client socket, connect to server →

Create output stream attached to socket →

Example: Java client (TCP), cont.

Create input stream attached to socket →
```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
```

Send line to server →
```
outToServer.writeBytes(sentence + '\n');
```

Read line from server →
```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();
      }
    }
```

Example: Java server (TCP)

```
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()));
```

Create welcoming socket at port 6789 → `ServerSocket welcomeSocket = new ServerSocket(6789);`

Wait, on welcoming socket for contact by client → `Socket connectionSocket = welcomeSocket.accept();`

Create input stream, attached to socket → `BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));`

Example: Java server (TCP), cont

Create output stream, attached to socket →
DataOutputStream outToClient =
   new DataOutputStream(connectionSocket.getOutputStream());

Read in line from socket →
clientSentence = inFromClient.readLine();

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

Write out line to socket →
outToClient.writeBytes(capitalizedSentence);
      }
    }
  }

End of while loop, loop back and wait for another client connection

Multi-Threaded Servers

- TCPServer processes one request at a time
  - … within while loop

- This blocks other incoming connections

- Instead, have separate parallel threads handle each request
  - Use Java's Runnable syntax
  - See MultiThreadedTCPServer.java for an example

Socket programming *with UDP*

UDP: no "connection" between client and server

- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides <u>unreliable</u> transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP

Server (running on **hostid**)

create socket,
port= x.
serverSocket =
DatagramSocket()

read datagram from
serverSocket

write reply to
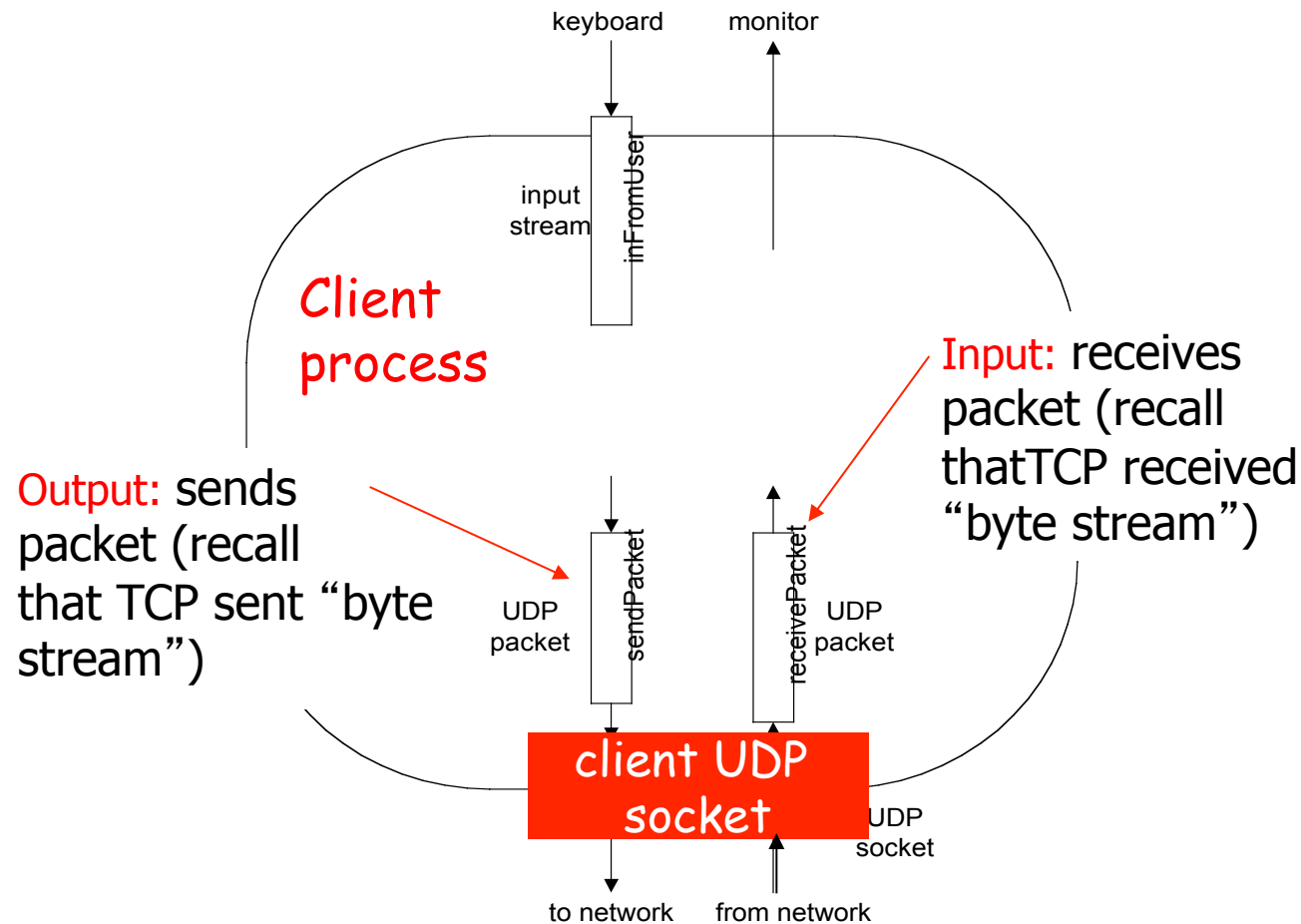serverSocket
specifying
client address,
port number

Client

create socket,
clientSocket =
DatagramSocket()

Create datagram with server IP and
port=x; send datagram via
 clientSocket

read datagram from
clientSocket

close
 clientSocket

Example: Java client (UDP)

Example: Java client (UDP)

```
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();
```

Create input stream →

Create client socket →

Translate hostname to IP address using DNS →

Example: Java client (UDP), cont.

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

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

Send datagram to server

```
clientSocket.send(sendPacket);

DatagramPacket receivePacket =
    new DatagramPacket(receiveData, receiveData.length);
```

Read datagram from server

```
clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
    }
}
```

Example: Java server (UDP)

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

class UDPServer {
  public static void main(String args[]) throws Exception
   {

      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);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

Example: Java server (UDP), cont

String sentence = new String(receivePacket.getData());

Get IP addr port #, of sender →

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

Create datagram to send to client →

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

Write out datagram to socket →

serverSocket.send(sendPacket);
        }
    }
}

End of while loop, loop back and wait for another datagram

Catching Exceptions

- Recall the Java syntax for catching/handling exceptions

```
try {
    // Code that might throw an exception
} catch (Exception e) {
    // This is called if an exception is thrown
}
```

- Exceptions thrown, for example, when
  - Opening a socket on a port that's already taken
  - Transmission timeouts
  - …
- Best to catch Exceptions as soon as possible

Additional Words of Advice

- For TCP sockets
  - Always remember to close the socket when you're finished with it
  - Always add '\n' to transmitted data to signal end of transmission
- Remember to launch server before connecting with client
- Don't try to open client and server on same port on same machine

Other Useful Resources

- Java API documentation
  - http://docs.oracle.com/javase/7/docs/api/

- Tutorial on Essential Java Classes (covers I/O)
  - https://docs.oracle.com/javase/tutorial/essential/index.html

- Java Networking Tutorial
  - https://docs.oracle.com/javase/tutorial/networking/index.html