

# SC250

## Computer Networking I

# Socket Programming

Prof. Matthias Grossglauser

School of Computer and Communication Sciences  
EPFL

<http://lcawww.epfl.ch>



1

## Transport vs. Network Layer

- Network layer:
  - logical communication between hosts (IP to IP)
- Transport layer:
  - logical communication between processes (IP+port to IP+port)
  - relies on + enhances network layer services

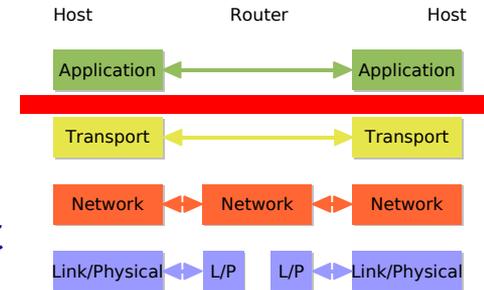
**Household analogy:**  
*12 kids sending letters to 12 kids*

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill handling all mail in each house
- network-layer protocol = postal service

3

## Today's Objectives

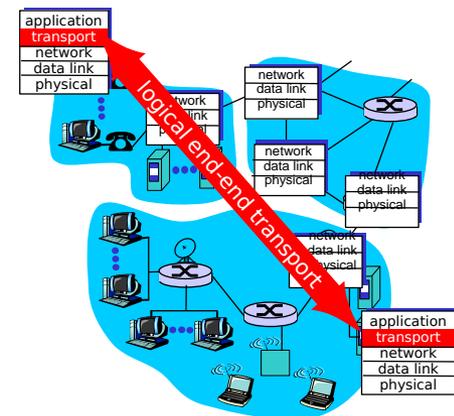
- Transport layer service through sockets
  - Multiplexing-demultiplexing:
    - How to make sure that multiple processes on a single machine can communicate concurrently without interfering with each other?
- Java sockets
  - stream sockets (TCP)
  - stream example
  - datagram sockets (UDP)
  - datagram example
- Multi-threaded servers
- Socket programming in C



2

## Internet Transport-Layer Protocols

- Reliable, in-order delivery: TCP
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees



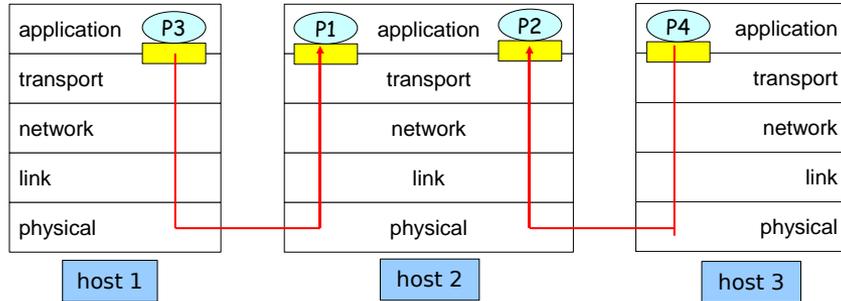
4

# Multiplexing/Demultiplexing

## Demultiplexing at rcv host:

delivering received segments to correct socket

■ = socket    ○ = process

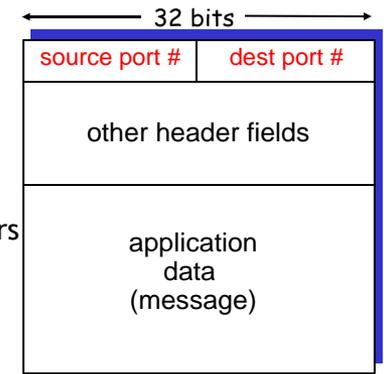


## Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

# How Demultiplexing Works

- Host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- Host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

# Connectionless Demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(99111);
DatagramSocket mySocket2 = new DatagramSocket(99222);
```

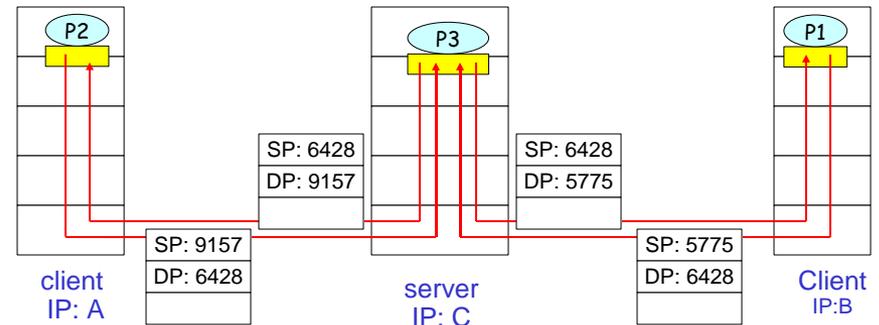
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

# Connectionless Demultiplexing (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



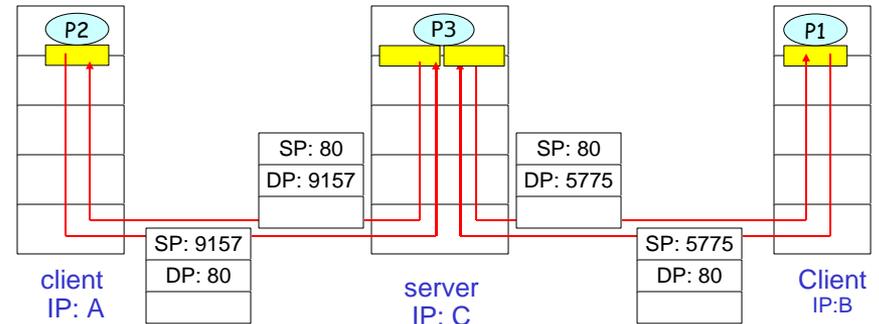
SP provides "return address"

## Connection-Oriented Demultiplexing

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

9

## Connection-Oriented Demux (cont)



10

## Network API: Sockets

- Application Programming Interface (API) provides a set of function types, data structures and constants
  - flexible, simple to use, standardized
- API to access transport protocols of the operating system is called a **socket**
  - Gives a file-system-like abstraction to the services of the transport protocols (open, close, read, write)
  - BSD sockets (introduced in BSD 4.1 Unix in 1981) are the most popular Internet sockets: FreeBSD, Linux, Windows, Mac OS X, ...
  - Sockets also used for other purposes (e.g., Unix interprocess communication)

11

## Client/Server Paradigm

- Network applications typically have two components:
  - Client:
    - initiates contact with the server
    - typically requests service from the server
  - Server:
    - passively listens for clients to connect (on a given port)
    - server process usually running all the time
- Applications implementing protocol standard
  - use well-known ports and adhere to standard (RFC)
- Proprietary protocol
  - complete control over design but should not use well-known port numbers

12

# Socket Programming with TCP

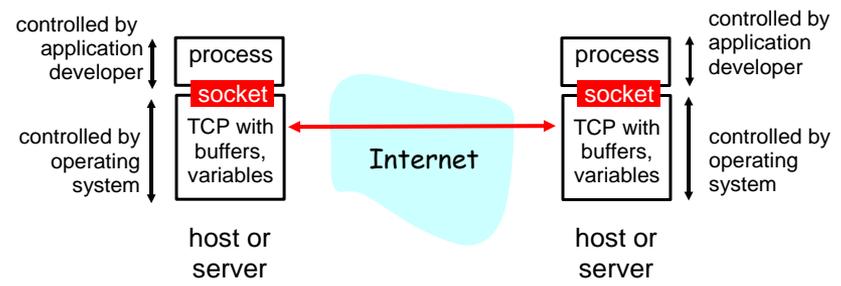
- Client must contact server
- server process must first be running
- server must have created socket (door) that welcomes client's contact
- Client contacts server by:
  - creating client-local TCP socket
  - specifying IP address, port number of server process
- When client creates socket: client TCP 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

**application viewpoint**  
*TCP provides reliable, in-order transfer of bytes ("pipe") between client and server*

# TCP Socket Class

- Used for both client and server
- New socket is created using a Socket() constructor
- 4 constructors + 2 protected
- Connect with creation

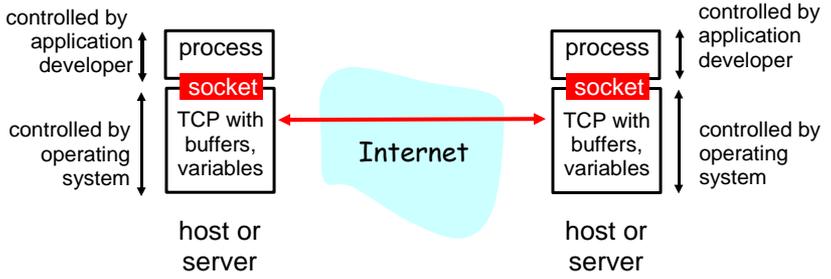
```
java.lang.Object
|
+--java.net.Socket
public class Socket
extends Object
```



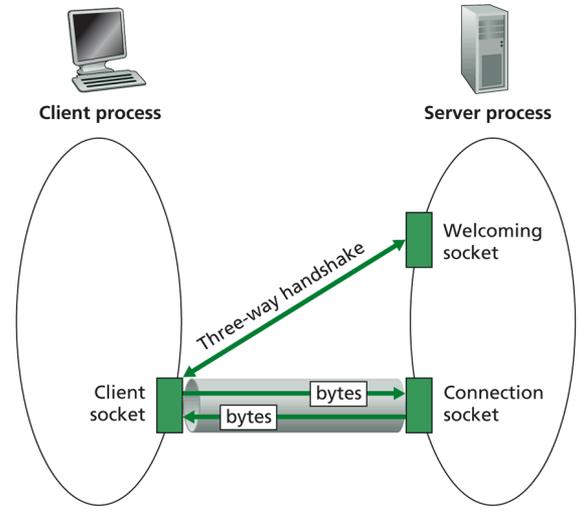
# TCP ServerSocket Class

- Used for server
- New socket is created using a ServerSocket() constructor
- 3 constructors
- Buffers incoming connection requests (SYNs)
- Use accept() to get the next connection

```
java.lang.Object
|
+--java.net.ServerSocket
public class ServerSocket
extends Object
```



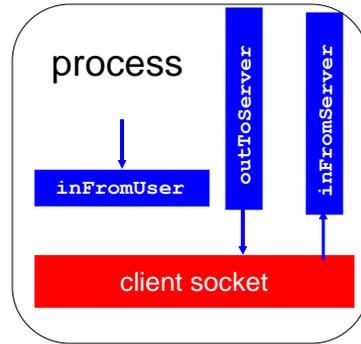
# TCP: Welcoming and Connection Sockets



# Socket Programming with TCP

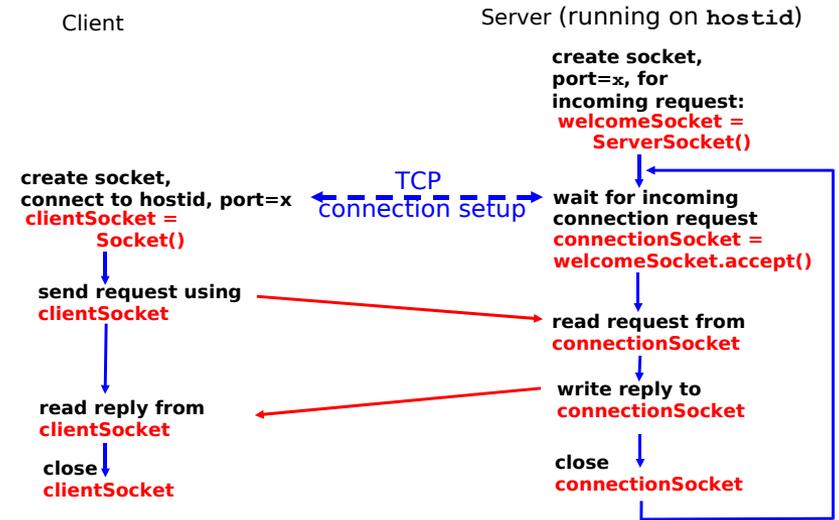
- **Example client-server app:**
- client reads line from standard input (`inFromUser` stream), sends to server via socket (`outToServer` stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (`inFromServer` stream)

- **Input stream:** sequence of bytes into process
- **Output stream:** sequence of bytes out of process



17

# Client/Server Socket Interaction: TCP



18

## Example: Java Client (TCP)

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

class TCPClient {
    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        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());
```

19

## Example: Java Client (TCP)

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

        sentence = inFromUser.readLine();          aaaa

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

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

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

        clientSocket.close();
    }
}
```

20

## Example: Java Server (TCP)

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



```
class TCPServer {
    public static void main(String argv[]) throws Exception
    {
        String clientSentence;
        String capitalizedSentence;

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

        Wait on welcoming socket for contact by client
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();

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

21

## Example: Java Server (TCP)

Create output stream, attached to socket

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

Read in line from socket

```
clientSentence = inFromClient.readLine();      aaaa
```

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

Write out line to socket

```
outToClient.writeBytes(capitalizedSentence);    AAAA
```

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

22

## Socket Programming with UDP

UDP: no "connection" between client and server

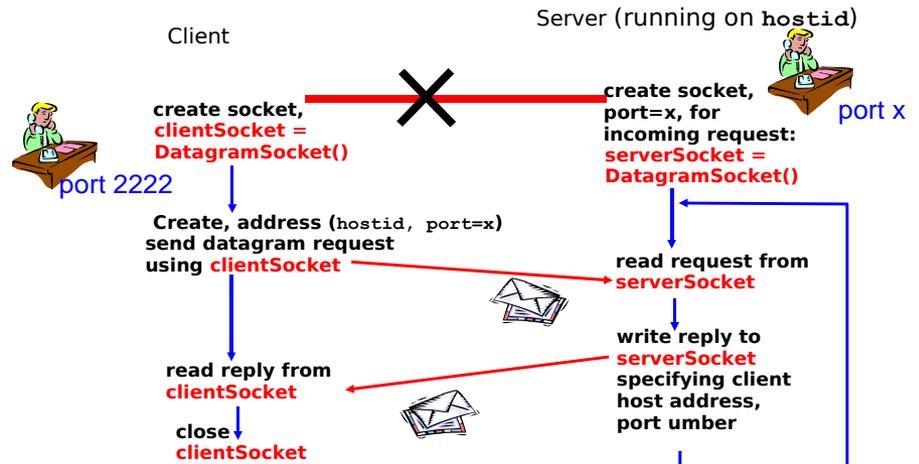
- no handshaking
- sender explicitly attaches IP address and port of destination
- receiver must extract IP address, port of sender from received datagram

application viewpoint  
 UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

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

23

## Client/Server Socket Interaction: UDP



24

## DatagramPacket Class

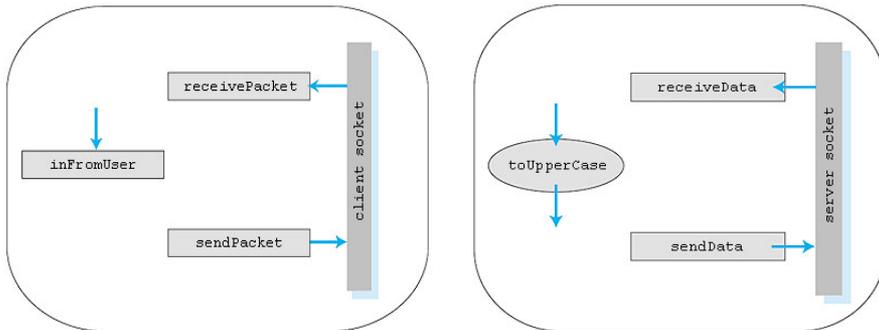
- Used for both client and server
- An independent message (datagram packet) is created using a DatagramPacket() constructor
- 4 constructors

```
java.lang.Object
|
+--java.net.DatagramPacket
```

```
public final class DatagramPacket
extends Object
```

25

## Example: Java Client and Server (UDP)



27

## DatagramSocket Class

- Used for both client and server
- New socket is created using a DatagramSocket() constructor
- 3 constructors

```
java.lang.Object
|
+--java.net.DatagramSocket
```

```
public class DatagramSocket
extends Object
```

26

## Example: Java Client (UDP)

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

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

Create input stream

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

Create client socket

```
DatagramSocket clientSocket = new DatagramSocket();
```

Translate hostname to IP address using DNS

```
InetAddress IPAddress = InetAddress.getByName("host");
```

Create input and output buffer

```
byte[] sendData = new byte[1024];
byte[] receiveData = new byte[1024];
```

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

28

## Example: Java Client (UDP)

```
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);

Read 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();
}

}
```

29

## Example: Java Server (UDP)

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

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



```
Create datagram socket at port 9876
DatagramSocket serverSocket =
    new DatagramSocket(9876);

byte[] receiveData = new byte[1024];
byte[] sendData = new byte[1024];

while(true)
{
    Create space for received datagram
    DatagramPacket receivePacket =
        new DatagramPacket(receiveData, receiveData.length);
    Receive datagram
    serverSocket.receive(receivePacket);
```

30

## Example: Java Server (UDP)

```
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
```

31

## DNS Lookups

- `InetAddress` contains an IP address as `String` and as `int`
- Can also be used for explicit DNS lookups:
  - `InetAddress addr = InetAddress.getByName("www.epfl.ch"); System.out.println(addr.getHostAddress());`
  - `InetAddress addr = InetAddress.getByName("128.178.50.137"); System.out.println(addr.getHostName());`
  - `InetAddress.getAllByName()` to get all IP addresses of a host name
- `InetAddress` is rarely needed: Socket constructors accept IP addresses, names, and objects of type `InetAddress` (implicit DNS)

32

## Reading Directly from a URL

- Java provides a number of functions that make programming much easier:

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

public class URLReader {
    public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                yahoo.openStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

Open URL  
as stream

Read and  
display the  
web page

No need to implement HTTP

33

## Writing to a URLConnection

- Possible to read and write to a URL (using HTTP)

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

public class URLWrite {
    public static void main(String[] args) throws Exception {
        URL url = new URL(
            "http://www.merriam-webster.com/cgi-bin/dictionary");
        URLConnection connection = url.openConnection();
        connection.setDoOutput(true);

        PrintWriter out = new PrintWriter(
            connection.getOutputStream());
        out.println("book=Dictionary&va=java&x=0&y=0");
        out.close();
    }
}
```

allow writing  
to the URL

This results in  
a POST request,  
rather than GET

34

## Writing to a URLConnection (2)

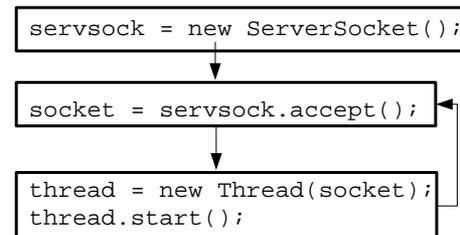
```
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

get dictionary  
entry on "java"

## Multi-Threaded TCP Server

To be able to handle a larger number of clients the server should not process clients in sequence but in parallel.

- Server continuously listens on server socket for client requests
- When `accept()` returns a socket, start a new thread to handle the client; hand over the socket to the thread
- Separate threads are usually only used for TCP, not for UDP



### Java Threads

- "light-weight" process
- shares memory, etc. with parent (possible conflicts!)
- Extend class `Thread` and overwrite `run()` (the "main" function of a thread)

35

36

## Example: Multi-Threaded TCP Server

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

public class TCPMultiServer {
    public static void main(String[] args) throws Exception {
        serverSocket = new ServerSocket(4444);
        while (true) {
            TCPMultiServerThread thread = new
                TCPMultiServerThread(serverSocket.accept());
            thread.start();
        }
        serverSocket.close();
    }
}
```

Create thread for new socket

37

## Example: Server Thread

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

public class TCPMultiServerThread extends Thread {
    private Socket socket = null;

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

    public void run() {
        String clientSentence;
        String capitalizedSentence;

        BufferedReader inFromClient =
            new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
    }
}
```

Socket handed over from main server

Create input stream, attached to socket

38

## Example: Server Thread (2)

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

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

outToClient.writeBytes(capitalizedSentence);    AAAA

close(socket);
}
```

Create output stream, attached to socket

Read in line from socket

Write out line to socket

39

## Exception Handling

- Distributed application: many things can go wrong!
  - Client request to server: host not available, overloaded, behind firewall, no server listening on destination port,...
  - Creating a server socket: socket already in use
  - DNS errors
  - Worst of all: transient thread-errors due to timing and synchronization problems
- Easy: writing network code that works if everything ok
- Hard: writing robust network code!
  - One aspect ignored in examples: exception handling
    - Required for almost all network-related operations
    - try {...} catch (IOException ex) {...}
  - Murphy's Law applies...

40

## Socket Programming in C/C++

- C/C++ still significantly faster than Java (although this only matters if the network *isn't* the bottleneck)
- Operating system (and therefore the OS side of sockets) traditionally programmed in C/C++
- C/C++ has a much lower level of abstraction
  - C/C++ provides more functionality
  - Java does many things automatically that you have to do by hand in C/C++
  - Network programming is easy to get wrong: C/C++ makes it even a bit harder

41

## TCP Socket Programming in C/C++

### Client side

- `socket()` returns client socket ID
- `connect()` with server IP address and port, sends connection request
- `send()` sends data via client socket
- `recv()` receives from socket
- `close()` closes connection

Note: no explicit `connect`, `listen`, and `bind` in Java

### Server side

- `socket()` returns server socket ID
- `bind()` binds socket to server IP address and port
- `listen()` waits for connection request on server socket
- `accept()` accepts connection request and returns id of a new socket for communication with the client
- `send()`, `recv()`, `close()` same as for the client socket

42

## UDP Socket Programming in C/C++

### Client side

- `socket()` returns client socket ID
- `sendto()` sends data via client socket; need to specify IP addr. and port
- `recvfrom()` receives from socket
- `bind()` is optional
- `close()` closes socket

Note: OS supplies local IP address and port if `bind()` is not used

### Server side

- `socket()` returns server socket ID
- `bind()` binds socket to server IP address and port
- `sendto()` sends data via client socket; need to specify IP addr. and port
- `recvfrom()` receives from socket
- `close()` closes socket

43

## Raw Sockets

- Raw sockets allow to create raw IP packets (bypassing the transport layer)
  - use type `SOCK_RAW` when calling `socket()`
  - no port numbers!
  - access similar to datagram sockets
- Necessary e.g. to implement ping (ICMP)
- Only the superuser (root) may create raw sockets

Note: Raw sockets are not supported in Java

44

## Example: TCP Server in C

```
#include "inet.h"

int main(int argc, char *argv[]) {
    int sd, newSd, rc, i, n, cliLen;

    struct sockaddr_in cliAddr, servAddr;           // addresses
    char msg[MAX_MSG];

    sd = socket(AF_INET, SOCK_STREAM, 0);         // create socket

    // bind socket
    servAddr.sin_family = AF_INET;
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servAddr.sin_port = htons(SERVER_PORT);
    rc = bind(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));

    listen(sd, 5);
}
```

45

## Summary : Socket Programming

### ▪ TCP

**Server** process must first be running and created server socket

**Client** creates client-local TCP socket specifying IP address, port number of server socket

**Client TCP** connects to server TCP

**Server** creates new TCP socket for server process to communicate with client (possibly in new Thread)

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

**Client and Server** processes use streams for input and output data

### ▪ UDP

**Server** process must first be running and have created a socket

**Client** creates client-local socket and group data in packets specifying each IP address, port number of server process at server socket

**UDP** provides unreliable transfer of datagrams between client and server

**Client and Server** processes use datagrams for input and output data

47

## Example: TCP Server in C

```
// server infinite loop

while(1) {
    cliLen = sizeof(cliAddr);
    newSd = accept(sd, (struct sockaddr *) &cliAddr, &cliLen);

    n = recv(newSd, msg, MAX_MSG, 0);

    msg = str_to_upper(msg);           // not implemented here

    write(newSd, msg, n);

    close(newSd);
} // end of infinite loop
}
```

46

## Summary

### ▪ Java socket programming

- higher level of abstraction than C/C++
- introduction to most important functions
- more sophisticated functions: access to socket options, multicast communication, etc.

### ▪ Different communication models

- TCP streams - byte data pipes
- server socket for accepting incoming connections
- UDP datagrams - isolated messages
- Raw IP sockets (not in Java)

### ▪ References:

- Pointers under “resources” on class web site
- Elliotte R. Harold, Java Network Programming (3<sup>rd</sup> ed), O'Reilly, 2004.

48