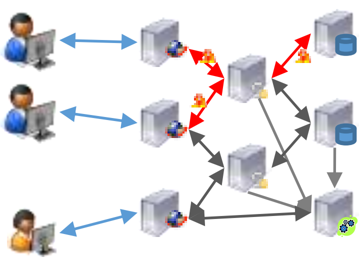


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

2

Overview
Interprocess Communication (IPC)

Overview IPC

- Within an Operating System (**OS**) threads and processes need to communicate
  - Threads can use shared variables to communicate
- For processes it is harder: they do not share memory
  - Local processes: different address space / protection
  - Remote processes: physical barrier
- Consequently, a set of techniques for the exchange of data among processes exist – they are called **Inter-process communication** (**IPC**) methods
- We discuss major ones, more exotic in appendix

IPC Types [local]: Files, Semaphores, Signals

- **Files**: all operating systems
- **Semaphores**: all POSIX systems; Windows
  - Synchronizes access to shared data and resources
  - A protected variable or abstract data type
  - Only one thread / process can change it at a time
- **Signals**: most OSs; Windows: only C run-time lib
  - Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred
  - When a signal is sent to a process, the operating system interrupts the process's normal flow of execution

IPC Types [local]: Shared memory, Memory-mapped Files

- **Shared Memory**: all POSIX systems
  - Memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies
  - Used e.g. to transfer images between the application and the X server on Unix systems
- **Memory-mapped files**: POSIX; Windows (Link)
  - **mmap** is a POSIX-compliant Unix system call that maps files or devices into memory
  - Initially file contents are not entirely read from disk and don't use physical RAM at all
    - The actual reads from disk are performed in "lazy" manner, after a specific location is accessed (demand paging)

IPC Types [local/remote]: Pipes

- **Pipes** (anonymous/named): all POSIX systems
  - A pipe is a buffered data stream between two processes
  - FIFO; treated as files on API-level
  - a pipe(line): a set of processes chained by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) of the next one
- Example in a POSIX-shell
  - ps | grep java
  - Output of “ps” is piped to a line filter, only lines containing “java” are printed on stdout
- Programmatically: **pipe** (int fd[])
  - fd[0] is a reference (handle) to “read-from“ end (pipe output)
  - fd[1] is a reference (handle) to “write-to“ end (pipe input)

IPC Types [local/remote]: Sockets, MP

- **Sockets**: most OSs
  - An API which allows communications between hosts or between processes on one computer
  - Using the concept of an Internet socket
  - More on this today
- **Message passing**:
  - Via Middleware: mainly **MPI**, Message Passing Interface
  - Communication via sending of messages to recipients
  - (Much) more on this later

IPC Types [local/remote]: RPC

- **RPC – Remote Procedure Calls**
  - It provides an API for calling a “normal” function
  - However, this function can be in the same process, or another process or even host (= computer)
  - Call parameters and return values are the actual data to be exchanged
  - Greatly simplifies programming of distributed applications

- Much more on this will follow:
  - General RPC concepts
  - Java RMI
  - Google RPC (gRPC) and protocol buffers
  - Web services, esp. REST

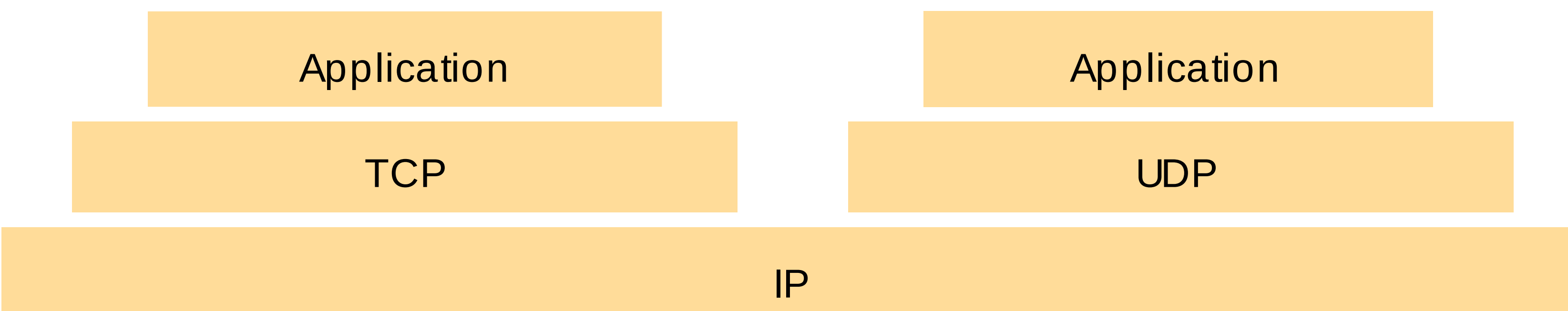
Sockets: Introduction

IP – Internet protocol

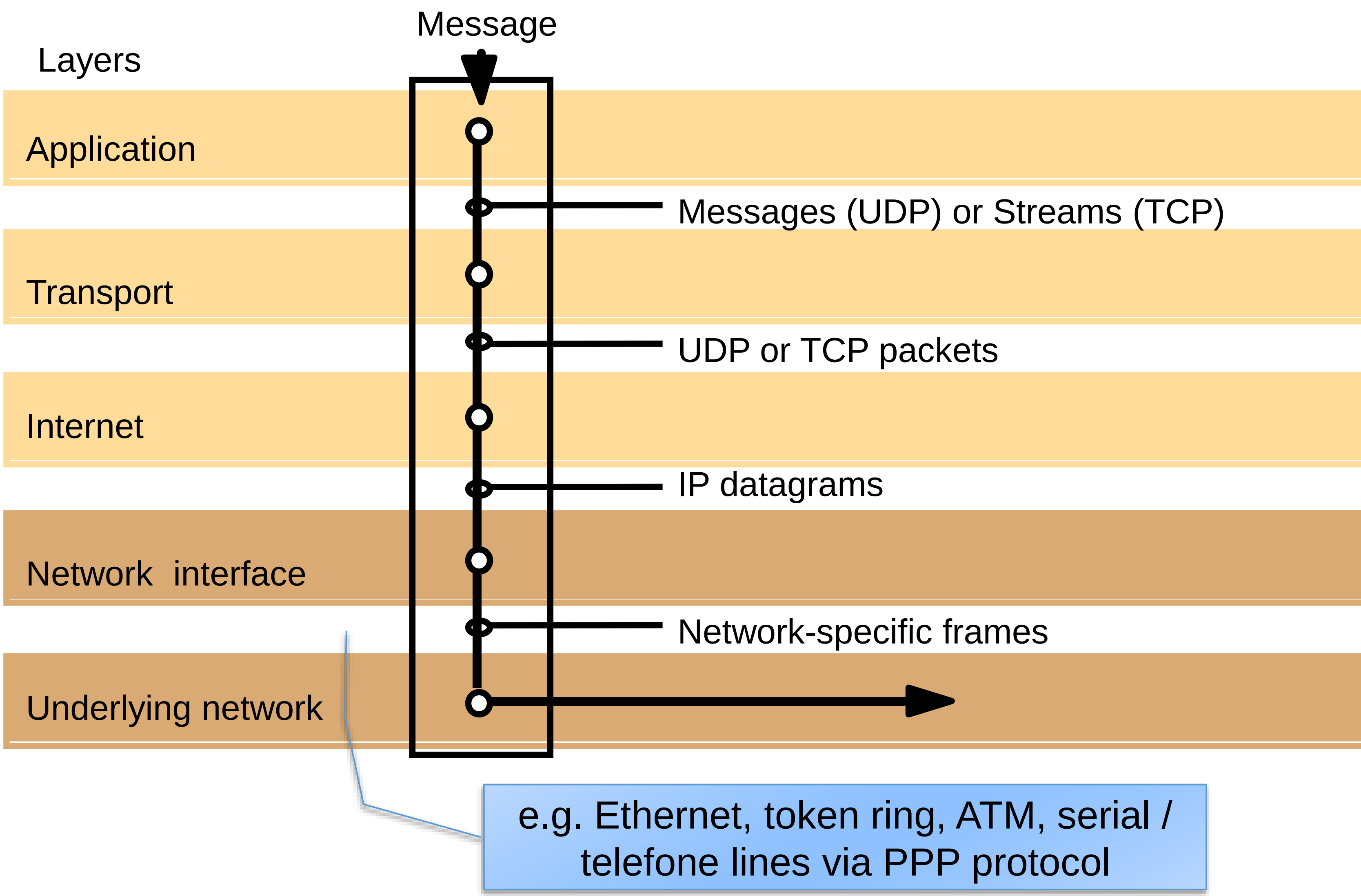
- Born 1969 as a research network of 4 machines
- Funded by Department of Defense's ARPA
  - Advanced Research Projects Agency, renamed to Defense Advanced Research Projects Agency (DARPA) in 1972
- In early 1970s used in ARPANET, first large-scale computer network
- Together with the TCP layer builds TCP/IP protocol stack
- Goal
  - *Build an efficient fault-tolerant network that could connect heterogeneous machines and link separately connected networks*

Layers of the Internet protocol stack

- IP (Internet Protocol) is the underlying network protocol of the Internet virtual network
  - Logical network on top of multiple physical networks
  - Provides basic transmission mechanisms
- Based on it are two transport protocols
  - **TCP**: Transport Control Protocol
    - Reliable connection-oriented protocol
  - **UDP**: User Datagram Protocol
    - Datagram protocol - does not guarantee reliable transmission



TCP/IP layers



Socket

- **Socket**: an application-created, OS-controlled interface/API (a "door") into which application process can both send and receive messages to/from another application process
- Sockets are easy to use: they mimic file operations
  - connect
  - read/write (stream) or send/receive (datagram)
  - disconnect
- **Socket API**
  - introduced in BSD4.1 UNIX, 1981
  - explicitly created, used, released by apps
  - client/server paradigm

Sockets and Ports

- **Port**
  - a **message destination** specified by a small integer (16 bits)
  - Any process can send a message to it
  - Internet protocols use the combination (IP address, local port)
  - IANA (Internet Assigned Numbers Authority) ports:
    - well-known ports: 1 – 1023
    - registered ports: 1024 – 49151
    - dynamic or private ports: 49152 - 65535
- Ports and sockets
  - A socket must be bound to a local port
  - A socket pair uniquely identifies a communication session
    - Pair: [local IP address, local port] + [remote IP address, remote port]

Types of Communication

- Sockets support both types of communication
  - connectionless
    - Via **datagrams** - implemented on network layer via **UDP**
  - connection-oriented
    - **Stream** communication - implemented on network layer via **TCP**
- Type must be specified when creating socket
  - Uses different calls for data exchange
- Berkeley Sockets:
  - Connectionless: **sendto**, **sendmsg**; **recvfrom**, **recvmsg**
  - Connection-oriented: **read** / **write**; **recv** / **send** (extra flags)

Sockets Programming
(Mainly with Python)

Socket programming basics

- Server must be <u>running</u> before client can send anything to it
- Server must have a socket through which it receives and sends segments
- Similarly client needs a socket
- Socket is locally identified with a <u>port number</u>
- Client <u>needs to know</u> server IP address and socket port number

Socket Programming with ***UDP***

- UDP: no "connection" between client and server
- no handshaking
- sender explicitly attaches IP address and port of destination to each segment
- OS attaches IP address and port of sending socket to each segment
- Server can extract IP address, port of sender from received segment

- UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server
- Programming with UDP is "straightforward"
  - See example of Java client/server in additional slides

Datagram (UDP) Communication in C

Sending a message (client)

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message (server)

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

Supplies sender's address with each message it delivers

socket()

- AF_INET – communication domain is Internet domain
- SOCK_DGRAM – specifies datagram communication
- 0 – lets the system select the protocol (here UDP)

bind()

- *ServerAddress* and *ClientAddress* are socket addresses
  - sender: uses any available local port
  - receiver: uses a "well-known" port of itself

Socket Programming with *TCP*

Client must contact server

- server process must first be running
- server must have created socket 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

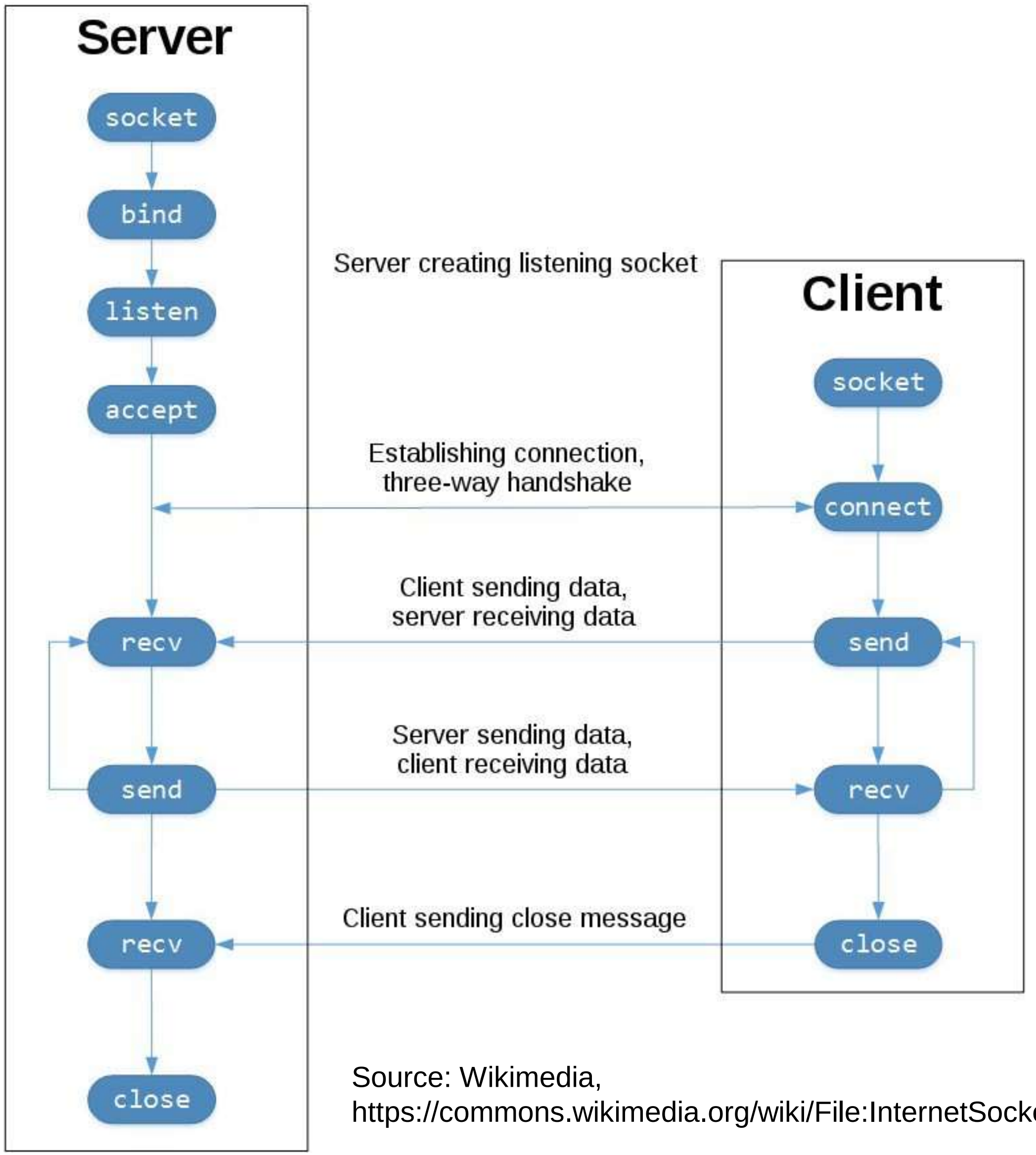
application viewpoint

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

TCP-Sockets in Python

- Based on the tutorial Socket Programming in Python by Nathan Jennings (https://realpython.com/python-sockets/)
- We use Python's socket module as an interface to the Berkeley sockets API
  - Remote hosts are accessible
  - Compare to: Unix domain sockets - allow only same-host communication
- Higher-level APIs available in Python:
  - socketserver module, a framework for network servers
  - Support for various protocols (link):
    - HTTP, FTP, POP, IMAP, SMTP, telnet, XMLRPC, …

Overview of the Interaction



Source: Wikimedia,
https://commons.wikimedia.org/wiki/File:InternetSocketBasicDiagram_zhtw.png

TCP Server in Python

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

TCP Server in Python

My “secret” function

Creates a socket object that supports the **context manager type**, so there is no need to call s.**close**()

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

TCP Server in Python

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

Specify the address family and socket type:

- AF_INET is the Internet address family for IPv4
- SOCK_STREAM is the socket type for TCP

TCP Server in Python

```python
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

Associate the socket with a specific network interface (here HOST = **127.0.0.1**) and port number (here PORT = 65432)

- HOST can be a hostname, IP address, or empty string
- For empty string, the server will accept connections on all available IPv4 interfaces

TCP Server in Python

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

Enables a server to accept() connections: it makes it a “listening” socket

- **listen**() has a *backlog* parameter
- It specifies the number of unaccepted connections that the system will allow before refusing new connections

TCP Server in Python

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

Important: we now have a **new** socket object conn used to communicate with the client. It's distinct from the listening socket that the server is using to accept new connections!

- Blocks and waits for an incoming connection
- When a client connects, it returns a **new socket object** representing the connection (conn) and a tuple holding the address of the client (addr)
- addr will contain (host, port) for IPv4 connections

TCP Server in Python

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

- An infinite while loop is used to loop over blocking calls to conn.**recv**().
- This reads whatever data the client sends us
- If conn.**recv**() returns an empty bytes object, then the client closed the connection and the loop is terminated

TCP Server in Python

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

- An infinite while loop is used to loop over blocking calls to conn.**recv**()
- This reads whatever data the client sends us
- If conn.**recv**() returns an empty bytes object, then the client closed the connection and the loop is terminated
- The **with** statement automatically closes the socket at the end of the block

TCP Server in Python

```
def prepareReply(inData):
    #...
import socket
HOST = "127.0.0.1"
PORT = 65432

with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print("Connected by", addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            response = prepareReply(data)
            conn.sendall(response)
```

- The received data is used to generate a reply (via my “secret” function)
- Response is sent as one “block” using **sendall**()

TCP Client in Python

```
import socket, sys
HOST = "127.0.0.1"
PORT = 65432

inputStr = sys.argv[1]
with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(inputStr.encode())
    data = s.recv(1024)

print("Received", repr(data.decode()))
```

- Read given command line argument

TCP Client in Python

```
import socket, sys
HOST = "127.0.0.1"
PORT = 65432

inputStr = sys.argv[1]
with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(inputStr.encode())
    data = s.recv(1024)

print("Received", repr(data.decode()))
```

- Create a TCP socket
- Same arguments as for the server

TCP Client in Python

```python
import socket, sys
HOST = "127.0.0.1"
PORT = 65432

inputStr = sys.argv[1]
with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(inputStr.encode())
    data = s.recv(1024)

print("Received", repr(data.decode()))
```

- Connects to the server (3-way handshake)
- Sends the cmd line string (as a byte-like object) to the server

TCP Client in Python

```
import socket, sys
HOST = "127.0.0.1"
PORT = 65432

inputStr = sys.argv[1]
with socket.socket (socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(inputStr.encode())
    data = s.recv(1024)

print("Received", repr(data.decode()))
```

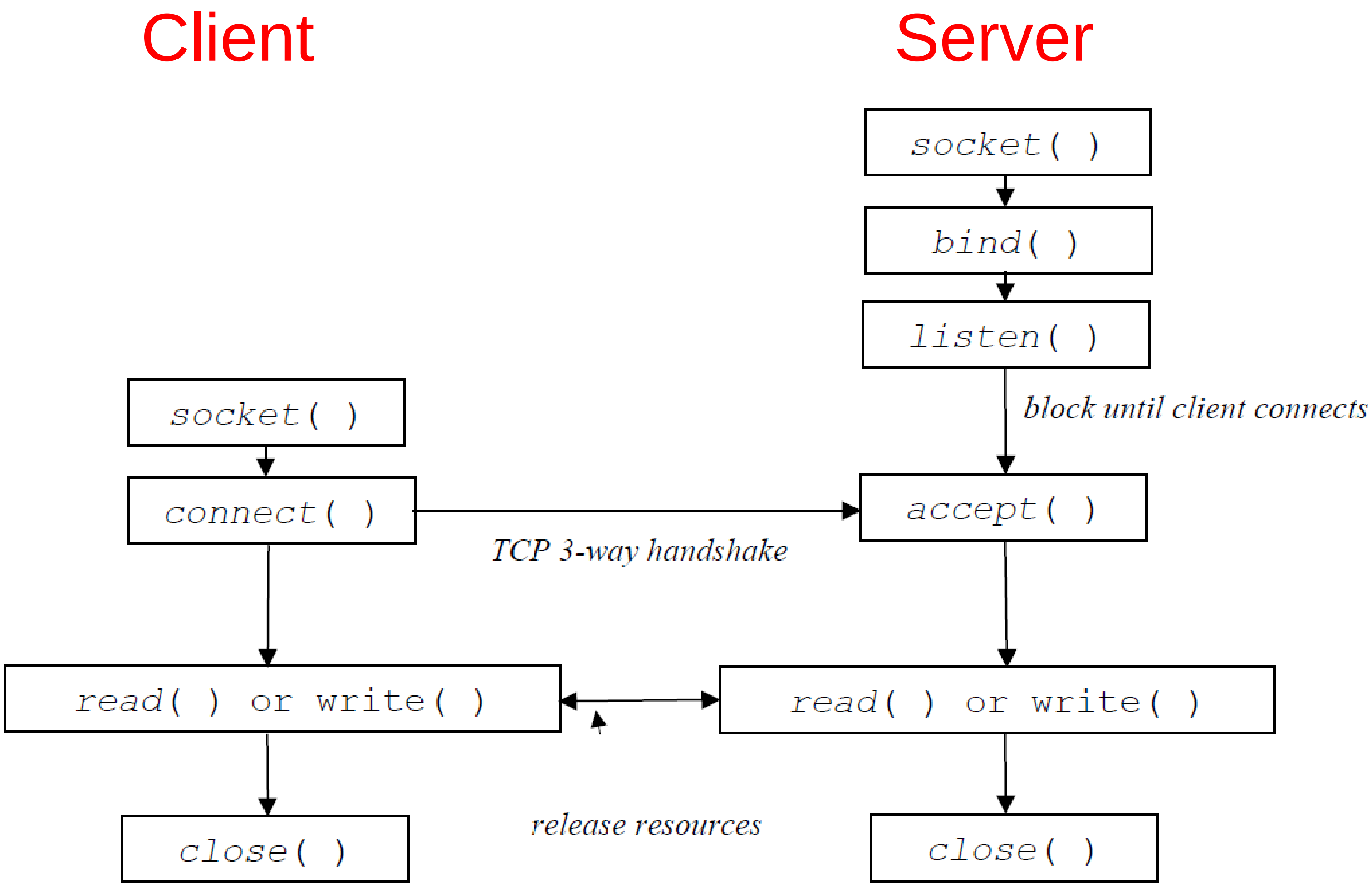
- Receive data from the server
- Object **socket** is closed automatically

- Converts the received data to string and prints it

It’s Demo Time!

- Server: Ubuntu, Client: Windows
- Using *Windows Subsystem for Linux* (link)
  - For graphical UI under linux (using X11)…
    - Install (on Windows) Xming X server (link)
    - Under linux, add to your “.profile” (in home dir):
      - export DISPLAY=:0
- Linux, starting server:
  - cd /mnt/c/Artur/OneDrive/AOD/Academic/Lehre/18.WS-IVS1/Code/Lectures/PythonSockets
  - python3 bondServer.py
- Windows:
  - Start from Jetbrain’s IDEA or Pycharm

TCP Communication in C - Overview



Stream-based Communication in C

Requesting a connection (client)

```
s = socket(AF_INET, SOCK_STREAM,0)
●
●
connect(s, ServerAddress)
●
●
write(s, "message", length)
```

Listening and accepting a connection (server)

```
s = socket(AF_INET, SOCK_STREAM,0)
●
bind(s, ServerAddress);
listen(s,5);
●
sNew = accept(s, ClientAddress);
●
n = read(sNew, buffer, amount)
```

socket()

AF_INET – communication domain is Internet domain
SOCK_STREAM – specifies stream communication
0 – lets the system select the protocol (here TCP)

listen(s, N)

N – maximum number of request for connections to be queued at this socket

Important:
here a new („connection“) socket is created (with its own port)

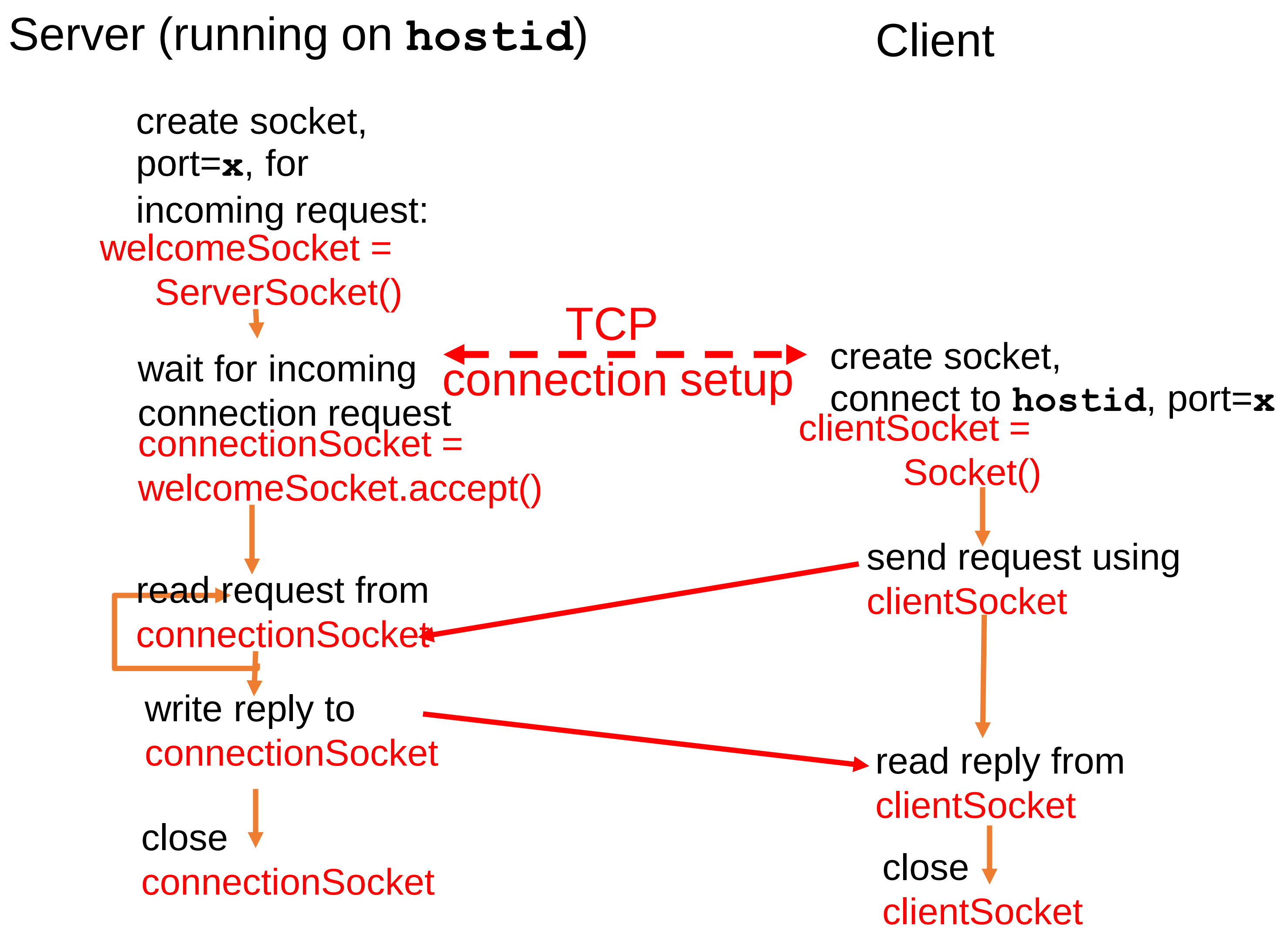
Sockets Programming: TCP with Java [Partially Skipped]

Sockets with TCP in Java

- 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)

Client/server socket interaction: Java



Example: Java client (TCP)

Create input stream

Create client socket, connect to server

Create output stream attached to socket

```
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());
```

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

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

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

TCP observations & questions

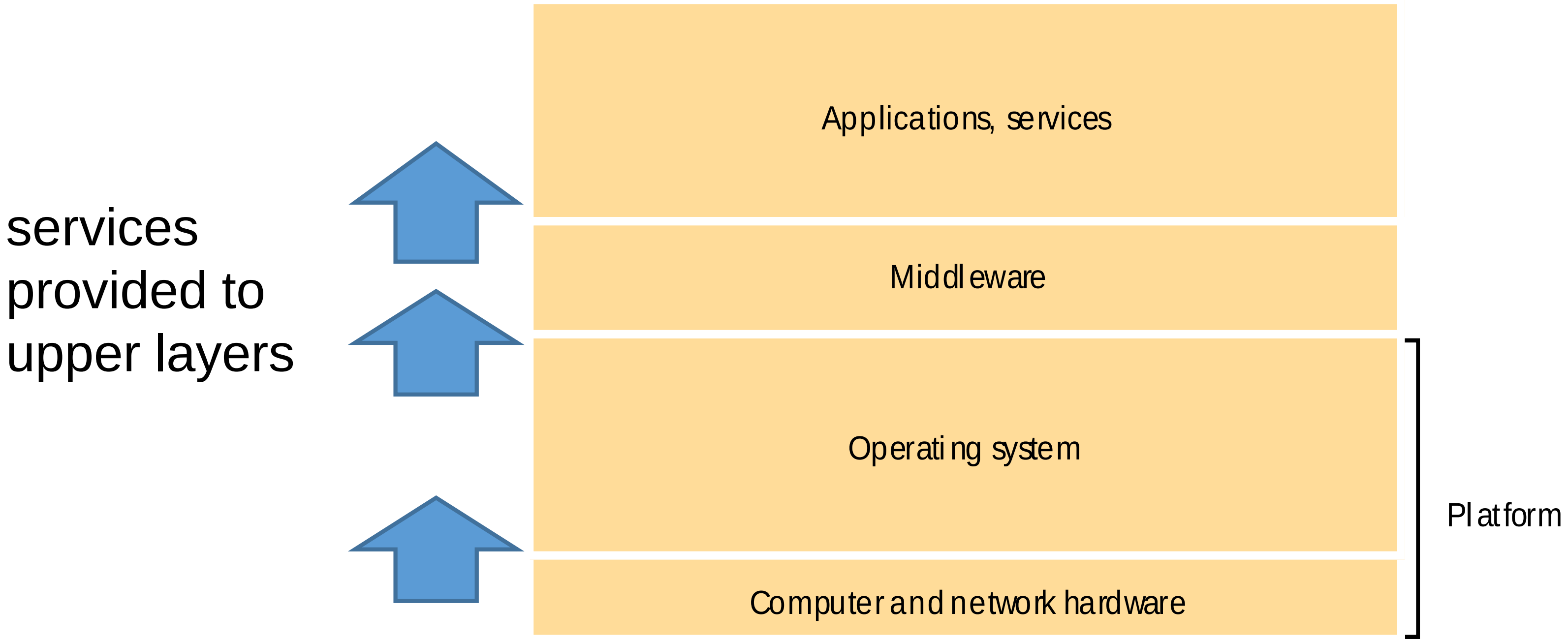
- Server has two types of sockets:
  - ServerSocket and Socket
- When client knocks on serverSocket's "door," server creates connectionSocket and completes TCP connection setup
- Dest IP and port are not explicitly attached to segment
- Can multiple clients use the server?

Thank you.

Additional Slides: Architectures and Topologies in Distributed Systems

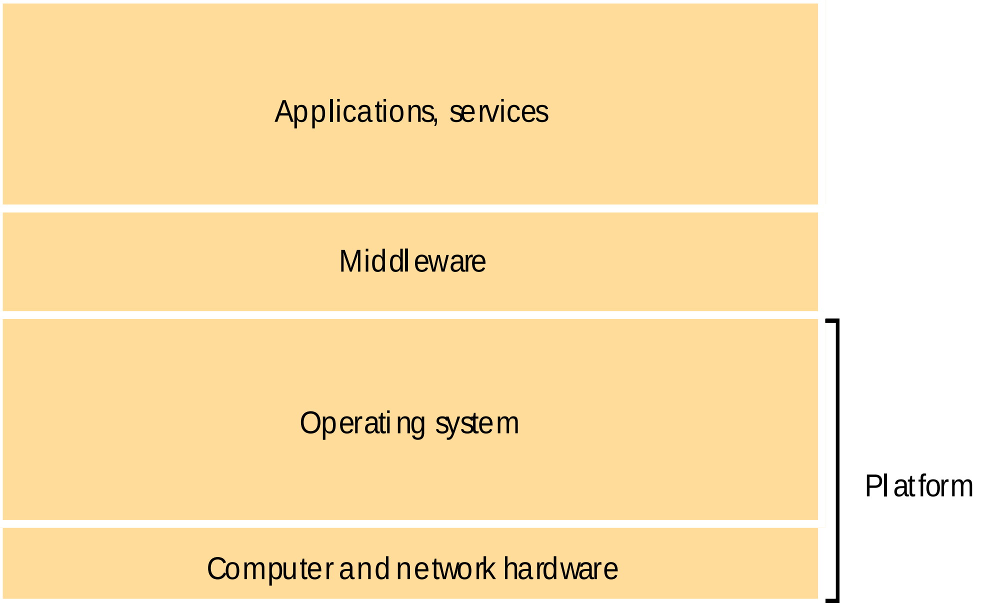
Layers

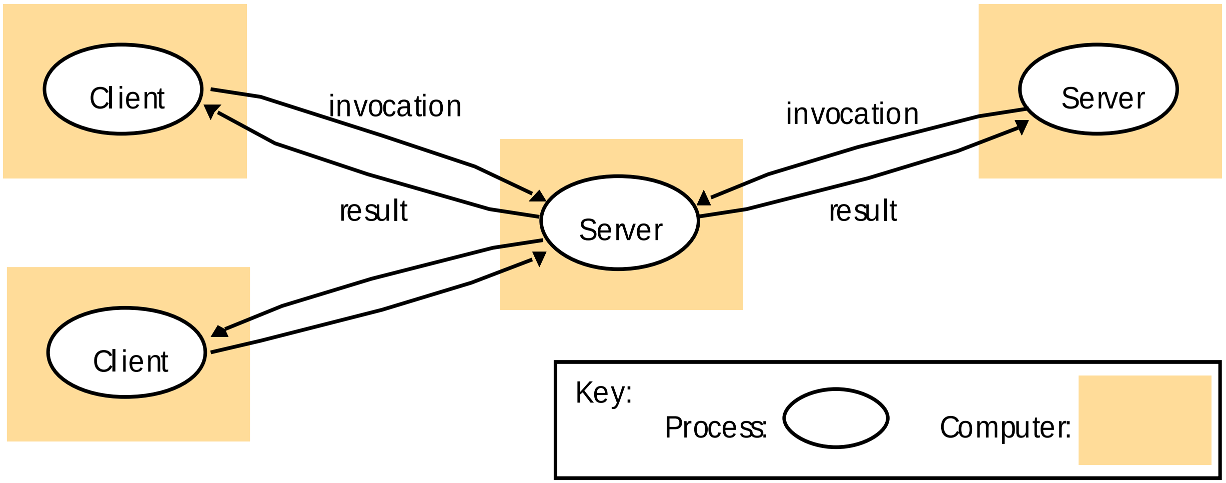
- Reduce the complexity of systems by designing them through **layers** and **services**
  - **Layer**: group of closely related and highly coherent functionalities
  - **Service**: functionality provided to an upper layer



Typical Layers

- **Platform**: Hardware and operating system
  - e.g. x86 with Linux vs. SPARC with Solaris
- **Middleware**: provides communication and resource sharing, e.g.
  - gRPC / protocol buffers (Google)
  - Java Remote Method Invocation (Java RMI)





- Examples:
  - http server: client (browser) requests page, server delivers page
  - file server: client requests file (or parts of it), server delivers data
  - Google apps: storage and processing on Google servers

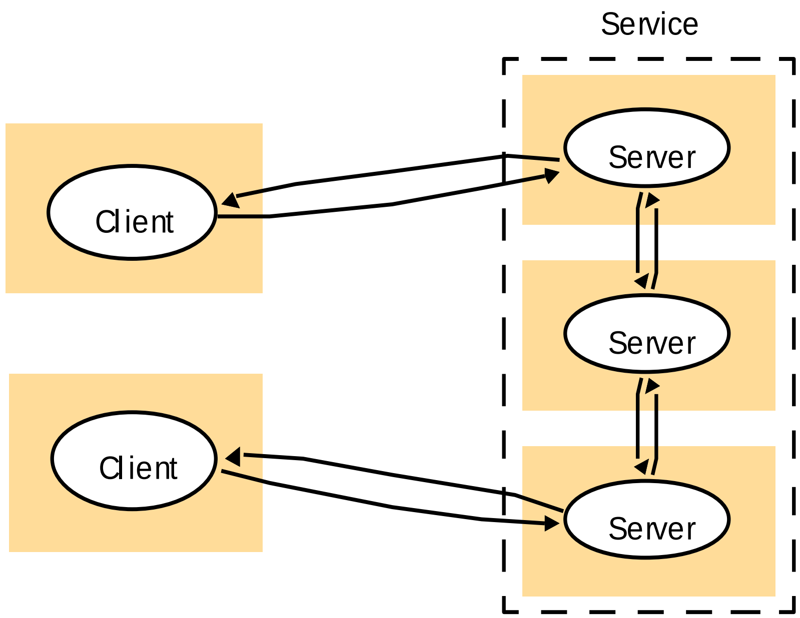
Client – Server Architecture

- **Client**: participant which wants to access data, use resources or perform operations on remote computer
- **Server**: participant managing data and all other shared resources; allows clients access to resource and performs computation

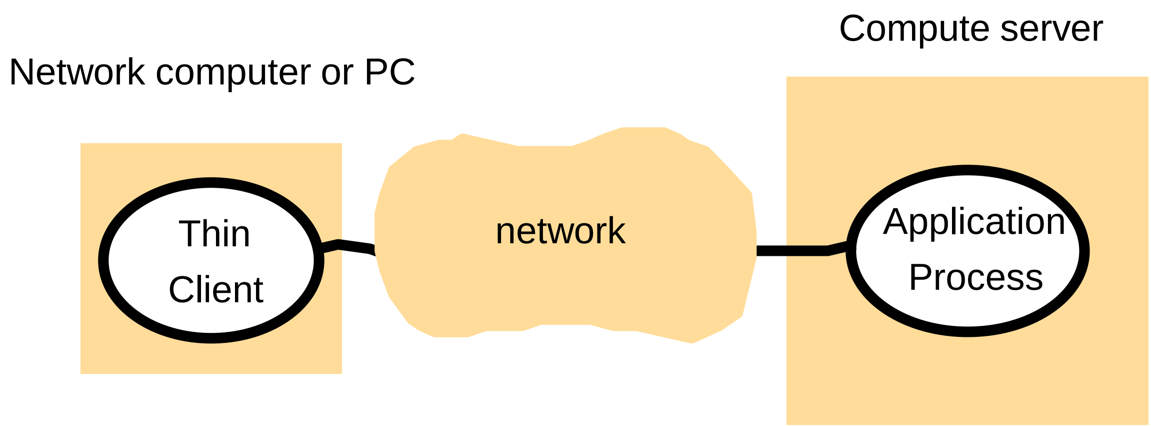
Client – Server Architecture

Variation:

- Service is provided by multiple servers



- Reasons
  - Scalability: e.g. Google search provided by thousands of servers
  - Fault tolerance: replicated services are more robust against failures



- Reasons
  - Low cost of clients
  - Continuous software update at server
- Examples
  - Google docs / spreadsheets (thin client = browser)
  - X11 display protocol in linux/unix
  - Dumb terminals in local network (old)

Client – Server Architecture

Thin clients and compute servers

- Thin clients is used merely for presentation and input collecting
- The „ultimate“ form of client/server paradigm

Additional Slides:
Interprocess Communication

IPC Types – Example Windows

- **Mailslots**
  - Mailslots provide one-way communication
    - Any process that creates a mailslot is a mailslot server
    - Other processes, called mailslot clients, send messages to the mailslot server (by writing a message to its mailslot)
  - The mailslot saves the messages (in FIFO fashion) until the mailslot server has read them
  - A process can be both a mailslot server and a mailslot client, so two-way communication is possible using multiple mailslots

Pipes Example „**who | sort**“

Adapted from Wikipedia:
http://de.wikipedia.org/wiki/Pipe_(Informatik)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void){
    int pipe_verbindung[2];
    pipe(pipe_verbindung);
    if (fork()==0){                              // code child process 1
        dup2(pipe_verbindung[1],1);              // connect „write-to“ end to std-out
        close (pipe_verbindung[0]);              // close the „read-from“ end
        execlp("who","who",NULL);                // execute „who“
    } else if (fork()==0){                       // code child process 2
        dup2(pipe_verbindung[0],0);              // connect „read-from“  to std-in
        close(pipe_verbindung[1]);               // close the „write-to“ end
        execlp("sort","sort",NULL);
}}
```

IPC Types – Mailboxes / MP / RPC

- **Message queue / mailbox**: most OSs
  - Typically performed by an off-the-shelf message-queuing software
  - application registers a routine that "listens" for messages placed onto the queue
  - second and subsequent applications may connect to the queue and transfer a message onto it
- **Message passing** and **Remote Procedure Calls**:
  - Via Middleware (MPI, Java RMI, CORBA, …)
  - Communication via sending of messages to recipients
  - (Much) more on this later

IPC Types – Example Windows

- **Clipboard**
- **COM** and **OLE** – (COM = Component Object Model)
  - “OLE supports compound documents and enables an application to include embedded or linked data that, when chosen, automatically starts another application for data editing.”
  - “COM objects provide access to an object's data through one or more sets of related functions, known as interfaces.”
- **Data Copy**
  - „Data copy enables an application to send information to another application using the WM_COPYDATA message.”
  - “This method requires cooperation between the sending application and the receiving application.”

IPC Types – Example Windows

- **DDE** (Dynamic Data Exchange)
  - DDE is an extension of the clipboard mechanism
  - It is also usually initiated by a user command, but it often continues to function without further user interaction
  - DDE is not as efficient as newer technologies
- **File Mapping**
  - See Memory-mapped file (mmap)
- **Anonymous Pipes**
- **Named Pipes**

Additional Slides:
UDP Sockets in Java

Running Example

- Client:
  - User types line of text
  - Client program sends line to server
- Server:
  - Server receives line of text
  - Capitalizes all the letters
  - Sends modified line to client
- Client:
  - Receives line of text
  - Displays

Client/server Socket Interaction: UDP

Server (running on **`hostid`**)

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

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)

```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();
```

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

Send datagram to server

Read datagram from server

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

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);
clientSocket.close();
}
}
```

Example: Java server (UDP)

```
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());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

        String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

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

serverSocket.send(sendPacket);
 }
}
}
```

Get IP addr port #, of sender

Create datagram to send to client

Write out datagram to socket

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

UDP Observations & Questions

- Both client server use DatagramSocket
- Dest IP and port are explicitly attached to segment
- Can the client send a segment to server without knowing the server's IP address and/or port number?
- Can multiple clients use the server?