Universität Stuttgart

Institute of Parallel and
Distributed Systems (IPVS)

Universitätsstraße 38
D-70569 Stuttgart

Net-based Applications: Network Programming

Adnan Tariq

(Based on the slides of Dr. Frank Dürr)

Sockets

- Transport services
- Client/server model
- Sockets API
- Create, bind, close sockets
- Send and receive data
- Queues for connections
- Examples in C and Java

Transport Services in the Internet

Transport Service offers inter-process communication

- End-to-end protocol connecting applications to each other
- Two classes of transport protocols: connection-less & connection-oriented

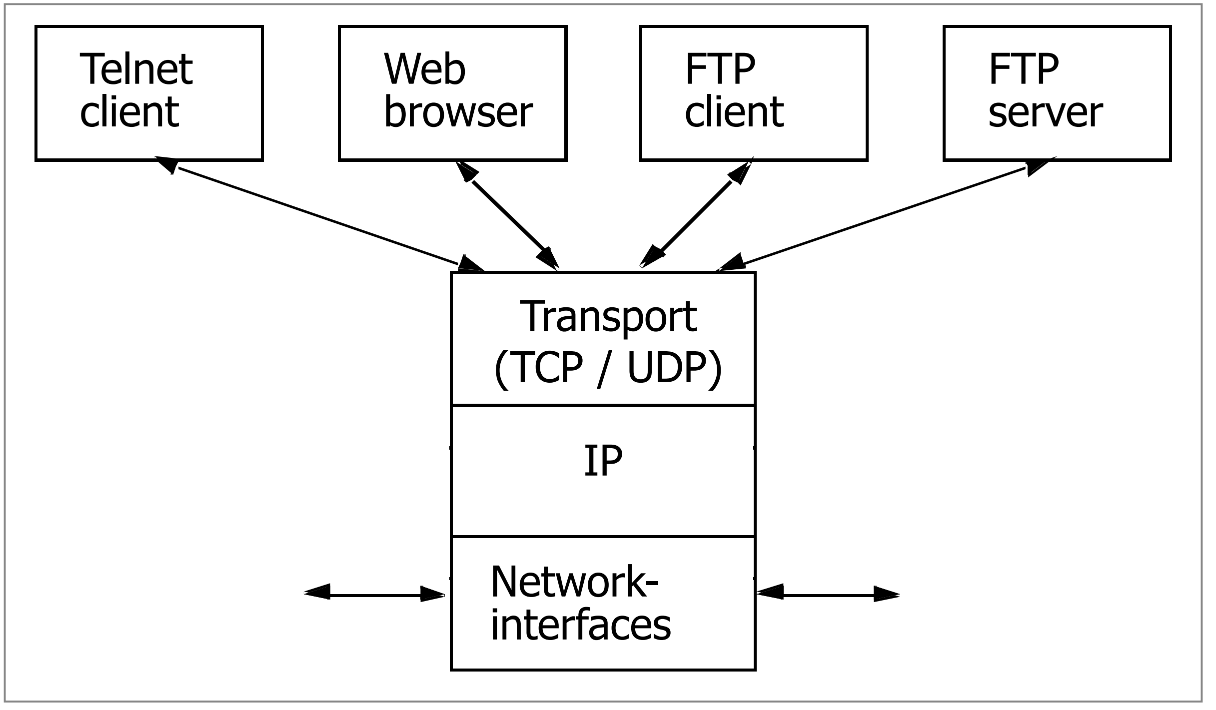
Connection-less protocol: UDP

- No reliable communication, no order guarantees
  - Correctness of delivered messages guaranteed with checksums
- No flow control

Connection-oriented protocol: TCP

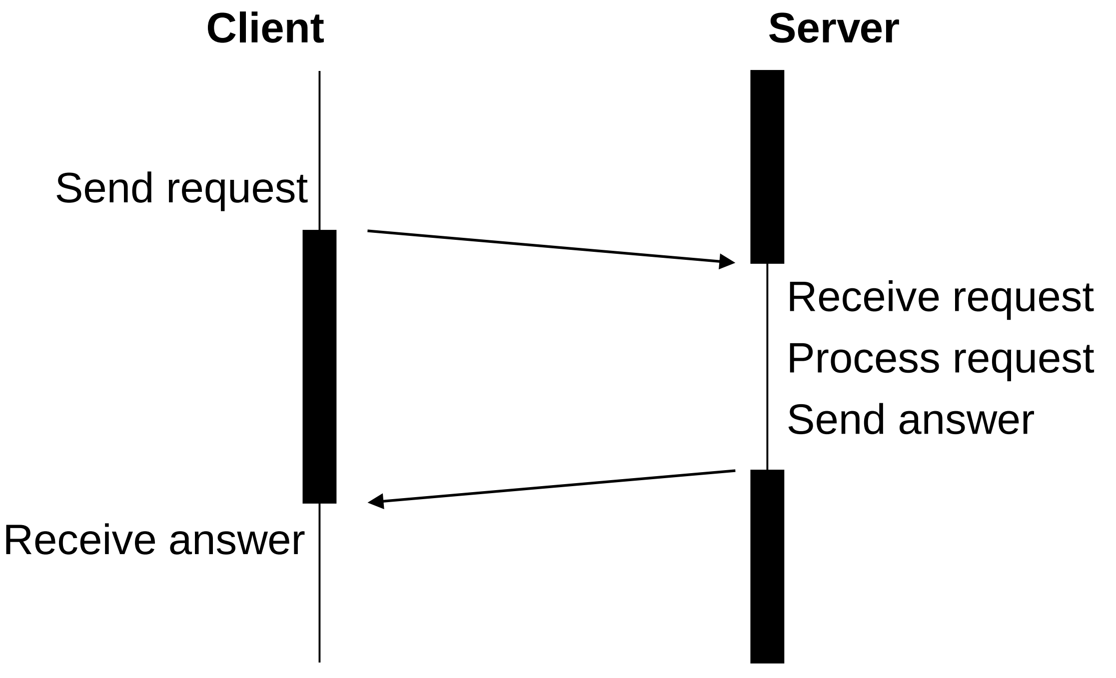
- Reliable, ordered communication
- Byte-oriented communication
- Flow control
- Congestion control

Addressing



- Network layer offers computer/computer communication and thus addressing of computers (network interfaces) → IP addresses
- Applications need inter-process communication and thus the addressing of services/applications → ports

Client/Server Model



Server can serve several clients sequentially or in parallel

➔ Server execution models

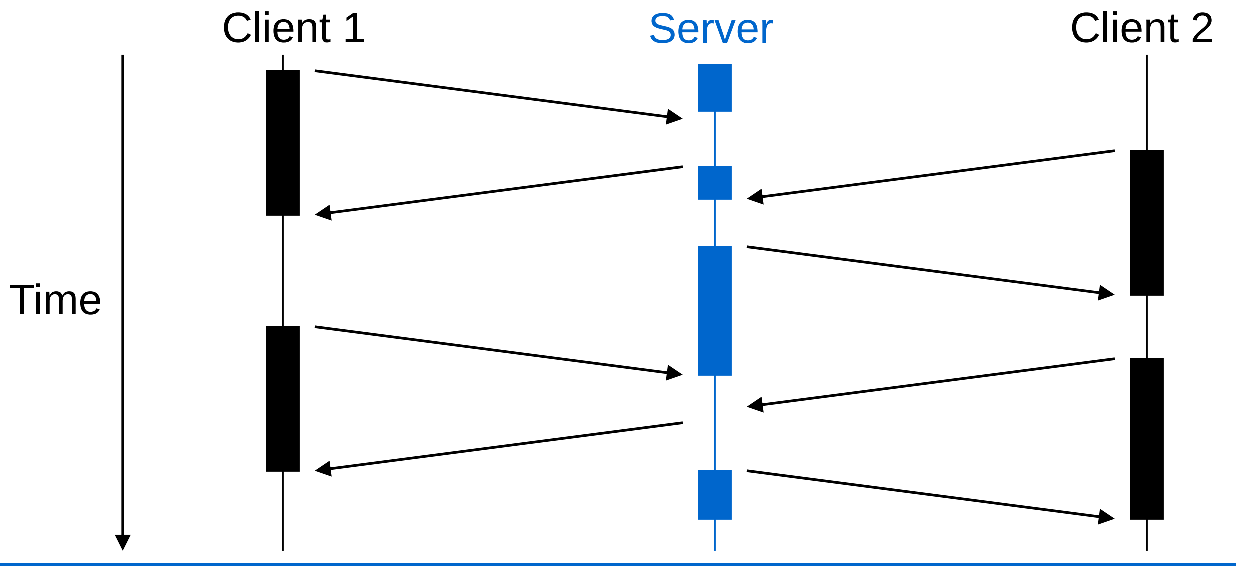
Server Execution Models

There are three server execution models:

1. Independent requests with single-threaded server
2. Session-oriented processing with single-threaded server
3. Parallel processing with multi-threaded server

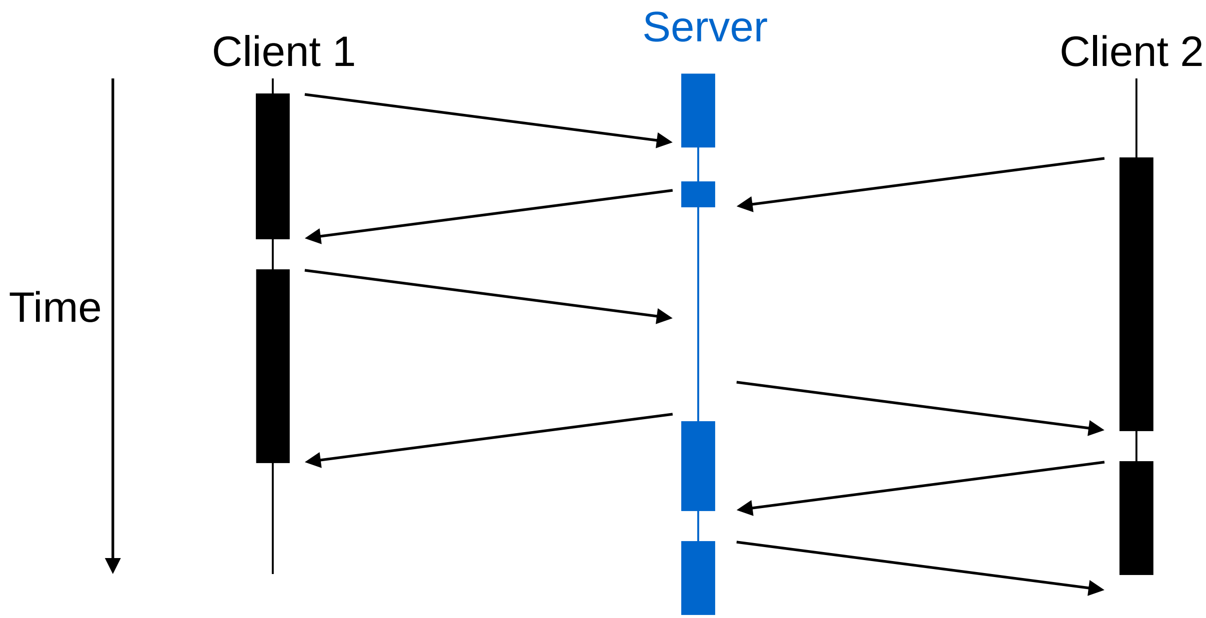
1. Independent Requests (Single-threaded Server)

- Requests are performed sequentially
- Typically no state is to be kept between requests
- Interleaved processing of requests from different clients
  - Server processes only one request at a time
  - To the clients, this looks like parallel processing of several clients

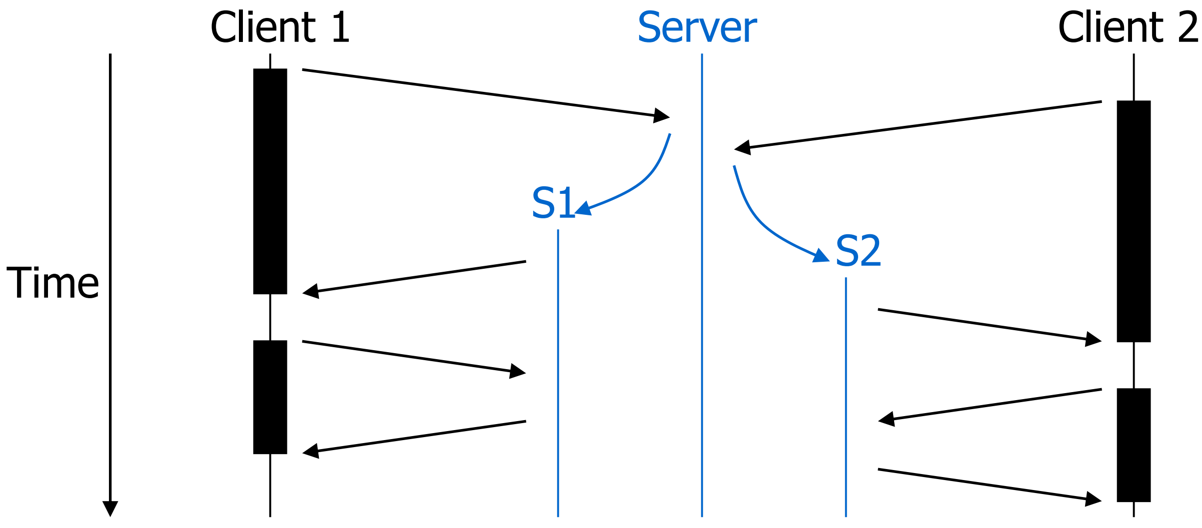


2. Session-oriented Processing (Single-threaded Server)

- Clients send requests within sessions
- Server executes requests sequentially
- Server executes all requests of single session in a row
- State for session can be maintained easily
- Reasonable and simple approach with connection-oriented transport protocol



3. Parallel Processing (Multi-threaded Server)



Session-oriented Processing

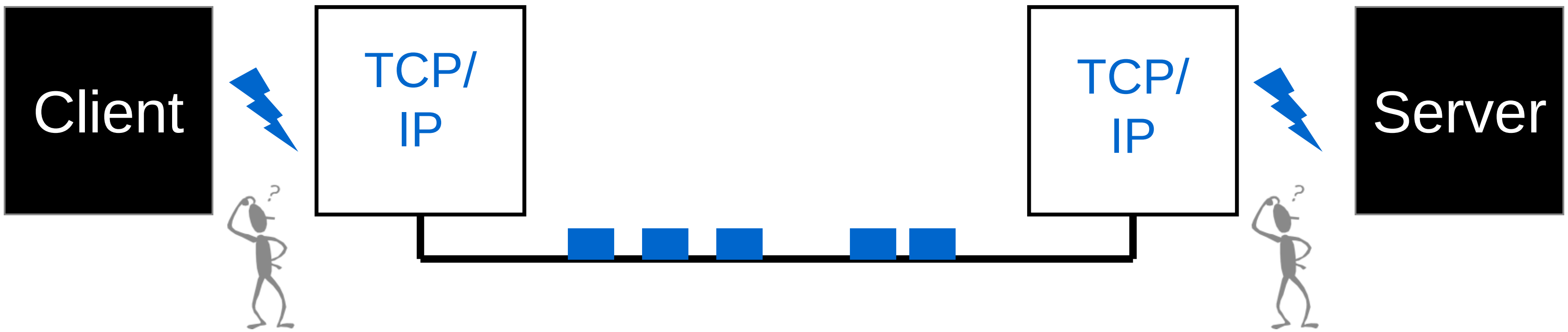
- Primary server accepts session
- For each session a secondary server is created (or assigned)

Independent requests

- All requests are sent to a single port
- Any idle server receives and processes request

Sockets: A Transport Service API

- Problem:



- Interface between application- and protocol-software
- Sockets are an API (Application Program Interface)
- API is not standardized for TCP/IP:
  - Because: One interface may not suit all applications
  - Anyway: Sockets are widely used (de facto standard)

API – Design Criteria

- In UNIX: I/O-operations usually follow the open-read-write-close paradigm
- This is not sufficient for complex interactions with network protocols, because:
  - ⇨ API must support both servers (passively waiting for requests) and clients (actively initiating transfers)
  - ⇨ To support datagrams: reasonable to give target address with datagram (not with open operation)
- API should not be restricted to TCP/IP

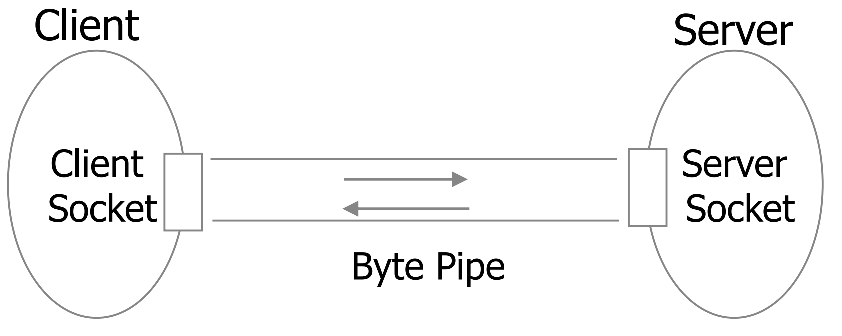
Socket Abstraction (1)

Sockets are communication end-points

- Processes send and receive data via their local sockets

Connection-oriented paradigm:

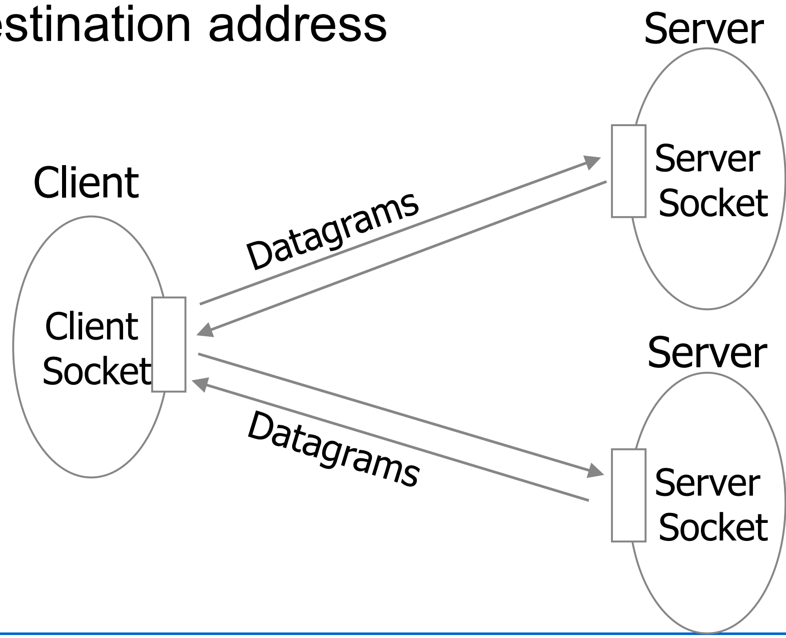
- Two processes establish bi-directional byte pipe by connecting a pair of their sockets
- Sender writes a number of bytes to the corresponding local socket
- Receiver reads a number of bytes from the corresponding local socket



Socket Abstraction (2)

Connection-less paradigm

- Sender sends datagrams through local sockets
  - Datagram includes destination address (IP-address, port number)
- Receiver receives datagrams through the local socket that is bound to the datagram's destination address

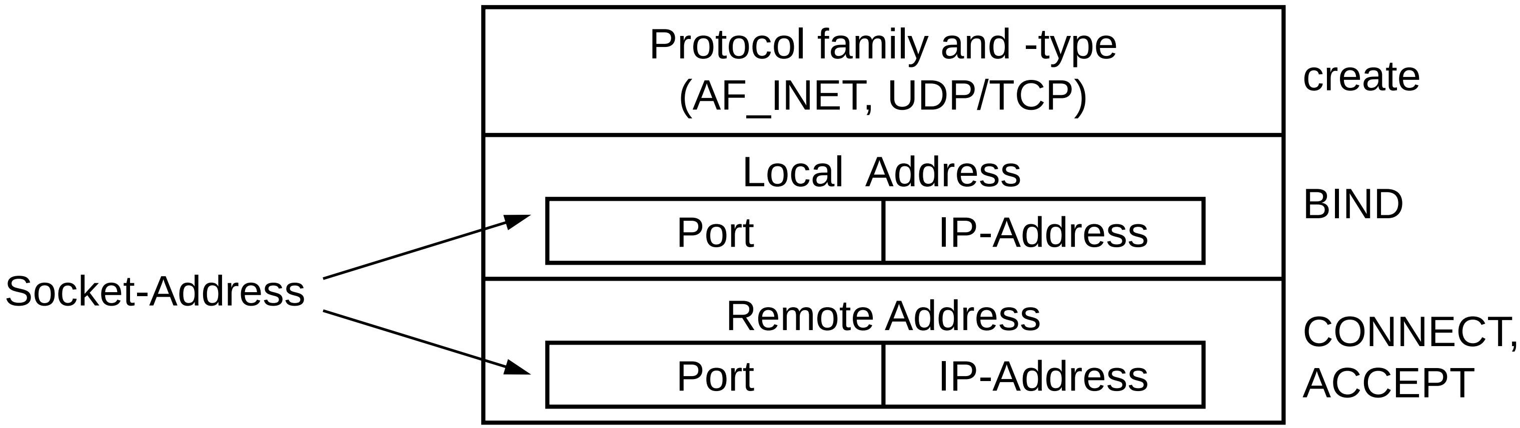


Socket Primitives – Overview

- Socket access: similar to file access mechanism in UNIX:
  - In UNIX "everything is a file"
  - Access is identified via *file descriptors.* A *file descriptor* is an integer, which is connected to the "real" file → FIFO, Pipe, Terminal, … , or network connection!
- Difference to files: sockets can be created without binding to a specific target address!

Files	Sockets	
	UDP	TCP
open	bind	connect, bind, accept
write	sendto	write
read	recvfrom	read
close	close	close, shutdown

Socket – Data Structure



- Specify remote address with *connect* (can be omitted with UDP)
- Specify the recipient with every packet (*sendto*)
- Missing arguments are added by the OS
- Wildcards possible for IP-addresses
- Outgoing packets contain complete information
- Socket that is responsible for incoming packets is determined by information in packet and socket

Socket Create and Close

result = socket(pf, type, protocol) ⇦ create signature for socket

close(socket) ⇦ close signature for socket

pf:

Protocol family (TCP/IP, AppleTalk, UNIX file system, ...)

type:

Type of communication

- Reliable and connection oriented service `(SOCK_STREAM)`
- Connection-less service `(SOCK_DGRAM)`
- *Raw* to permit privileged programs access to all protocol elements `(SOCK_RAW)`

protocol:

Specify protocol further (if *type* is not unique)

result:

Socket descriptor

Bind Socket to Local Address

bind(socket, localaddr, addrlen)

socket:

socket descriptor

localaddr:

local address structure, e.g. in TCP/IP: port-number and IP-address

addrlen:

address length (Bytes)

Connect Socket to Target Address

connect(socket, destaddr, addrlen)

socket:

socket descriptor

destaddr:

destination address structure, e.g. in TCP/IP: port-number and IP-address

addrlen:

address length (Bytes)

- Required for connection-oriented services
- Possible for connection-less services
  - No need to specify target address with every send operation

Send Data

write(socket, buffer, length)

socket:

socket descriptor

buffer:

address pointing to data to be sent

length:

number of bytes to be sent

- *send*, *sendto*, *sendmsg*, *write,* and *writev* are possible sending operations
  - *send*, *write* and *writev* only for connected sockets

Receive Data

read(socket, buffer, length)

socket:

socket descriptor

buffer:

address where to store received data

length:

maximal length of bytes to be received

- *read, readv, recv, recvfrom,* and *recvmsg* are possible receive operations

Request Queue

Problem in a client/server system:

What happens if the next client wants to connect before the current client is served?

Solution:

To avoid refused connections, connection requests can be stored in a waiting queue.

listen(socket, qlength)

socket:

Socket Descriptor

qlength:

length of the waiting queue for this socket

Accepting a Connection (1)

Problem in a client/server system:

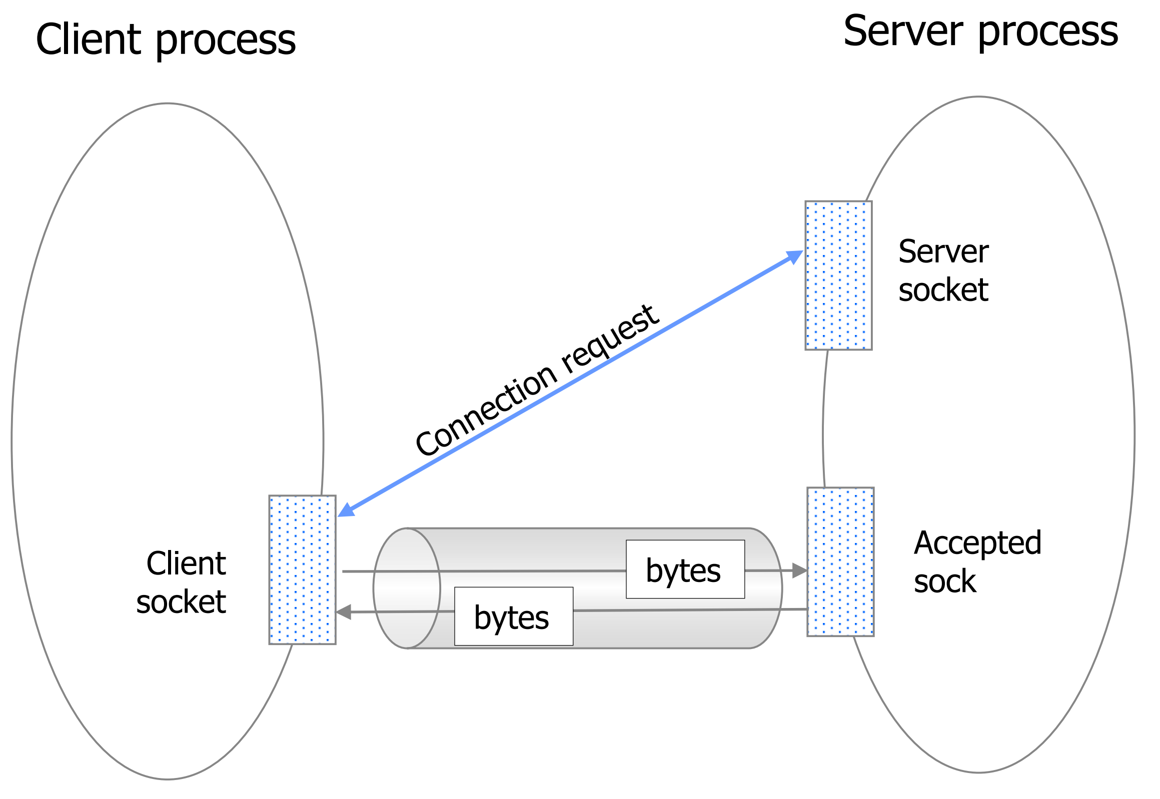
A server executes the operations *socket*, *bind,* and *listen*. How to establish a concrete connection?

Solution:

accept operation: newsock = accept(socket, addr, addrlen)

- When a connection request comes from a client, *addr* and *addrlen* are set (by the system) according to the client address
- A new socket is created. This socket is connected to the client. It is now possible to communicate with the client through *newsock*
- The original socket stays open and continues to listen to connection requests.

Accepting a Connection (2)



Sockets in Practice

There are further operations:

- Support of (single-threaded) servers that offer several services
  - Select readable socket from a set of server sockets
- Socket options (e.g. timeouts)
- ...

In practice:

- The operations described above do not necessarily exist exactly like that in every OS / are encapsulated in library functions
- Examples for C and Java following

C Server (1)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main() {
   const int MAXBUF = 1024;
   const short SERVER_PORT = 4444;
   int serverSocketID, clientSocketID;
   struct sockaddr_in serverSocketAddr, clientSocketAddr;
   int addrLen = sizeof(struct sockaddr_in);
   char buffer[MAXBUF];
   int msgLen;

   serverSocketID = socket(PF_INET, SOCK_STREAM, 0);

   memset(&serverSocketAddr,0,addrLen);
   serverSocketAddr.sin_family = AF_INET;
   serverSocketAddr.sin_addr.s_addr = INADDR_ANY;
   serverSocketAddr.sin_port = htons(SERVER_PORT);
   bind(serverSocketID, (struct sockaddr *)&serverSocketAddr, addrLen);
```

C Server (2)

```
  listen(serverSocketID, 5);
  printf("Server running on port %d. Waiting for connections.\n",
        SERVER_PORT);
  do {
        clientSocketID = accept(serverSocketID,
          (struct sockaddr *) &clientSocketAddr, &addrLen);
        printf("Accepting connection.\n");
        msgLen = recv(clientSocketID, buffer, MAXBUF, 0);
        buffer[msgLen] = '\0';
        printf("Received message: %s\n", buffer);

        /* this is the place where you could do something useful */

        send(clientSocketID, buffer, msgLen, 0);
        close(clientSocketID);
        printf("Closed connection. Waiting for new work.\n");
  } while(1); // endless loop
}
```

C Client (1)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main() {
  const int MAXBUF = 1024;
  const short PORT = 4444;
  char *message = "Hello World!";
  int clientSocketID;
  struct sockaddr_in serverSocketAddr;
  int addrLen = sizeof(struct sockaddr_in);
  struct hostent *serverInfo;
  char buffer[MAXBUF];
  int msgLen;

  clientSocketID = socket(PF_INET, SOCK_STREAM, 0);
```

C Client (2)

```
  memset(&serverSocketAddr,0,addrLen);
  serverInfo = gethostbyname("taranis");
  serverSocketAddr.sin_family = AF_INET;
  serverSocketAddr.sin_addr = *((struct in_addr *) serverInfo->h_addr);
  serverSocketAddr.sin_port = htons(PORT);

  connect(clientSocketID, (struct sockaddr *) &serverSocketAddr,addrLen);

  send(clientSocketID, message, strlen(message)+1, 0);

  msgLen = recv(clientSocketID, buffer, MAXBUF, 0);
  buffer[msgLen] = '\0';
  printf("Received the following message from server: %s\n",buffer);

  close(clientSocketID);
}
```

C Server – Parallelism

```
do {
   clientSocketID = accept(serverSocketID, (struct sockaddr *)
          &clientSocketAddr, &addrLen);
   pid_t pid = fork();
   if (pid == -1) {
          // error
          perror("Could not fork");
   } else if (pid == 0) {
          // child process:
          msgLen = recv(clientSocketID, buffer, MAXBUF, 0);
          // this is the place where you could do something useful
          close(clientSocketID);
          exit(0); // child process exits after serving request
   } else {
          // fork returns child process id to parent process, i.e. != 0
          // parent process:
          close(clientSocketID); // close duplicate of client socket
   }
   // Avoid zombies calling wait, waitpid, ...
} while(1); // endless loop
```

Protocol Independence

The previous code is protocol dependent: tailored to IPv4

- Using explicitly IPv4 definitions and 32 bit IPv4 addresses structures in code:
  - `struct sockaddr_in, PF_INET, AF_INET`
- New protocols required source code modifications to use new definitions and structures, e.g. IPv6 with 128 bit addresses
  - `struct sockaddr_in6, PF_INET6, AF_INET6`
- How can we write protocol agnostic code?
  - Compatible with IPv4, IPv6, etc. *without* code modifications
  - Application only specifies needs (“hints”) instead of concrete protocol
    - “I need reliable byte stream to host” (I don’t care which protocol)

→ This is possible using function `getaddrinfo()`

User-friendly Name Strings – Lookup

- Resolve names using DNS, resolve services using file /etc/services
  - Replaces `gethostbyname()`

```
int getaddrinfo(const char *host, const char *service,
         const struct addrinfo *hints, struct addrinfo **result);
```

- Parameters
  - `host`: hostname or address string
  - `service`: service name or decimal port number string
  - `hints`: null pointer or pointer to `addrinfo` with type of information requested
  - `result`: linked list of `addrinfo`, allocated by `getaddrinfo`, must be freed with `void freeaddrinfo( struct addrinfo *ai);`

```
struct addrinfo {
  int ai_flags;      /* e.g., AI_PASSIVE */
  int ai_family;       /* AF_xxx */
  int ai_socktype;     /* SOCK_xxx */
  int ai_protocol;     /* 0 or IPPROTO_xxx */
  socklen_t ai_addrlen; /* length of
                           ai_addr */
  char *ai_canonname;   /* canonical name */
  struct sockaddr *ai_addr;   /* socket
                                 address */
  struct addrinfo *ai_next;   /* next list
                                 element */
};
```

- Return value: 0 if OK, nonzero on error
- Results contain **all** address/port information for calling `socket`, `bind`, `connect`, `sendto`

Protocol-independent Client Code

```
int sockfd; struct sockaddr_storage addr; socklen_t addrlen;

void init_socket(const char *hostname, const char *service) {
  struct addrinfo hints, *res, *ressave;
  bzero(&hints, sizeof(hints));
  hints.ai_family = AF_UNSPEC; /* any protocol will match our query */
  hints.ai_socktype = SOCK_STREAM /* reliable byte stream */
  if ( getaddrinfo(hostname, service, &hints, &res) != 0 ) handle_error();
  ressave = res;
  do { /* try to open a socket with each result entry until one works: */
    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sockfd < 0) continue; /* ignore this one */
    if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0) break;
      /* success */
    close(sockfd); /* ignore this one */
  } while ( (res = res->ai_next) != NULL );
  if (res == NULL) handle_error();
  freeaddrinfo(ressave);
  /* now you may use sockfd to communicate */
}
```

without any protocol specific code!

Protocol Independent Server Code

```
int sockfd; struct sockaddr_storage addr; socklen_t addrlen;

void init_socket(const char *hostname, const char *service) {
  struct addrinfo hints, *res, *ressave;
  bzero(&hints, sizeof(hints));
  hints.ai_flags = AI_PASSIVE; /* server passively waiting for conn. */
  hints.ai_family = AF_UNSPEC; /* any protocol will match our query */
  hints.ai_socktype = SOCK_STREAM /* reliable byte stream */
  if ( getaddrinfo(hostname, service, &hints, &res) != 0 ) handle_error();
  ressave = res;
  do { /* try to open a socket with each result entry until one works: */
    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sockfd < 0) continue; /* ignore this one */
    if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0) break;
       /* success */
    close(sockfd); /* ignore this one */
  } while ( (res = res->ai_next) != NULL );
  if (res == NULL) handle_error();
  freeaddrinfo(ressave);
  /* now you may use sockfd to communicate */
}
```

without any protocol specific code!

Java Client (1)

```
import java.io.*;
import java.net.*;
public class EchoClient {
  public static void main(String[] args) {
    Socket echoSocket = null;
    PrintWriter out = null;
    BufferedReader in = null;
    try {
      echoSocket = new Socket("taranis", 4444);
      out = new PrintWriter(echoSocket.getOutputStream(), true);
      in = new BufferedReader(new InputStreamReader(
                                  echoSocket.getInputStream()));
    } catch (UnknownHostException e) {
      System.err.println("Don't know about host: taranis.");
      System.exit(1);
    } catch (IOException e) {
      System.err.println("Couldn't get I/O for "
                         + "the connection to: taranis.");
      System.exit(1);
    }
```

Java Client (2)

```
  BufferedReader stdIn = new BufferedReader(
    new InputStreamReader(System.in));
    String userInput;
    while ((userInput = stdIn.readLine()) != null) {
      out.println(userInput);
      System.out.println("echo: " + in.readLine());
    }
    out.close();
    in.close();
    stdIn.close();
    echoSocket.close();
  }
}
```

Java Client (Explanations)

echoSocket = new Socket("taranis", n)

⇨ Creates socket and connects to host *taranis* at Port n

out = new PrintWriter(echoSocket.getOutputStream(), true);

⇨ Get socket output stream and connect to new *PrintWriter* (“true” turns on “autoFlush”)

in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));

⇨ Get socket input stream and open a new *BufferedReader*

To send data to server: write to PrintWriter
To receive data from server: read from BufferedReader

Java Server (1)

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

public class EchoServer {
  public static void main(String[] args) {
   ServerSocket serverSocket = null;
   try {serverSocket = new ServerSocket(4444);}
   catch (IOException e) {
     System.err.println("Could not listen on port: 4444.");
     System.exit(1);
   }
  Socket clientSocket = null;
  try {clientSocket = serverSocket.accept();}
  catch (IOException e) {
     System.err.println("Accept failed.");
     System.exit(1);
   }
```

Java Server (2)

```
   PrintWriter out =
     new PrintWriter(clientSocket.getOutputStream(), true);
   BufferedReader in = new BufferedReader(new
     InputStreamReader(clientSocket.getInputStream()));
   String inputLine, outputLine;
   while ((inputLine = in.readLine()) != null) {
         outputLine = inputLine;
         out.println(outputLine);
         if (inputLine.equals("Bye."))
         break;
   }
   out.close(); in.close();
   clientSocket.close(); serverSocket.close();
   }
}
```

Java Client & Server with UDP

Client

```
// create socket and bind it
DatagramSocket clientSocket = new
   DatagramSocket(6001);
// send one datagram
byte[] data = new byte[1];
data[0] = 42;
DatagramPacket packet =
   new DatagramPacket(data,
   data.length);
// set receiver address
packet.setAddress(
   InetAddress.getByName(
     "localhost"));
packet.setPort(6000);
clientSocket.send(packet);
```

Server

```
// create server socket and bind it
DatagramSocket serverSocket = new
   DatagramSocket(6000);
// allocate space for received
   datagram
byte[] data = new byte[256];
DatagramPacket packet = new
   DatagramPacket(data,
   data.length);
while (true) {
  // receive one datagram
  serverSocket.receive(packet);
  byte[] dataReceived =
    packet.getData();
  int bytesReceived =
    packet.getLength();
  // do something useful with data
}
```

Java Server (Explanations)

serverSocket = new ServerSocket(4444)

⇨ Creates Server Object and listens to clients on port 4444

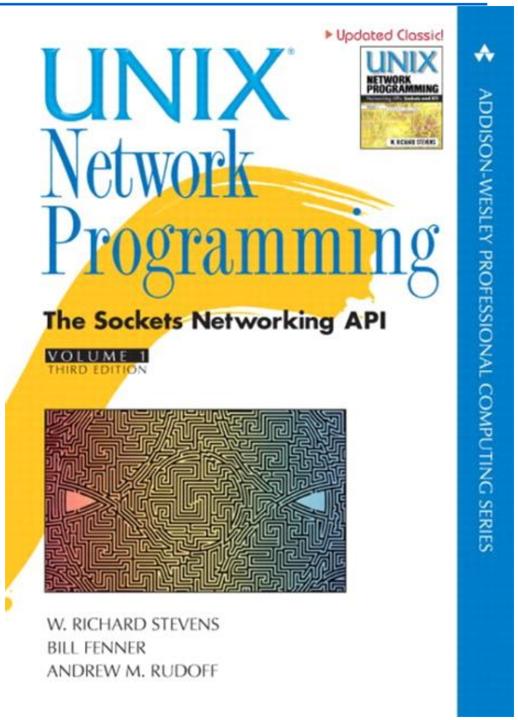
clientSocket = serverSocket.accept();

⇨ *accept*-method waits for client to connect to port 4444.
When a client connected successfully, *accept* returns a new socket

To send data to client: write to PrintWriter
To receive data from client: read from BufferedReader

Further Literature

- Network programming in C:
  - W.R. Stevens, B. Fenner, A.M. Rudoff. UNIX Network Programming, Volume 1: The Sockets Networking API. 3rd Edition. Addison-Wesley. 2004.



- Network programming in Java:
  - Sun Microsystems: *Java Tutorial – Custom Networking Trail*. Available online: http://docs.oracle.com/javase/tutorial/networking/

Summary

- Sockets are an API for the transport layer
  - Transfer byte stream or datagrams
    (connection-oriented and –less transport services)
- Based on the “file philosophy“ in Unix
  - Socket is a file descriptor
- Specific functions for creating, binding, accepting, connecting sockets
- Programming is “low-level”
  - Application has to parse and interpret transmitted data (bytes)
- Often used as basis for “higher“ interactions like RPC