

**Telematics Course:**  
**Introduction to Sockets and Practice**  
(Winter Semester 2003/04)

Telematics group  
University of Göttingen, Germany

---

## Content

- Socket: programming interface for TCP/IP networks
  - Basic idea
  - How to program it
- Practices (optional):
  - Exercises for socket programming
  - UNIX networking configuration files and tools

### Credits:

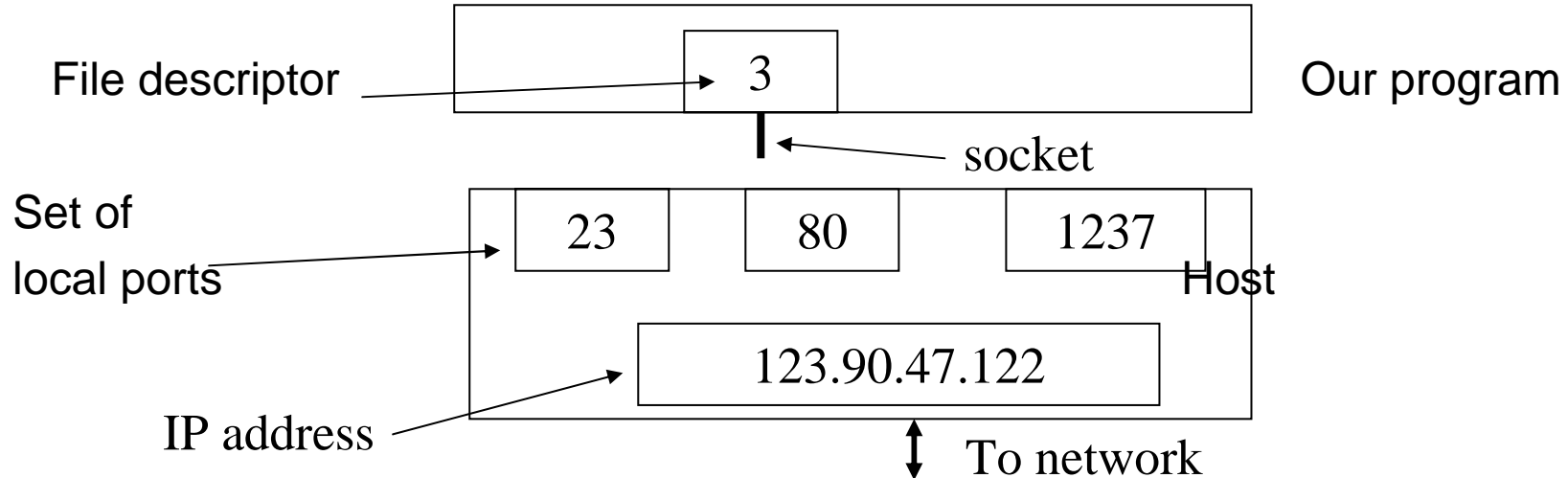
- Jim Kurose & Keith Ross, Computer Networking (2nd ed.)
- Some materials from various online sources

# Overview of Sockets: the TCP/IP user interface

- Socket: Create a new communication end point
- Bind: Attach a local address to a socket
- Listen: Announce willingness to accept connections
  - Give queue size
- Accept: Block the caller until a connection attempt arrives
  - Accept next connection request
- Connect: Actively attempt to establish a connection
- Send: Send some data over the connection
- Recv: Receive some data from the connection
- Close: Release the connection

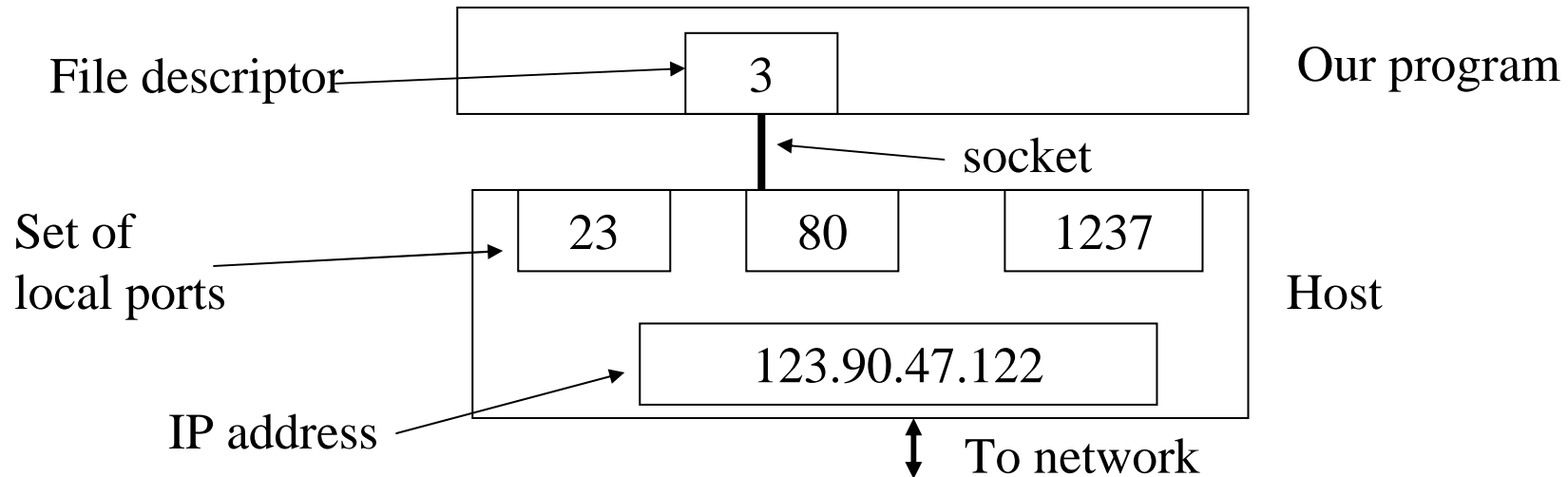
## Step 1: Create a socket

- Suppose we make the call:  
`s = socket( ... assorted parameters here ... );`  
and the result is file descriptor `s = 3`
- We have the following situation:



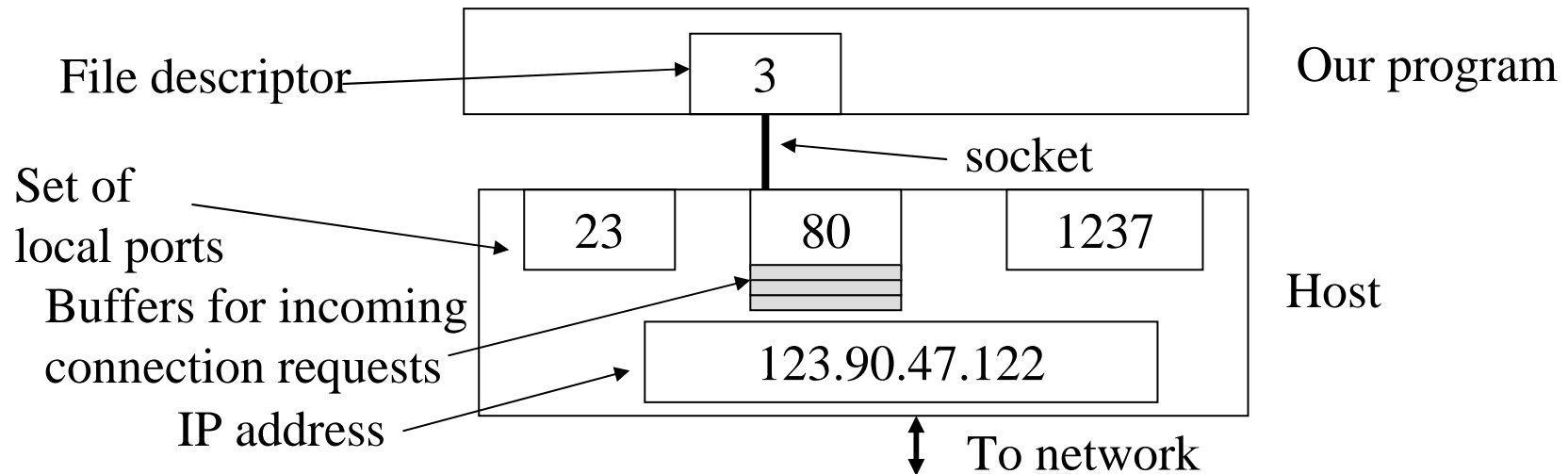
## Step 2: Bind the socket to a port

- Suppose we do:  
`result = bind( ... 80, ... );`  
and the result is not -1 (that is, it worked!)
- We now have the following situation:



## Step 3: Listen for connections

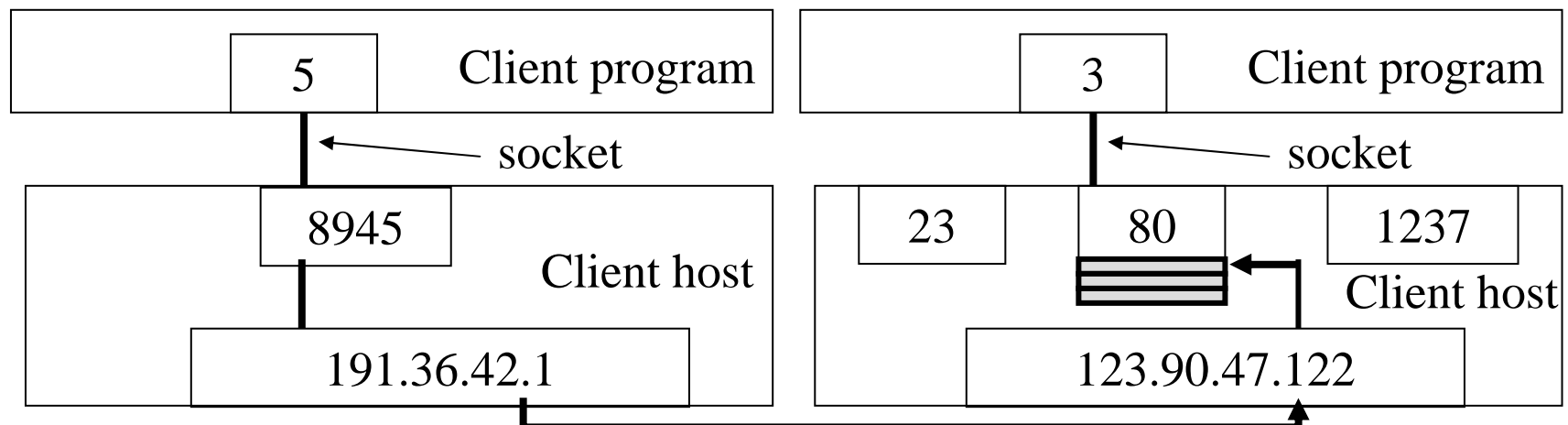
- Suppose we do:  
`result = listen( 2 );`  
and the result is not -1 (that is, it worked!)
- We now have the following situation:



## Step 4: Incoming connection request

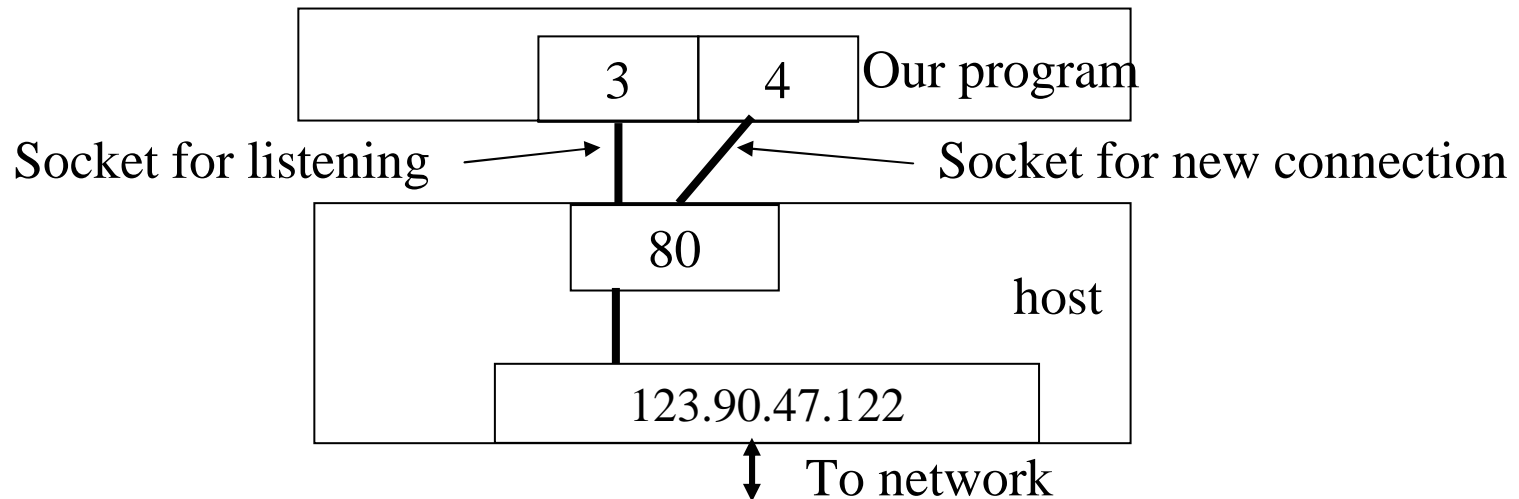
- Suppose we do:  
`name.sin_port = 80;`  
`name.sin_addr.s_addr = inet_addr("123.90.47.122" );`  
`result = connect( s, name, sizeof( name ) );`  
and the result is not -1 (that is, it worked!)

- We now have:

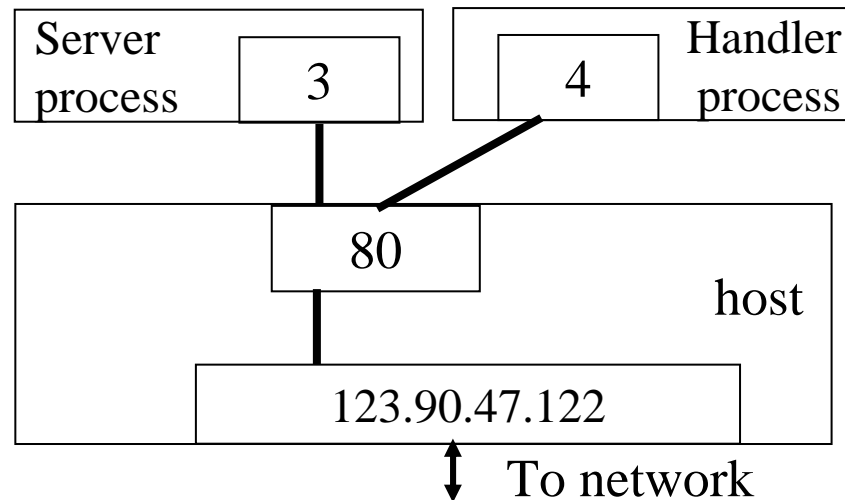


## Step 5: Accept a connection request

- Suppose we do:  
`newSocket = accept( s, addr, &addr_length );`  
and the result is a new socket with file descriptor 4.
- We now have the following situation:



## But, what we really want is...



- Why?
  - Server can handle multiple, simultaneous connection requests
  - A separate handler process can be created for each request

## Programming with sockets

- **Styles of communication:**
  - stream: reliable, two-way byte streams
  - datagram: unreliable, two-way record-oriented
  - etc.
- **Communication domains**
  - UNIX
    - endpoints (sockets) named with file-system pathnames
    - supports stream and datagram
  - Internet
    - endpoints named with IP addresses
    - supports stream and datagram
  - others
- **Protocols**
  - e.g., TCP/IP, UDP/IP

# Using Datagram Sockets (1)

- **Receiver steps**

- 1) **create socket**

- **socket system call**

```
int socket(int domain, int type, int protocol);  
fd = socket(PF_UNIX, SOCK_DGRAM, 0);
```

- 2) **set up receiver's name**

- **put name in *sockaddr\_un* structure**

```
struct sockaddr_un {  
    short sun_family; /* PF_UNIX */  
    char sun_path[108]; /* path name */  
} name;
```

```
name.sun_family = PF_UNIX;  
memcpy(name.sun_path, path, strlen(path));
```

## Using Datagram Sockets (2)

- **Receiver steps (continued)**
  - 3) bind receiver's name to socket**
    - **bind takes a generic *struct sockaddr* argument and is given the combined length of the first part of the structure and that portion of the second part that is used**

```
name_len = sizeof(name.sun_family) +  
           strlen(name.sun_path);  
bind(fd, (struct sockaddr *)&name, name_len);
```

## Using Datagram Sockets (3)

- **Receiver steps (continued)**
  - 4) receive (and send) data**
    - use *recvfrom* system call to obtain caller's address

```
int recvfrom(int s, char *buf, int len, int flags, struct  
    sockaddr *from, int *fromlen);
```

```
struct sockaddr_un sender_name;
```

```
int sender_len = sizeof(sender_name);
```

```
recvfrom(fd, buf, sizeof(buf), 0,
```

```
    (struct sockaddr *)&sender_name, &sender_len);
```

## Using Datagram Sockets (4)

- **Sender steps**
  - 1) **create socket**
  - 2) **set up sender's name (optional)**
  - 3) **bind sender's name to socket (optional)**

## Using Datagram Sockets (5)

- **Sender steps (continued)**

### 4) set up receiver's name

```
struct sockaddr_un recvr_name;  
int recvr_len;
```

```
recvr_name.sun_family = PF_UNIX;  
memcpy(recvr_name.sun_path, path, strlen(path));  
recvr_len = sizeof(recvr_name.sun_family) +  
strlen(recvr_name.sun_path);
```

## Using Datagram Sockets (6)

- **Sender steps (continued)**
  - 5) send (and receive) data**
    - use *sendto* system call to send datagram

```
int sendto(int s, const char *msg, int len, int flags, const  
struct sockaddr *to, int tolen);
```

```
sendto(fd, buf, sizeof(buf), 0,  
(const struct sockaddr *)&recvr_name, recvr_len);
```

# Using Stream Sockets (1)

- **Server steps**
  - 1) **create socket**
    - ***socket* system call**

```
int socket(int domain, int type, int protocol);  
fd = socket(PF_INET, SOCK_STREAM, 0);
```

## Using Stream Sockets (2)

- **Server steps (continued)**
  - 2) **set up server's address**
    - put “wildcard” internet address in *sockaddr\_in* structure
      - server may have multiple interfaces; we want to be able to receive on all of them
      - must convert from *host byte order* to *network byte order*

```
struct sockaddr_in {  
    short sin_family;  
    u_short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8]; /* padding */  
} my_addr;
```

```
my_addr.sin_family = PF_INET;  
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  
my_addr.sin_port = htonl(port);
```

## Using Stream Sockets (3)

- **Server steps (continued)**

- 3) bind server's name to socket**

```
bind(fd, (struct sockaddr *)&my_addr,  
      sizeof(my_addr));
```

- 4) set up socket to be in listening mode**

– *backlog* is the max length of the queue of waiting connections

```
int listen(int fd, int backlog);
```

## Using Stream Sockets (4)

- **Server steps (continued)**
  - 5) wait for a connection**
    - when connection is received, a new socket is created for communication on it
    - as an option, the address of the connector (client) is passed back

```
int accept(int fd, struct sockaddr *addr, int *addrlen);
```
  - 6) receive and send data**
    - simple *read* and *write* system calls can be used

## Using Stream Sockets (5)

- **Client steps**
  - 1) **create socket**
    - as an option, one can bind the socket to a particular port number
    - port numbers less than 1024 are reserved for privileged users

## Using Stream Sockets (6)

- **Client steps (continued)**
  - 2) **find internet address of the server**
    - look it up in *domain name service* (DNS)
    - each host may have a list of interfaces; we choose the **first one**

```
struct hostent *hostinfo;  
hostinfo = gethostbyname("botrytis.cs.brown.edu");  
memset(&server_addr, 0, sizeof(server_addr));  
server_addr.sin_family = PF_INET;  
memcpy(&server_addr.sin_addr,  
hostinfo->h_addr_list[0], hostinfo->h_length);  
server_addr.sin_port = htons(port);
```

## Using Stream Sockets (7)

- **Client steps (continued)**

- 3) connect to server**

- `connect(fd, (struct sockaddr *)&server_addr,  
sizeof(server_addr));`

- 4) send and receive data**

- **simple *write* and *read* system calls can be used**

## Exercise: Basic Socket Programming

- Objective: to write a client-server program that does simple file transfer across network.
- Two parts: client and server
  - The client requests the server send a specified file from the computer where the server resides. When the client receives the file, the file is saved on the local disk where the client is running.
- Hints:
  - The server listens to a specific port (non-reserved one, i.e., above 1024) that has been mutually agreed upon between the client and the server. When the server receives a request from a client, the request should contain the target file name with the complete path.
  - The server responds to the client in two steps:
    - First, the server sends back an acknowledgment indicating whether or not the file requested by the client can be transferred. If the requested file cannot be transferred (e.g. the file is not readable, the file does not exist), the client should print an error message and quit the connection. If the acknowledgment says the file is transferable
    - Second, the server actually sends the file to the client. The client will save the file to the local disk.
- Basic client-server example: [http://user.informatik.uni-goettingen.de/~fu/teaching/prog/socket\\_tcp/](http://user.informatik.uni-goettingen.de/~fu/teaching/prog/socket_tcp/), [socket\\_udp/](http://user.informatik.uni-goettingen.de/~fu/teaching/prog/socket_udp/)

## Understanding TCP/IP Configuration Files

- Configuration files
  - /etc/hostname
  - /etc/hosts
  - /etc/services
  - /etc/host.conf
  - /etc/nsswitch.conf
  - /etc/resolv.conf
- Exercise:
  - Find out the transport protocol and its port:
    - ftp, telnet, smtp, http, rsvp, rtp
  - Find out the dns server(s) used by your computer

## Familiar with TCP/IP Configuration Commands

- Configuration commands:
  - /sbin/ifconfig
  - /sbin/route
  - /sbin/netstat
- Exercise:
  - Find your network interfaces and the gateway of our university

## Familiar with TCP/IP Diagnosis Tools

- Diagnosis Tools:
  - ping
  - host, dnslookup
  - traceroute
  - Tcpdump
- Exercise:
  - Check your self-defined packet to [www.mit.edu](http://www.mit.edu), and explain why/where there is a significant change of the transmission delay.
  - Find the topology from your computer to [www.dfn.de](http://www.dfn.de)

## Exercise: Streaming File Transfer Using Sockets

- Objective: implementing a file server and “streaming media” client
  - You should let the file transfer complete quickly, and let the file played back concurrently, beginning as soon as a minimum amount of data has been received.
  - You can't simply use `xmms` <http://www.cs.uni-goettingen.de/~fu/tmp/song.mp3>, because `xmms` has no input buffer, hence it will have to stay connected to the server until it exits.
- Work individually