



Real-Time Programming TI00AA55

Lecture 10 - 01.04.2015

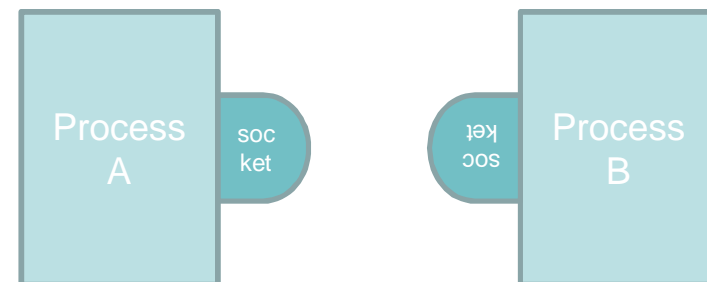
Jarkko.Vuori@metropolia.fi

Introduction 1

- Earlier we have learned **pipes and FIFOs**
 - These communication methods require that processes need to run **in the same computer**, and in the case of pipes processes must be relatives
 - One more restriction of pipes and FIFOs is that they are **unidirectional**
- **Sockets** can be used for communication between processes in the same computer or **between processes in different computers** (over the network)
 - Sockets provide **bidirectional** data transfer
- Now we concentrate on the Unix domain sockets which are used to communicate between processes in the same computer
 - The basic idea is the same for internet domain sockets
 - The only difference is in addresses
 - Unix domain sockets use file name as an address
 - Internet sockets use ip-addresses together with the port numbers
- Sockets are suitable for client server systems, because each client has their own communication channel to the server

Introduction 2

- We can use two different communication methods with sockets:
 - connection oriented communication and
 - connectionless communication
- We start by studying connection oriented communication
- You can think a socket as a wall socket through which you can send and receive information
- Sockets are represented by the file descriptors
- Before you can use sockets to send or receive data, you need to bind them with addresses
 - The operating system needs to know where the data you write to the socket is sent to (unix domain socket)
 - If processes are running in different computers, the operating system in the receiving computer needs to know, what descriptor in what process the data is sent to, when it comes in from the network interface
- The first step is to create the socket and the next step is to connect a socket to another socket in another process
- POSIX defines the socket interface
- A socket is created using the function `socket`
 - The function returns a file descriptor
- A more detailed description is on the next page



Creating a socket

- The function `socket` creates a socket

```
<sys/socket.h>
int socket(int domain, int type,
int protocol);
```

 - It returns a file/socket descriptor, if OK, and -1 if error
- The first parameter, `domain` determines the nature of the communication, including the format of the address
 - POSIX defines the following possible options for this parameter (AF comes from Address Family):
 - • `AF_INET` IPv4 Internet domain
 - `AF_INET6` IPv6 Internet domain
 - • `AF_UNIX` UNIX domain
 - `AF_UNSPEC` unspecified ("any" domain)
- The second parameter, `type` specifies further the way of communication
 - POSIX defines the following possible values for this parameter
 - • `SOCK_DGRAM` (Datagram)
 - • `SOCK_STREAM` (connection oriented byte stream)
 - `SOCK_RAW` (Datagram interface to IP)
 - `SOCK_SEQPACKET` (connection oriented messages)
- The third parameter (`protocol`) is most often 0
 - The system selects the default protocol according the first two parameters

How to use socket descriptor

- A file descriptor represents the socket
- Common system calls (that we have used with file descriptors, like `write`, `read` and `close`) can be used also for socket descriptors
- On the other hand, all file descriptor related system calls cannot be used for socket descriptors
 - Examples of these are `lseek` and `mmap`
- In addition to that, socket descriptors have some system calls of their own, as we will see later

- Example

```
int socket_fd;  
socket_fd = socket(AF_UNIX, SOCK_STREAM, 0);  
  
// Make a connection as we will see later  
  
write(socket_fd, &chr, 1); // write to the socket  
read(socket_fd, &chr, 1); // read from the socket  
close(socket_fd);
```

- As you can see from the example above, it is possible to write to the socket descriptor using system call `write` and read from the socket descriptor using system call `read`
 - Later you will see other functions with additional parameters and options
- The only missing part in the given example program is, how to make a connection to another socket in an another process

Connection oriented communication

- The socket for connection oriented communication is created putting `SOCK_STREAM` as the second parameter in socket function call as we saw on the previous page
- This communication type requires that we specifically create a “permanent” connection between two sockets
- After this we have a bidirectional connection between processes
- When we send data, there is no need anymore to specify the receiver (see the example on the previous page)
- This communication method could be compared to the telephone connection
 - After we have identified the receiver (by dialing the number), it is possible to send and receive information to/from the receiver
- Data consists of bytes (byte stream)
 - The read operation can return, before the required amount of bytes are read
- To create a connection to another socket in another process you need to know the address of that socket
- Addresses is the next topic

Socket addresses

- The bare wall socket is not useful without any connection
- The address of the socket is simply called socket address
- Socket addresses are specified using address structures
- The address is different in different domains (address families)
- The address is **filename**, when Unix domain sockets are used
- The address consists of the **ip-address and the port number** in the case of internet domain sockets
- **All system calls that handle addresses are generic**
 - They are used in a similar way and are not dependent on the address type
 - This is possible, because system calls use **generic address structure**
- Remark. IP-address specifies the computer, and port specifies the process and socket

Generic address structure

- The socket functions are kept independent of the different domains
- This goal is achieved by adopting the generic address structure `struct sockaddr`
 - All real address structures need to be converted to this type, before they are passed to the socket functions
- The generic address structure is defined in the following way (POSIX):

```
struct sockaddr {
    sa_family_t sa_family; // address family
    char        sa_data[]; // variable length
    address
    .
    .
    .
};
```

- In Linux it is defined as follows:

```
struct sockaddr {
    sa_family_t sa_family; //address family
    char        sa_data[14]; //variable length
    address
};
```
- The key thing here is that the first member in the structure (`sa_family` member) indicates the real address type (the layout of the rest of the structure)
 - This way the system calls can interpret it correctly

Unix domain address structure

- In the UNIX-domain (AF_UNIX) the type of the address structure is `struct sockaddr_un`
- This address structure type is casted to the generic address structure `struct sockaddr`, when it is passed as a parameter to the socket functions
 - (Actually the destination type of the pointer is casted)
- In Linux the definition of `struct sockaddr_un` is:

```
struct sockaddr_un {
    sa_family_t sun_family; //address family
    char sun_path[108];     // name in the
file system
};
```
- Example

```
struct sockaddr_un saddru;
saddru.sun_family = AF_UNIX;
strcpy(saddru.sun_path, "mysock.sock");
```
- Now the address structure is ready
 - The pointer to this structure is passed to the socket functions as the type `struct sockaddr*` (and `socklen_t`)
- For example, the prototype of `bind` function is

```
int bind(int fd, const struct
sockaddr* addr, socklen_t len);
```
- The unix domain socket address `saddru` from the given example could be passed to this function as follows:

```
bind(socket_fd, (const struct
sockaddr*)&saddru, sizeof(saddru));
```

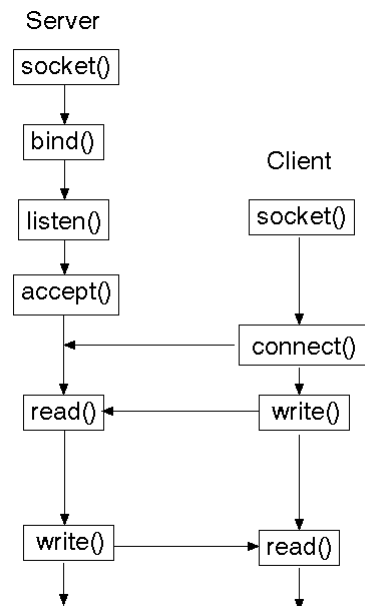
Creation of connection 1

- Now we know how to use socket addresses as such
- The “open question” still is, how to create a connection via sockets
- Next we learn how to create a connection oriented (SOCK_STREAM) connection between two processes
 - It is the most common connection type in client-server systems
- We will study how the connection is created between the server and the client
 - We will study what happens in the client end and in the server end from the programmers point of view
- The principle is the same whether the server and clients are running in the same computer or if they work over the network
- The creation of connection oriented connection consists of several steps in the client as well as in the server
- The client initiates the communication
 - That’s why it is important that the address of the server is made known for all possible clients
- Remark. It is also possible to implement client-server system using connectionless communication (SOCK_DGRAM)

Creation of connection 2

- The steps in the client:
 1. Create a socket (**socket**)
 2. Initialize the address structure that specifies the server
 3. Connect to the server (**connect**)
 4. When connect returns, the file descriptor provides a unique bidirectional connection to the server

- The steps in the server:
 1. Create a socket (**socket**)
 2. Initialize the address structure that specifies the server itself
 3. Bind the address to the socket descriptor (**bind**), so that operating system knows to “contact” this process and this socket when a client connects to this address
 4. Tell operating system that server is ready for connection requests (**listen**)
 5. Go to wait for connection requests (**accept**)
 6. Accept returns, when a client calls connect in another process. The return value from the accept function is a **new socket descriptor** that provides a unique bidirectional connection to the client



Server: Bind an address to the socket

- Let's start from the server, because server needs to be ready before clients can connect
- We have already handled phases 1 and 2 in the server. In the phase 3 bind function is used
- The prototype of bind:

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```
- The parameter `addr` has the following requirements:
 1. The address needs to be the address of the server (the address of the process itself where bind is called)
 2. The member `sa_family` in the address structure must be the same as what was the first parameter in the socket function when the socket was opened

Server: listen

- Before the client can contact the server, calling the connect function, the server needs to tell the kernel that it is ready to accept connection requests
- This is done calling the function listen:

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```
- Parameter `backlog` tells how many connection requests should be queued (in the kernel) in the case, when server is not capable to accept them
 - The constant `SOMAXCONN` specifies the maximum value for this
- If the queue is full, the kernel discards connection request after that

Server: accept

- When the server has created the socket (`socket`), bound it to the address (`bind`) and told the kernel that it is ready to accept connection requests (`listen`), it starts to wait for connection requests
- This is done with the function `accept`

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr
*addr, socklen_t len);
```
- On success, it return a nonnegative integer that is a descriptor for the accepted socket
 - On error, -1 is returned, and `errno` is set appropriately
- When the client connects, the `accept` function in the server returns
 - The `addr` parameter tells the server the address of the client
- This address is not usually needed in the server, because `accept` function returns a socket descriptor that is directly connected to the client
- For that reason, we often pass `NULL` as a second parameter and 0 as the third parameter for the `accept` function (if no need for the explicit address information of the client)
- Remark 1. By default, the `accept` function blocks, if there is no connection requests
 - It is possible, however, to put the socket descriptor in non-blocked mode
- Remark 2. Socket descriptor can also be added to the descriptor set of `select` function
 - Then the `select` function returns, when there already is a waiting connection request from the client (and `accept` is guaranteed to return)

Server: Several clients at the same time

- Servers can have many clients simultaneously and it needs to serve them all
- It is possible to use several techniques we have already learned to manage that problem:
 - fork the child process to serve a client
 - use multiplexed i/o or asynchronous i/o
 - (polling)
- If forking is used, the server usually has a loop that does the following things:
 - Wait for a connection request (accept)
 - When accept returns, fork a child (or create a thread) that serves the client. The child process inherits the descriptor that accept returned. It provides bidirectional connection between the child and the client (Parent does not use it)
 - Parent process returns back to wait for the next connection request (back to phase 1)
- Remark 1. In this model, functions socket, bind and listen are called outside the loop (before the loop) and accept inside the loop
- Remark 2. If the server uses datagrams for the communication, the functions listen and accept are not called
 - The connect function in the client and the sequence of listen and accept functions in the server belong to the connection establishment in connection oriented communication (SOCK_STREAM)

Client: Create a connection

- When the server has set up all things (bind, listen and accept), the client can connect to the server
- The client uses the function connect for this purpose. The prototype is as follows:

```
#include <sys/socket.h>
int connect(int sockfd, const struct
sockaddr *addr, socklen_t len);
```
- If the connection or binding succeeds, zero is returned
 - On error, -1 is returned, and errno is set appropriately
- The address structure `addr` (the second parameter) determines the address of the server the client wants to connect to
- If the client socket is not yet bound to any address, it is done automatically now (internet sockets only)
- Calling connect in the client causes accept in the server to return
- The connection attempt can fail for many reasons:
 1. The server process is not running
 2. The address is not bound to the server socket
 3. The connect queue is full in the server
- Remark 1. It is possible to set flag `O_NONBLOCK` in the file flags of the socket descriptor
 - In that case connect returns immediately even the connection attempt did not succeed
- Remark 2. The connect function can also be used with the datagram socket. Then all packets go to address given as a parameter to the connect function
 - Address parameter of `sendto` is discarded

Transferring data 1

- When all the things described on the previous pages are done in the server and in the client, there is a connection between the client and the server
 - Data can be transferred to both directions between the client socket and the server socket (returned by the accept function)
- Because we now have a kind of permanent connection between sockets, we can use ordinary simple system calls **write** and **read** to send and receive data
- The data that is written to the socket in the client can be read in the server from the socket that was returned by the accept, when client connected
- In similar way, the data that the server writes to the socket that was returned by the accept can be read in the client from it's own socket
- Instead of these ordinary system calls, it is possible to use functions `send` and `recv`. Their prototypes are:

```
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```
- The difference between these functions compared to `write` and `read` is, that there is an additional parameter `flags` for further options
- Some possible flags are listed on the following page

Transferring data 2

The function `send` can take the following flags

MSG_DONTWAIT	Don't block (O_NONBLOCK)
MSG_NOSIGNAL	Do not send SIGPIPE
MSG_OOB	Send out-of-band data

Function `recv` can take the following flags

MSG_OOB	Receive out-of-band data
MSG_PEEK	Return packet contents without consuming
MSG_WAITALL	Wait until all data is available (SOCK_STREAM only)

How to find out the address

- In connection oriented communication it is possible to find out the addresses of both ends of the connection
- It is possible to find out the address that is bound to the socket. This can be done with the function **getsockname**

```
#include <sys/socket.h>
int getsockname(int sockfd, struct
sockaddr *addr, socklen_t *len);
```
- When the function is called, the variable, where `len` is pointing to, indicates the size of the address buffer
- When function has returned, this variable indicates the size of the real address that was produced by the function
- If the buffer is too small, the address is cut off
- If the socket is connected to another socket, you can ask the address of the other end of the connection with the function **getpeername**

```
#include <sys/socket.h>
int getpeername(int sockfd, struct
sockaddr *addr, socklen_t *len);
```
- Remark 1. If Unix domain socket are used, the client has no explicit address, if it is not bound some filename to itself
- Remark 2. If internet domain sockets are used, connect automatically generates and binds an address to the client

Socket pair

- We have discussed about the socket communication that is suitable for client server systems having many clients that can be independent of the server
- If you need only simple connection oriented communication channel between two relative processes (between parent and child, for example) it is easy to create with the function `socketpair`
 - For example, if you need socket connection between the parent and the child, just call `socketpair` in the parent before the fork
 - This function creates a socket connection and gives the socket descriptors to both ends of the connection
 - The basic idea is very similar to the pipe but now the connection is bidirectional
- See the example `socketPair.c` in the web
- Connection oriented socket connection is very similar to the pipe (and fifo) connection in many sense:
 1. Eof is received if empty socket is tried to read and there is no open descriptor to the other end
 2. Signal SIGPIPE is sent if write is done and there is no open descriptor to the other end
 - Write returns after that with an error EPIPE

Connectionless communication

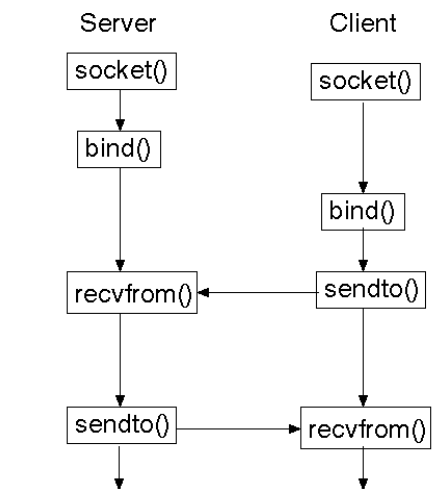
- In **SOCK_DGRAM** communication method there is no need to create any connection
- This method is called connectionless communication
- Communication can be pier-to-pier communication
 - The process can send a data package to the socket passing the destination address as a parameter
- The system creates a datagram that contains the destination address, the source address and the data itself
 - This communication method corresponds to the sending of a letter by post
- If sending succeeds, it does not guarantee that the message is delivered to the destination
- If you send several datagrams there is no guarantee that they are delivered in the same order they were sent
- To be capable to receive datagrams, the program needs to bind the address of it's own, so that operating system is capable to deliver the arriving packets to the correct process and to correct descriptor
- The receiving process can also get the address of the sender from the datagram

Connectionless communication

- In connectionless communication (SOCK_DGRAM) you need a special function `sendto`, so that you can specify the receiver address as a parameter
- There is a counter-part for this function that has a parameter that can be used to find out the sender. This function is `recvfrom`
- Remark 1. If function `sendto` is used to connected (connection oriented) socket, the parameter `destaddr` is discarded
- Remark 2. It is possible to pass NULL as an address parameter `addr` and 0 as address length to the function `recvfrom`, if we are not interested who is the sender

- The prototypes of these functions are as follows:

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags, const struct sockaddr *destaddr, socklen_t destlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *addr, socklen_t *addrlen);
```
- The function `recvfrom` returns, when the whole message has been received



Connection oriented communication

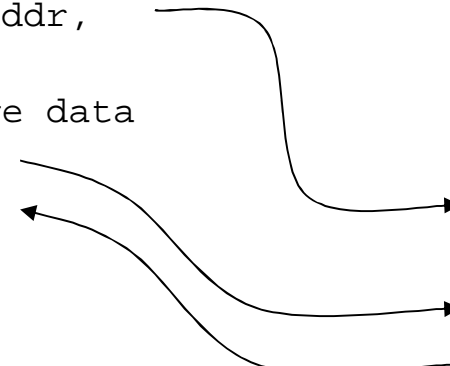
Client

```
int fd;
struct sockaddr_un srv_addr;
char chr;
//Create socket
fd = socket(x,x,x);
//Initialize the address structure
//srv_addr that specifies the server

//Connect to the server
connect(fd, &srv_addr,
sizeof(srv_addr));
// Send and receive data
write(fd, "A", 1);
read(fd, &chr, 1);
```

Server

```
int srv_socket, client_socket;
struct sockaddr_un srv_addr;
char chr;
// Create socket
srv_socket = socket(x,x,x);
// Initialize the address structure
// srv_addr that specifies the server
// Bind address to this process
bind(srv_socket,
      (struct sockaddr*)&srv_addr,
      sizeof(srv_addr));
// Start listening
listen(srv_socket, 5);
// Accept connection requests
client_socket = accept(srv_socket, NULL, 0)
// Send and receive data
read(client_socket, &chr, 1);
write(client_socket, "B", 1);
```



Connectionless communication

Client (sender)

```
int sockfd;
struct sockaddr_un srv_addr, clnt_addr;
char chr;
//Create socket
sockfd = socket(x,x,x);
// If you want that the receiver gets the sender
// address, initialize the address structure
clnt_addr that spesifies the client and bind it.
bind(sockfd, (struct sockaddr*)&clnt_addr,
      sizeof(clnt_addr));
// Initialize the address structure srv_addr that
// spesifies the server (receiver)
// Send data
sendto(sockfd, "A", 1, 0,
        (struct sockaddr*)&srv_addr, size));
// Receive data
recvfrom(sockfd, &chr, 1, 0, NULL, NULL);
```

Server (reader)

```
int sockfd, size;
struct sockaddr_un srv_addr, client_addr;
char chr;
// Create socket
sockfd = socket(x,x,x);
// Initialize the address structure srv_addr
that
// spesifies the server
// Bind address to this process
bind(sockfd, (struct sockaddr*)&srv_addr,
      sizeof(srv_addr));
// Receive data
size = sizeof(client_addr);
recvfrom(sockfd, &chr, 1, 0,
          (struct sockaddr*)&client_addr, &size));
// Send data
sendto(sockfd, "B", 1, 0,
        (struct sockaddr*)&client_addr, size));
```

When recvfrom has returned, client_addr tells address of sender

Remark. Different file names are needed in server and client.