Systems programming

Week 2 – Lab 4

Client-Server with datagram sockets

Sockets are an evolution of FIFOS that could only send data in one direction and had a few other issues. To allow sending of data int two direction (full duplex communication) we have sockets that can work in the UNIX domain (for processes in the same machine or the Internet domain.

The way to establish communication using sockets or FIFOS is a bit different.

With FIFOS there is a single entity (the FIFO that has a name) and whose programs open to transfer data. FIFOS are created and the opened.

For two processes to communicate with sockets it is necessary to create one socket on each program: each communication participant should create a socket. Sockets are not user to transfer data (as a FIFO) but to send and receive. If the program need to receive data from a socket it is necessary to assign it a address (in the case of UNIX domain sockets this address will be file name). When sending the message it is necessary to specify what socket is going to receive the message.

1 Datagram sockets (simplex-communication)

The example-1 directory contain a two programs (a client and a server) the exchange messages in a simplex-communication pattern: the client sends messages, the server receives messages.

It is necessary to include the following header files:

```
#include <sys/socket.h>
#include <sys/un.h>
```

The next table shows the main parts of the code with a simple explanation of the code:

Socket creation

Each program create his own socket. The domain ad type should be corrected taking into consideration the scope and type of communication

Server

```
int sock_fd;
sock_fd = socket(
     AF_UNIX, SOCK_DGRAM, 0);
if (sock_fd == -1){
     perror("socket: ");
     exit(-1);
}
```

Client

```
int sock_fd;
sock_fd = socket(
       AF_UNIX, SOCK_DGRAM, 0);
if (sock_fd == -1){
     perror("socket: ");
     exit(-1);
}
```

Socket identification

In this example only the server need to assign (bind) one address, because only the server will receive messages. Since this address corresponds to a file name it is necessary to remove it with the **`unlink`** command (in case it exists) and verify if the address was correctly assigned/binded.

Server

```
struct sockaddr_un local_addr;
local_addr.sun_family = AF_UNIX;
```

```
strcpy(local_addr.sun_path, SOCK_ADDRESS);

unlink(SOCK_ADDRESS);
int err = bind(sock_fd,
              &local_addr, sizeof(local_addr));
if(err == -1) {
     perror("bind");
     exit(-1);
}
```

Reception of the message

The server needs to call the **recv** function to receive messages sent from the other processes

Server

```
nbytes = recv(sock_fd, buffer, 10, 0);
printf("received %d bytes:\n",nbytes);
```

Sending of messages

In datagram sockets, every time a program sends a message it is necessary to specify what is the address of the recipient. The **sendto** function receives the data (similarly to the write function), but has two extra arguments that refer to the address of the recipient.

Client

```
struct sockaddr_un server_addr;
server_addr.sun_family = AF_UNIX;
strcpy(server_addr.sun_path, SOCK_ADDRESS);
nbytes = sendto(sock_fd,
                message, bytes_message, 0,
            &server_addr, sizeof(server_addr));
```

Since only the server has an address assigned to the socket only it is impossible for the client to receive any response from the server. Nonetheless this configuration replaces FIFOs.

1.1 Exercise 1

Observe the files in the **example-simplex** directory, compile both program and execute various combinations of them in different windows:

- just client-simplex (without running server-simplex)
- server-simplex and afterwards client-simplex
- one server-simplex and multiple client-simplex

Every time you run one of the previous options experiment with various message lens (in the client-simplex).

After running multiple times the various options answer to the following questions

- what happens if the client-simplex sends a message without a receiving server-simplex?
- What happens if the sent message is larger that the reception buffer (10 bytes)
- What happens if the sent message is smaller that the reception buffer (10 bytes)
- What happens if the client-simplex is killed (with Ctr-C) and restarted afterwards?
- What happens if multiple clients send data at the same time?

Compare FIFOs with datagram sockets.

2 Exercise 2 - Client-server character remote control

In this exercise students will replace the FIFOS used in the previous laboratory with datagram sockets.

Students should modify the result of Exercise 3 (aserver and the two clients) from Lab3 to start to use Unix domain datagram sockets.

Te following instruction specify the places were code needs to be changed (with respect to Exercise 1. The steps performed in Exercise 3 do not need to be changed.

2.1 TODO 1

In this exercise students can use the same message that was defined in Lab2-3.1

2.2 TODO 2

Define the name/address of the server socket. It should also be a name that refers to a non existent file in /tmp/

2.3 TODO 3

The server should create a UNIX/datagram socket and bind to it the address defined in TODO 2.

2.4 TODO 4

Create a UNIX/datagram socked in the human-control-client. This socket does not need an address because will not receive any message from the server.

2.5 TODO 5

Does not need to be changed

2.6 TODO 6

To send a message to the server the client can use the same structure, but the function to be called is different. With sockets it is necessary to call the **sendto** function. This function receives the socket variable, the data and the address of the server.

2.7 TODO 7

The server will need to call the **recv** function to read a message. It receives as argument the socket previously created and the buffer where the message will be put

2.8 TODO 8

Does not need to be changed

2.9 TODO 9

Does not need to be changed

2.10 TODO 10

Use the **sendto** function to send a message to the server

2.11 TODO 11

Does not need to be changed

3 Duplex communication

In order to implement duplex communication it is necessary that all participant program bind/assign one address to their sockets. After this, it is possible for the message recipient retrieves the address of the sender (using the **recvfrom** function) and reply (with the **sendto** function).

The **example-duplex** directory contains two programs (a client and a server) that exchange messages in a duplex-communication pattern: the client sends messages, the server replies to the client.

The next table shows the main parts of the code with a simple explanations

Socket creation

Each program create his own socket. The domain ad type should be corrected taking into consideration the scope and type of communication

Server

```
int sock_fd;
sock_fd = socket(
     AF_UNIX, SOCK_DGRAM, 0);
if (sock_fd == -1){
     perror("socket: ");
     exit(-1);
}
```

Client

```
int sock_fd;
sock_fd = socket(
       AF_UNIX, SOCK_DGRAM, 0);
if (sock_fd == -1){
     perror("socket: ");
     exit(-1);
}
```

Socket identification

In this example only the server assigns (bind) one well know address to its socket. This address should be know by every client that whats to communicate with the server.

Server

```
struct sockaddr_un local_addr;
local_addr.sun_family = AF_UNIX;
strcpy(local_addr.sun_path, SOCK_ADDRESS);

unlink(SOCK_ADDRESS);
int err = bind(sock_fd,
              &local_addr, sizeof(local_addr));
if(err == -1) {
     perror("bind");
     exit(-1);
}
```

Socket identification

The client also need to assign (bind) one address to his socket, but must be different from every other client address..

To accomplish this distinction we use the **getpid** function that returns the process id.

Client

```
struct sockaddr_un local_client_addr;
local_client_addr.sun_family = AF_UNIX;
sprintf(local_client_addr.sun_path,
        "%s_%d",
        SOCK_ADDRESS, getpid());

unlink(local_client_addr.sun_path);
int err = bind(sock_fd,
               &local_client_addr,
               sizeof(local_client_addr));
if(err == -1) {
   perror("bind");
   exit(-1);
}
```

Reception of the message

Since the server want to send a reply to the client it needs to retrieve the client address This is accomplished with the **recvfrom** function. This function receives the message, but also the client address.

Server

```
struct sockaddr_un client_addr;
socklen_t client_addr_size =
             sizeof(struct sockaddr_un);
nbytes = recvfrom(sock_fd, buffer, 100, 0,
               &client_addr, &client_addr_size);
```

Sending of messages

In datagram sockets, every time a program sends a message it is necessary to specify what is the address of the recipient. The **sendto** function receives the data (similarly to the write function), but has two extra arguments that refer to the address of the recipient.

Client

```
struct sockaddr_un server_addr;
server_addr.sun_family = AF_UNIX;
strcpy(server_addr.sun_path, SOCK_ADDRESS);
nbytes = sendto(sock_fd,
                message, bytes_message, 0,
            &server_addr, sizeof(server_addr));
```

Server

```
nbytes = sendto(sock_fd,
     reply_message, strlen(reply_message)+1, 0,
     &client_addr, client_addr_size);
```

3.1 Exercise 3

Observe the files in the **example-duplex** directory, compile both program and execute various combinations of them in different windows:

- just client-duplex (without running server-duplex)
- server-duplex and afterwards client-duplex
- one server-duplex and multiple client-simplex
- server-duplex and one client-simplex
- server-simplex with a client-duplex

Experiment sending messages of various lengths and killing the client /server at various stages of execution.

After running multiple times the various options, answer to the following questions

- What happens if the client-simplex tries to contact the server-duplex?
- What happens if the client-duplex tries to contact the server-simplex?

4 Exercise 4 - Remote control with notification

Modify the exercise 2 so that every time one character goes over another one (two characters at the same place. A message is sent back to the client that originated the movement.

The client flashes the screen (**flash()**) when his character is on tom of some other.

Follow the next steps to accomplish duplex-communication:

4.1 Step 1

Modify the message so that it can be used in both ways. The simplest modification is to add two new message types.

4.2 Step 2 (client)

Define the format for the address of the client.

The address of each client can be the concatenation of the server address with the process id (**getpid()**)

4.3 Step3 (client)

Assign the client address to the client socket.

After creating the socket it is necessary to do the **bind**. This will assign a unique address and will allow the server to send messages back to the client.

4.4 Step 4 (server)

Replace the **recv** function with the **recvfrom** function.

The **recvfrom** function will receive as arguments pointers to an address and its size, and update them when receiving a message.

The address stored by the **recvfrom** will be used later to send a reply to the client.

4.5 Step 5 (server)

Verify if the new character position is already occupied by other character. This is accomplished by looping over the array that stores the various characters.

4.6 step 6 (server)

After moving the character and verifying if there is any collision, send a message back to the client. Use the **sendto** message, where the recipient address is the one updated by the **recvfrom**.

This message should be sent even if there is no collision.

4.7 Step 6 (client)

Put a **recv** after the send (TODO 10) and verify the type of message received.

If there is a collision call the **ncurses flash()** function.