CS4375 Operating Systems Concepts
Fall 2024. Homework Assignment 2
Due: 11/03/2024 2:00AM MST
Team Assignment

This second assignment covers networking aspects with TCP/IP and UDP/IP in the POSIX world, as expressed in C. The assignment is to be done in teams of two students, together, step by step.

Proceed in the order indicated below. Proceeding in another order or proceeding in parallel (you and your team partner) will make the assignment harder.

For this assignment, you have to turn in an archive containing:

- the source file `send_udp.c`,
- the source file `receive_udp.c`,
- the source file `reply_udp.c`,
- the source file `send_receive_udp.c`,
- the source file `tunnel_udp_over_tcp_client.c`,
- the source file `tunnel_udp_over_tcp_server.c` and
- your report (as a PDF) covering all tasks in all sections of this assignment.

The Linux manual pages, which you can open using the shell command e.g. `man 2 socket`, are your friend. Read and understand them in their entirety for all system calls that are used in the programs to be written in this assignment.

You can only use level-2 calls to read from and write to file descriptors, including standard input and standard output. As a special exception, you can use `fprintf` on `stderr`, possibly together with a call to `strerror` for error messaging.

In order to perform proper testing, you should use `dandelion` together with your own computers to send TCP and UDP through the UTEP network. In order to access ports on dandelion opened by your own programs, you must be at UTEP, using a UTEP-provided IP address. Access to `dandelion` over the Internet is not possible for this purpose due to firewall restrictions.

For this homework assignment, no boilerplate code is provided by your instructor. You are free to organize your code in any way you please. However, as the programs have a certain amount of code complexity, you need to work on designing your software in a reasonable manner. It is very easy to end up with spaghetti for this assignment. No! Use functions. Comment your code.

1 The `send_udp` Program

Write a program `send_udp` that takes a server name and a port name in argument (verifying it got at least these arguments), opens a UDP socket to the server at the port indicated and then keeps on reading standard input, until standard input reaches the End-Of-File condition. Each chunk of bytes read off standard input is sent in a UDP packet. Use `getaddrinfo` function with hints configured as seen below to get the correct UDP socket file descriptor.

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_protocol = 0;
hints.ai_flags = 0;
gai_code = getaddrinfo(server_name, port_name, &hints, &result);
```

Use the regular `read` system call on standard input, reading into a buffer of exactly 480 bytes. Then send off the number of bytes read in this chunk (this number may be less than 480) in UDP packets using the `send` system call. When reaching the End-Of-File condition on standard input, your program must send a UDP packet of size 0 to the other UDP peer (this is possible as per POSIX rules); it must then terminate.

Do not forget to close all file descriptors you opened. Do not use any calls to `malloc`, `calloc` etc. but ensure that you properly free the linked list allocated by `getaddrinfo` with `freeaddrinfo`.

Make sure your program fails properly if an error condition arises, i.e. with a clear error message.

2 The `receive_udp` Program

Write a program `receive_udp` that takes a port name in argument (verifying it got at least this argument), converts the port name to a 16bit unsigned integer using the function given below, opens a UDP socket to receive on that port, then receives UDP messages on the port, writing their contents out on standard output, until it receives an empty UDP packet (UDP packet of size 0), at which moment it terminates.

```
static int convert_port_name(uint16_t *port, const char *port_name) {
  char *end;
  long long int nn;
  uint16_t t;
  long long int tt;

  if (port_name == NULL) return -1;
  if (*port_name == '\0') return -1;
  nn = strtoll(port_name, &end, 0);
  if (*end != '\0') return -1;
  if (nn < ((long long int) 0)) return -1;
  t = (uint16_t) nn;
  tt = (long long int) t;
  if (tt != nn) return -1;
  *port = t;
  return 0;
}
```

Use the `socket` and `bind` system calls to create a UDP socket (file descriptor) at the required port. Use `recv` system call to receive UDP packets. Use the `write` system call, wrapped as usual is some `better_write` function, to write to standard output.

Do not forget to close all file descriptors you opened. Do not use any calls to `malloc`, `calloc` etc. nor any other kind of dynamically allocated datastructure. Use a buffer with a size of exactly 2^{16} bytes as the buffer for `recv`.

Make sure your program fails properly if an error condition arises, i.e. with a clear error message.

Use your `send_udp` and your `receive_udp` programs to transmit a larger file (such as the nanpa file from the last assignment) over an IP link. Showcase the fact that a UDP transmission is not reliable, in the sens that the output of `receive_udp` may be incomplete (if packets were lost on the way) or permutated (if packets got reordered). The result of your experiments will go into your report.

3 The `reply_udp` Program

Using the experience gained with the two programs `send_udp` and `receive_udp`, write a program `reply_udp` that takes a port number as a single argument (and verifies that that argument is there). It converts the port number to a 16bit unsigned integer and opens a UDP socket at that port, just like `receive_udp` does. In a loop, it then receives

UDP packets using the `recvfrom` system call, keeping the sender's address and port in a variable, and immediately sends the UDP packet back to the sender, using `sendto` system call. The `reply_udp` program does not terminate when receiving an empty UDP packet (a message of size 0) but sends that empty UDP packet back.

Do not forget to close all file descriptors you opened. Do not use any calls to `malloc`, `calloc` etc. nor any other kind of dynamically allocated datastructure. Use a buffer with a size of exactly 2^{16} bytes as the buffer for `recvfrom` and `sendto`.

Make sure your program fails properly if an error condition arises, i.e. with a clear error message.

You will be able to test your `reply_udp` only when you finish the program `send_receive_udp`, required for the next Section.

4 The `send_receive_udp` Program

Now write a `send_receive_udp` program that does the following:

- It receives a server name and a port name in arguments and verifies that these arguments are indeed given.
- It opens a UDP socket to send UDP packets to the server at the port indicated. It uses `getaddrinfo` for this purpose, just like `send_udp`.
- It then keeps on reading chunks of data of up to 480 bytes on standard input –whenever reading on standard input does not block– and sends them as UDP packets to the specified server at the port, just like `send_udp`. **At the same time**, it receives UDP packets on the UDP socket it opened –whenever `recv` does not block. It uses a buffer of size 2^{16} for this reception. It writes the contents of these packets to standard output, using `write` resp. `better_write`. In order to figure out which of the two system calls (`read` on standard input or `recv` on the UDP socket file descriptor) does not block, it uses `select`.
- The program terminates when standard input reaches the End-Of-File condition or when it receives an empty UDP packet.

In the case you do not wish to use `select`, you can use threads. A solution with `select` is simpler, though.

Do not forget to close all file descriptors you opened. Do not use any calls to `malloc`, `calloc` etc. but ensure that you properly free the linked list allocated by `getaddrinfo` with `freeaddrinfo`. Use a buffer with a size of exactly 480 bytes for the `read` on standard input and of exactly 2^{16} bytes for `recv`.

Make sure your program fails properly if an error condition arises, i.e. with a clear error message.

You can now test your `send_receive_udp` program together with `reply_udp`. Observe that there may be reorderings and lost packets, as everything is done over UDP, which is an unreliable, connectionless IP transmission protocol.

5 The `tunnel_udp_over_tcp_client` Program

It can be observed that UDP is sufficiently reliable when used locally, i.e. in networks inside one home or one institution. UDP is perfectly reliable when a computer talks to itself, typically using `localhost`, bound to the local network interface. UDP is unreliable for Wide Area Networks, i.e. the open internet. It has its advantages though. For instance, UDP is inherently packet-oriented. A packet arrives as a whole packet if it does arrive.

TCP is a reliable stream protocol. This means the Operating Systems running on both ends of a TCP connection will ensure every byte written to the TCP file descriptor on one side will eventually come out in the same order on the other side, where it can be read from the corresponding file descriptor. TCP is not packet oriented, but byte oriented. This means, calls to `read` on a TCP file descriptor will end up spitting out every byte of a message, but the message (which may be a single "packet") may be spread over the bytes returned by several `read` calls.

In this Section, you will write a `tunnel_udp_over_tcp_client` program that will allow UDP packets, received by the program, to be forwarded ("tunnelled") over TCP, where they will be sent further using UDP. The `tunnel_udp_over_tcp_client` program will need to use an instance of the `tunnel_udp_over_tcp_server` program, which you will write in the next Section below, to function.

The `tunnel_udp_over_tcp_client` program does the following:

- The program takes three arguments:
  - A UDP port. It converts that port indication to a 16bit unsigned integer.
  - A TCP server name.
  - A TCP port name.

  It verifies it got all these arguments.
- It opens a UDP socket at the indicated UDP port. In that respect, it behaves like `reply_udp`.
- It opens a TCP connection to the indicated TCP server name and port, using `getaddrinfo` with the hints `hints.ai_family = AF_INET;` and `hints.ai_socktype = SOCK_STREAM;`.
- It then uses `select()` to read (resp. receive) on one or the other of the 2 file descriptors (the one for the UDP port and the other for the TCP connection). It does the following with the received data:
  - When receiving a UDP packet into a buffer of size 2^{16} with `recvfrom`, it stores the UDP packet's sender information into a variable and marks that information to be available. It then creates a message of a size corresponding to the received UDP packet's size but maximally $2^{16} + 2$ bytes, formatted as follows: 2 bytes forming a 16bit unsigned integer **in network byte order** (big endian) indicating the length of the received UDP packet, followed by the bytes of the received UDP packet. In the special case of an empty UDP packet, the 2 bytes of length indication are still (part of) the message, even though they are set to zero, followed by no bytes of content; this means, in this case a 2byte message is created. The program then uses `write`, properly wrapped into `better_write`, to send the message out over the TCP connection.
  - When receiving bytes over the TCP connection (with `read`, using a buffer of size $2^{16} + 2$ bytes), the program analyzes these bytes, accumulating them into a reconstruction buffer of size $2^{17} + 4$ bytes. The bytes coming in over the TCP connection form messages in the format described above: 2 bytes of "header" forming a 16bit length indication in network byte order, followed by 0 to $2^{16} - 1$ bytes of content. At reception of a byte, the program needs to decide if these bytes are part of a header (length indication) or if they are part of the content. As soon as the 2 bytes of header are received, the program figures out how many more bytes it needs to receive until the message is complete and it can again switch to receiving header information (for the next message). All that logic can be implemented easily by copying bytes one-by-one from the TCP reception buffer of size $2^{16} + 2$ into the reconstruction buffer of size $2^{17} + 4$, maintaining a rolling index into the reconstruction buffer that gets reset to zero whenever a message is complete. The indexing logic ensures that the 2 bytes of header always land on the first 2 bytes of the reconstruction buffer. As soon as a complete message is received, the program sends it out (without its header) using the UDP file descriptor and the `sendto` system call. For `sendto`, the program needs to provide a destination address, for which it uses the one recovered with `recvfrom` at UDP packet reception. If a TCP-received message needs to be sent out with UDP before any UDP packet was ever received by the program, the program discards the TCP-received message.
- The program terminates regularly on the End-Of-File condition on the TCP connection file descriptor. The program does not terminate when it receives an empty UDP message (see above for details). The program terminates with an error condition when an error arises.

Do not forget to close all file descriptors you opened. Do not use any calls to `malloc`, `calloc` etc. nor any other kind of dynamically allocated datastructure, besides what is required for `getaddrinfo`. Deallocate the linked list returned by `getaddrinfo` with `freeaddrinfo`. Use buffers of the sizes that have been indicated above.

Make sure your program fails properly if an error condition arises, i.e. with a clear error message.

Ensure that your program can properly decode TCP-received messages, even if the content of this TCP-received messages are spread over the data chunks received with several, consecutive calls to `read`.

You will be able to test your `tunnel_udp_over_tcp_client` only when you finish the program `tunnel_udp_over_tcp_server`, required for the next Section.

6 The `tunnel_udp_over_tcp_server` Program

Now write the `tunnel_udp_over_tcp_server` program, which is very similar to the `tunnel_udp_over_tcp_client` program, and does the following:

- The program takes three arguments:
  - A TCP port. It converts that port indication to a 16bit unsigned integer.
  - A UDP server name.
  - A UDP port name.

  It verifies it got all these arguments.
- It opens a TCP listen socket at the indicated TCP port.
- It opens a UDP socket to the indicated UDP server name and port, using `getaddrinfo`. In that respect, it behaves like `send_receive_udp`.
- It listens on the TCP listen socket using the `listen` system call. When a connection comes in, it accepts that connection using `accept`. This accepted connection is identified with a TCP-bound file descriptor.
- It then uses `select()` to read (resp. receive) on one or the other of the 2 file descriptors (the one for the UDP port and the other for the TCP connection) and to forward the data in both directions, just like `tunnel_udp_over_tcp_client`, with the following differences:
  - UDP packets are received with `recv` instead of `recvfrom`. The sender's address information of the received UDP packets is not recorded, as it is always known for this program.
  - As soon as they are complete, TCP-received messages can always be sent out over UDP with `send` (instead of `sendto`), as the UDP destination address is always known.

Do not forget to close all file descriptors you opened. Do not use any calls to `malloc`, `calloc` etc. nor any other kind of dynamically allocated datastructure, besides what is required for `getaddrinfo`. Deallocate the linked list returned by `getaddrinfo` with `freeaddrinfo`. Use buffers of the sizes that have been indicated above. Due to the fact that you needed to create a listening TCP socket, you have one file descriptor more to close than you had in `tunnel_udp_over_tcp_client`.

Make sure your program fails properly if an error condition arises, i.e. with a clear error message.

Ensure that your program can properly decode TCP-received messages, even if the content of this TCP-received messages are spread over the returned data chunks received with several, consecutive calls to `read`.

Now create a whole transmission chain to test your `tunnel_udp_over_tcp_client` and `tunnel_udp_over_tcp_server` programs. In that chain, `send_receive_udp` sends data read from standard input over UDP, the UDP gets tunneled with `tunnel_udp_over_tcp_client` over TCP and sent to a `tunnel_udp_over_tcp_server` instance, which spits out UDP again, which gets sent to `reply_udp`, that sends everything back the whole chain until `send_receive_udp` prints it out again.

7 Testing and Report Writing

Test all your programs thoroughly. Answer all questions and respond to all experimental tasks described above.

Again, eventually you need to create a whole transmission chain: `send_receive_udp` sends data read from standard input over UDP, the UDP gets tunneled with `tunnel_udp_over_tcp_client` over TCP and sent to a `tunnel_udp_over_tcp_server` instance, which spits out UDP again, which gets sent to `reply_udp`, that sends everything back the whole chain until `send_receive_udp` prints it out again.

Write a report with all answers, experimental data and additional remarks. If you have access to a machine where you can run `tcpdump`, you may want to enhance your report with explaining dumps of the different UDP and TCP packets that are sent over the tool chain `send_receive_udp` ↔ `tunnel_udp_over_tcp_client` ↔ `tunnel_udp_over_tcp_server` ↔ `reply_udp`.