– Lab3 –
Reliable transportation protocol on top of UDP

1 General Instructions

You can solve this lab individually or together with one other student. If you work in a group, both students must participate actively in the implementation and demonstration of the solutions.

Be sure to be well prepared prior to the scheduled lab and use the time to ask questions and to demonstrate your solutions.

Deliverables

- Oral demonstration of your solutions for the lab assistant. Be sure that you understand what you have done – it is not sufficient with a working solution.
- Submit your answers to Blackboard. In case two students work together, both must submit their answers to Blackboard and you must write both names on the answers.
- All program codes must be well documented (undocumented code will not be approved). As a minimum, you should include program name, author(s), date, **how to execute the program**, and a general description of the solution. Also, you should write comments on important sections in the code.
  Use your comments to answer the question `why` and now `how` (the code itself often explains how something is done)
- Make sure that you check the return values on all system/library calls (e.g., malloc, socket, etc.) that you use.
- Make sure that the output format at the client and server sides have **at least timestamps** of sent and received packets.
- The lab contains two parts. Part 1 must be demonstrated no later than April 25/27, 2017, part 2 - May 23, 2017. There will **not** be any extra occasions for you to demonstrate your solution after this deadline.

2 Introduction

The goal of this assignment is to create a reliable and efficient transport protocol on top of UDP. The UDP protocol is by default a connectionless protocol, i.e., it does not provide connection setup, error control, or any other TCP-like functions.

The task is to create a protocol stack on top of UDP which provide:

- Connection setup and teardown.
- A sliding window mechanism for efficiency.
- An error check mechanism.

Both a client and a server should be implemented. We prefer that you use C for the implementation, C++ or Java may be used but it is not recommended.

The assignment is divided into two separate parts. In the first part you are to describe and discuss, theoretically, what your solution will look like. This includes drawing of state-machines for connection setup and teardown, a description of how the sliding window functionality will be implemented, and the functions to be used. The sate-machines should also provide information on how to handle corrupt packets, packets out-of-order, and lost packets. **Observe**

that packets may be lost during connection setup, connection teardown, and when data is transferred (during the sliding window mechanism). All these cases must be addressed. Examples of state-machines can be found in the textbook (e.g., in the chapter on TCP) and on the Internet. One example is also provided in the end of this lab specification. **To minimize complexity you must describe the sender and receiver separately, i.e., the sender always act as a sender while the receiver always act as a receiver (a one-way stream of data).** Six state machines should be presented: connection setup, sliding window protocol and teardown for both sender and receiver. Solutions that do not satisfy any of these points will not be approved.

The second part of the assignment is to *implement* all details addressed in the first part. It is compulsory to pass the first part of the assignment before continuing with the second part.

3 The overall architecture

The reliable transport layer can be divided into several sub-layers as shown in figure 1. An *application*, for example a simple file transfer protocol, will feed data into the *sliding window layer* which in turn feeds the CRC layer with data. Finally the CRC-layer calls the UDP layer, which corresponds to the UDP socket in the UNIX environment.

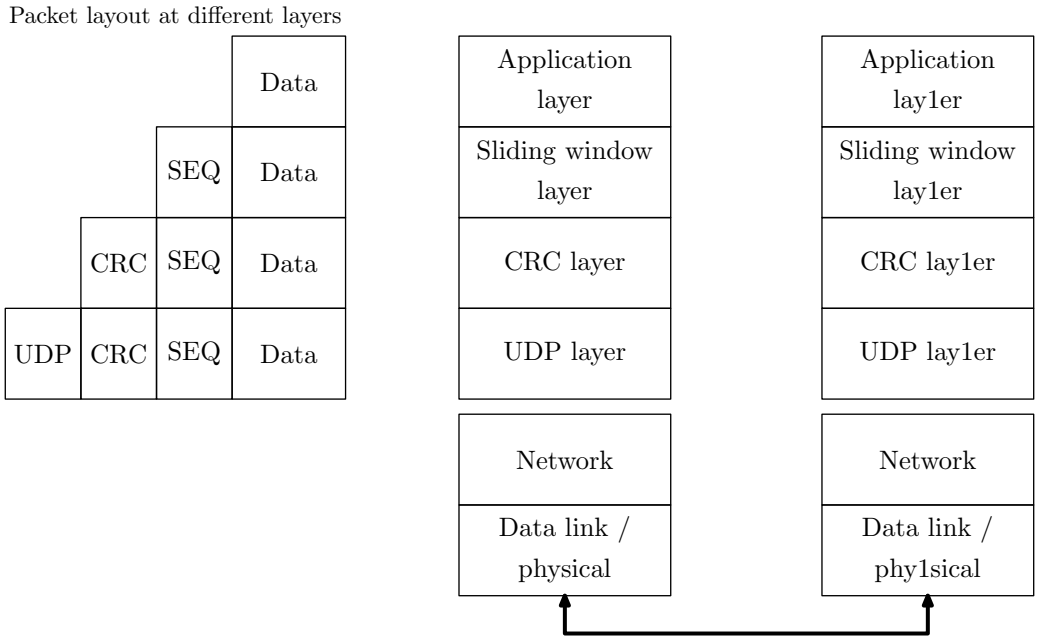

Figure 1: A layered architecture

The data is transferred to the other peer, using the existing Internet. On the other peer the packet traverse up through the protocol stack until it reach the application layer.

On the left hand side of the protocol stack a picture of the packet layout is shown. At the application layer the packet only contains *data*, then additional field are added to the packet until it finally is sent to the UDP socket. This picture is not complete since there is a need for other fields such as a field for flags (described later in this document).

You do not have to implement your solution in this layered fashion. The description above is just an example.

Your solution must be able to simulate lost packets, corrupt packets, packets out-

of-order, etc. This is a must since you need to show that the solution is correct.

4 Error check code

Each UDP packet sent must contain an error check code. This is to guarantee that the packet hasn't been modified. You have to ensure that the error check code is calculated using both the packet header and the contained data (you may exclude the UDP header information). Examples of error check codes can be found in the textbook and on the Internet.

5 Sliding window

Regarding the sliding window protocol; **you solution must be more efficient than a simple stop-and-wait protocol.** Your solution must be able to handle lost packets, retransmission of lost/corrupt packets and reorganization of packets.

You can either implement "go-back-N" or "selective repeat". Both versions of the sliding window protocol are described in the textbook. We suggest that you implement a stop-and-wait protocol first! That is, send one packet and receive an ack, send next packet, receive an ack, etc. Then refine your solution into a "go-back-N" or "selective repeat" implementation.

6 Connection setup and teardown

Since the receiving side of the reliable transport protocol can handle several senders (you do not have to implement that feature), each sender must have a unique identifier. This unique identifier is obtained during a three way handshake (much like the one TCP uses). The following procedure should be implemented:

```
S -> R: send a SYN (synchronization) packet
R -> S: respond with a SYN+ACK packet
S -> R: end handshake with an ACK
```

During the three-way-handshake the two peers must decide on the window size to be used, a unique connection identifier, etc.

When a transfer is completed the connection must be closed. This should also be done using TCP-like procedures (using e.g. a FIN flag). Note that *close()* on the UDP socket is not enough for closing the connection between S and R.

In the textbook there are state-machines describing TCP. You can use them to understand and describe how your own protocol will work. Remember that your solution must be able to handle lost packets, corrupt packets, and packets out-of-order not only during data transmission, but also during the three-way-handshake.

7 Deliverables

The deliverables for the first part of this assignment should be a written report that describes how you intend to implement the sliding window protocol, the error check mechanism, the three-way-handshake, error handling, etc. State machines describing all parts are mandatory. **Note that there must be in total 6 state machines in the report (2 for three-way-handshake, 2 for sliding windows protocol, and 2 for teardown). Solutions with state machines that do not follow the exact conventions of Mealy or Moore will NOT be approved.** Also describe any simplifications or assumptions made.

The deliverables for the second part of the assignment should be a written report and the program code. The report must include extensive debug output from your implementation. **It**

should be obvious how your solution handles packet loss, corrupted packets, the three-way-handshake, etc. In order for this to be obvious, you should put an error generator (that will generate random errors such as corrupt/lost packets) in the program function (in your code) which you use for sending packets (in both sender and receiver). In this way, your implementation of the three-way-handshake AND the sliding window protocol will be properly tested. You must comment your debug output as well as differences between the theoretical discussion made in part one and the actual implementation in part two.

Checklist:

1. State machines describing all complex functionality.
2. You should try to follow the coding standard described in http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf.
3. Extensive debug output to describe the functionality of your program.
4. Implement an error generator in send-function for both sender and receiver.
5. Changes from the first report.

A Some initial help

A.1 State machines

An example of a state-diagram is shown in figure 2. This state-diagram describes a program that removes comments from a C/C++ program. That is, the state diagram functions as a filter. There are two types of comments in C/C++ programs. The first starts with `/*` and ends with `*/` while the second type of comments starts with `//` and ends with a new line.

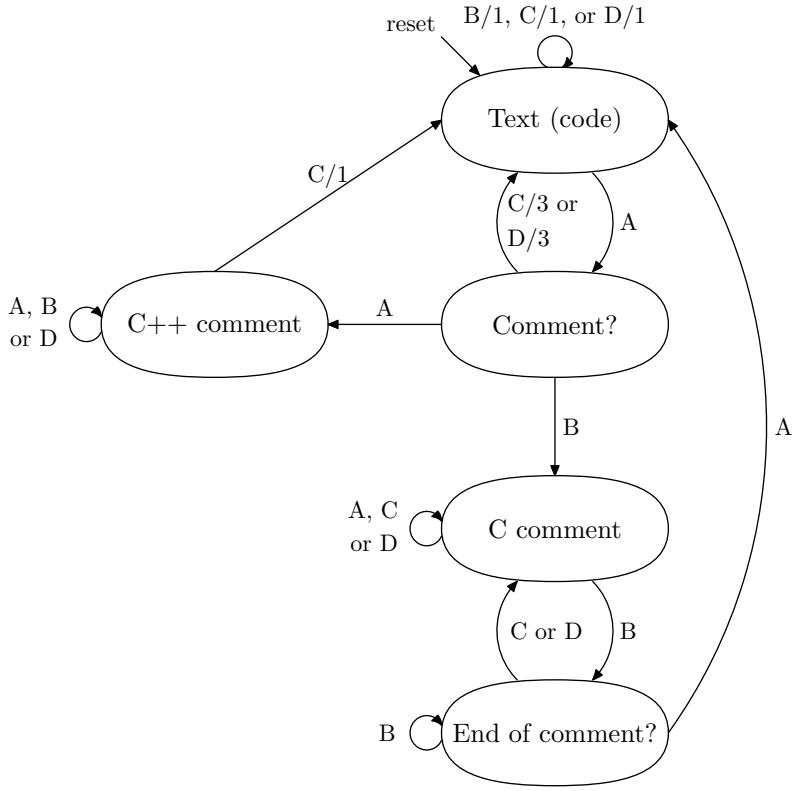

Figure 2: A state machine

Each letter (A,B,C,D) corresponds to an *event*. An event is in this example when the program reads a character. Each number (1,2,3) corresponds to the operations generated. An operation could be to print a character.

The following events are defined for the state machine in figure 2.

1. A: slash character (/)
2. B: star character (*)
3. C: new line
4. D: other character

And the following operations are generated in the state diagram:

1. 1. print the last read character
2. 2. print a slash character (/)
3. 3. print a slash character (/) followed by the last read character (i.e., operation 2 followed by operation 1)

More information about state-machines can be found in the textbook and on the Internet.

A.2 Implement a state machine

Since the state-machine in this assignment aren't too complicated you may implement your state-machine using state variables. The example below show a few possible states and how to navigate between them .The function *getEvent()* may be a function that read packets from the socket. Then, *getEvent()* will return a value depending on the received packet content.

```
#define INIT 0
#define WAIT_SYN 1
#define WAIT_SYNACK 2
#define WAIT_ACK 3

void statefunction() {
   event = getEvent();

  while (1) {
    switch (state) {
      case INIT:
        if (event == send_data) {
          state = WAIT_ACK;
          send (DATA_TO_RECEIVER);
        }
        break;

      case WAIT_SYN:
        if (event == got_syn) {
          state = WAIT_ACK;
          send (SYN_ACK_TO_SENDER);
        }
        break;

      case WAIT_SYNACK:
        ......
        break;

      case WAIT_ACK:
        ......
        break;

      default:
        // We got no packet
        if (state == WAIT_ACK) {
          resend (DATA_TO_SERVER);
        .
        .
        .
    }
  }
}
```

A.3 UDP sockets

As stated earlier you will use UDP sockets in this assignment. First, a pseudo-code example on how to use the UDP socket in the UNIX environment is described. For more information I suggest looking in the manual pages for socket(), sendto(), and recvfrom().

Remember that UDP is packet oriented while TCP (used in lab 1) is stream oriented.

```
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
  fprintf (stderr, "Can't create UDP socket\n");

memset (&destAddrUdp, 0, sizeof(destAddrUdp));
destAddrUdp.sin_family = AF_INET;
destAddrUdp.sin_addr.s_addr = inet_addr(dstHost);
destAddrUdp.sin_port = htons(dstUdpPort);

if ( (result = sendto(socket_id, send_buffer, send_buffer_size,
                      flags, dest_address,
                      sizeof(struct sockaddr_in))
                     < send_buffer_size )
  fprintf (stderr, "Couldn't send on socket\n");
```

The above example creates a UDP socket, then the destination address and port is specified. The sendto() shows how to send a single packet using the UDP socket. There are numerous examples on how to use UDP sockets in textbooks and on the Internet.

A.4 Transport layer header

Another important thing is to define your reliable transport protocol header. An example of such header is shown below.

```
typedef struct rtp_struct {
  int flags;
  int id;
  int seq;
  int windowsize;
  int crc;
  char *data;
} rtp;
```

For example, packets associated with the three-way-handshake need flags such as SYN, SYN-ACK, and ACK. Every connection between a sender and a receiver needs an unique identifier (id). To be able to handle the sliding window functionality you must consider sequence numbers (seq) and the window size. The crc-field is self-explaining. The data field in this example is a pointer to some data buffer that is going to be sent.

A.5 Timeouts

Timeouts must be used if either the sender or the receiver waits for a lost packet. For example, if the sender waits for an ack for the last sent packet, but does not receive it, then instead of being blocked by the receive function (*recvfrom*), the sender can resend the packet. Timeouts can be handled either using signals (see manual page for *signal* and *sigaction*) or by using the *select* function. Also threads or *fork* can be used.