DVA218, LAB3b

Leslie Dahlberg (ldg14001), Jonathan Larsson (jln14010)

Introduction

In this report we discuss how to implement a reliable transport protocol upon the existing UDP-protocol. This project is based upon the state machines in the report for LAB3a and written in C for Linux. For the general architecture of our protocol see the report for LAB3a.

Brief description

Our program can be used as a library by including transport.h and using the provided functions starting with “u_”. These functions include u_start(), u_listen(), u_connect(), u_send(), u_set_rcvr() and function similarly to TCP-sockets. Documentation of the usage is provided in the source code together with a demo program (main.c) which demonstrates how a simple text string can be sent reliably over the network.

The core of our transport protocol is a finite state machine implemented in C using enums for inputs and states. The state machine runs in a constant loop in it’s own thread. Additional threads are used for receiving packets from the server or from the client. These threads place received packets onto buffers that the finite state machine works with. Sending data works similarly. The user calls u_send(char* data) which places *data* onto a buffer. These buffers are stored in structs that contain sequence numbers (sequence of the next expected ACK, next incoming packet, next outgoing packet, etc.) which enable the state machine to use it as it’s sliding window.

Since our state machine runs in constant loop we were able to implement timers easily by creating structs which record start time, status and length. Then all we have to do is check whether they have timed out each time the state machine loops around.

The sliding window protocol uses Go-Back-N as it’s algorithm and uses a variable sliding window size which is negotiated during the three-way-handshake.

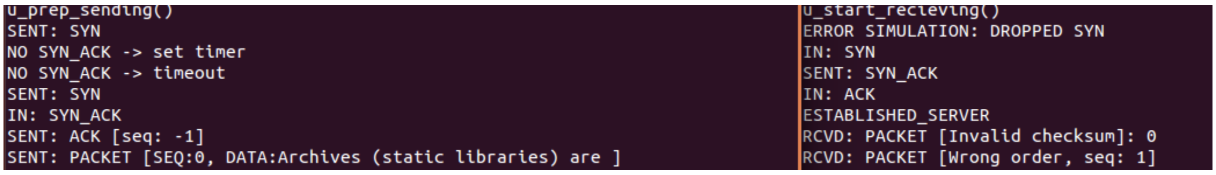
To simulate errors we randomly drop packages and invalidate checksum in both the server and receiver. In practice this also create the error of “package out of order”.

Differences between LAB3a and our implementation

- The sliding window algorithm used an infinite set of sequence numbers in the sketches. In our program we set the buffer to a fixes size and let sequence numbers wrap around that length with a modulo function.
- In our implementation every package communicates the used window size and the receiver can choose to accept it and send it back or return it’s own preference for a window size which the sender will be forced to accept.

- We use the internet checksum to verify all packets including ACKs and handshake packets
- An error generator randomly drops and corrupts packets in order to test the resiliency of our ARQ-mechanism. We use rand()%DROP_RATE == 1 where drop rate is the inverse probability of a packet being dropped or corrupted to decide when an error should occur.
- To send received packets to the application layer the user of the protocol set a function pointer which will accept incoming data and process it.
- The receiving of packages and placing of packages onto the sending buffer is handled separately from the rest of the finite state machine.

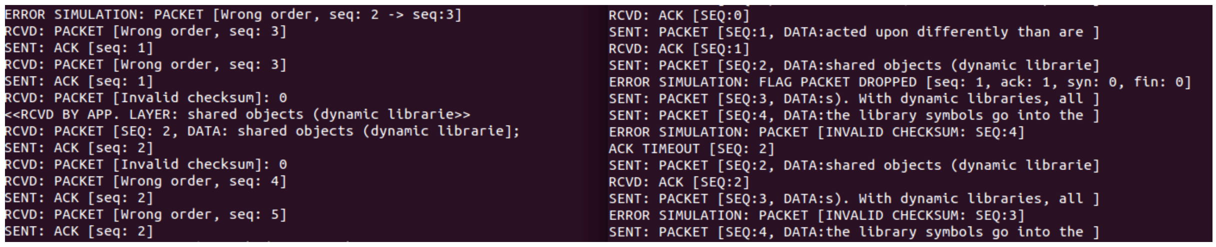
Debug output



```
u_prep_sending()
SENT: SYN
NO SYN_ACK -> set timer
NO SYN_ACK -> timeout
SENT: SYN
IN: SYN_ACK
SENT: ACK [seq: -1]
SENT: PACKET [SEQ:0, DATA:Archives (static libraries) are ]
```

```
u_start_recieving()
ERROR SIMULATION: DROPPED SYN
IN: SYN
SENT: SYN_ACK
IN: ACK
ESTABLISHED_SERVER
RCVD: PACKET [Invalid checksum]: 0
RCVD: PACKET [Wrong order, seq: 1]
```

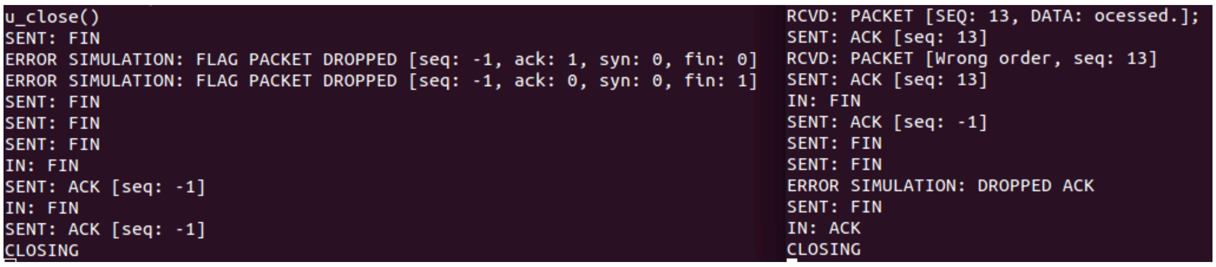
Figure 1: Client (left) and server (right) during the 3-way-handshake simulating dropped packets



```
ERROR SIMULATION: PACKET [Wrong order, seq: 2 -> seq:3]
RCVD: PACKET [Wrong order, seq: 3]
SENT: ACK [seq: 1]
RCVD: PACKET [Wrong order, seq: 3]
SENT: ACK [seq: 1]
RCVD: PACKET [Invalid checksum]: 0
<<RCVD BY APP. LAYER: shared objects (dynamic librarie>>
RCVD: PACKET [SEQ: 2, DATA: shared objects (dynamic librarie];
SENT: ACK [seq: 2]
RCVD: PACKET [Invalid checksum]: 0
RCVD: PACKET [Wrong order, seq: 4]
SENT: ACK [seq: 2]
RCVD: PACKET [Wrong order, seq: 5]
SENT: ACK [seq: 2]
```

```
RCVD: ACK [SEQ:0]
SENT: PACKET [SEQ:1, DATA:acted upon differently than are ]
RCVD: ACK [SEQ:1]
SENT: PACKET [SEQ:2, DATA:shared objects (dynamic librarie]
ERROR SIMULATION: FLAG PACKET DROPPED [seq: 1, ack: 1, syn: 0, fin: 0]
SENT: PACKET [SEQ:3, DATA:s). With dynamic libraries, all ]
SENT: PACKET [SEQ:4, DATA:the library symbols go into the ]
ERROR SIMULATION: PACKET [INVALID CHECKSUM: SEQ:4]
ACK TIMEOUT [SEQ: 2]
SENT: PACKET [SEQ:2, DATA:shared objects (dynamic librarie]
RCVD: ACK [SEQ:2]
SENT: PACKET [SEQ:3, DATA:s). With dynamic libraries, all ]
ERROR SIMULATION: PACKET [INVALID CHECKSUM: SEQ:3]
SENT: PACKET [SEQ:4, DATA:the library symbols go into the ]
```

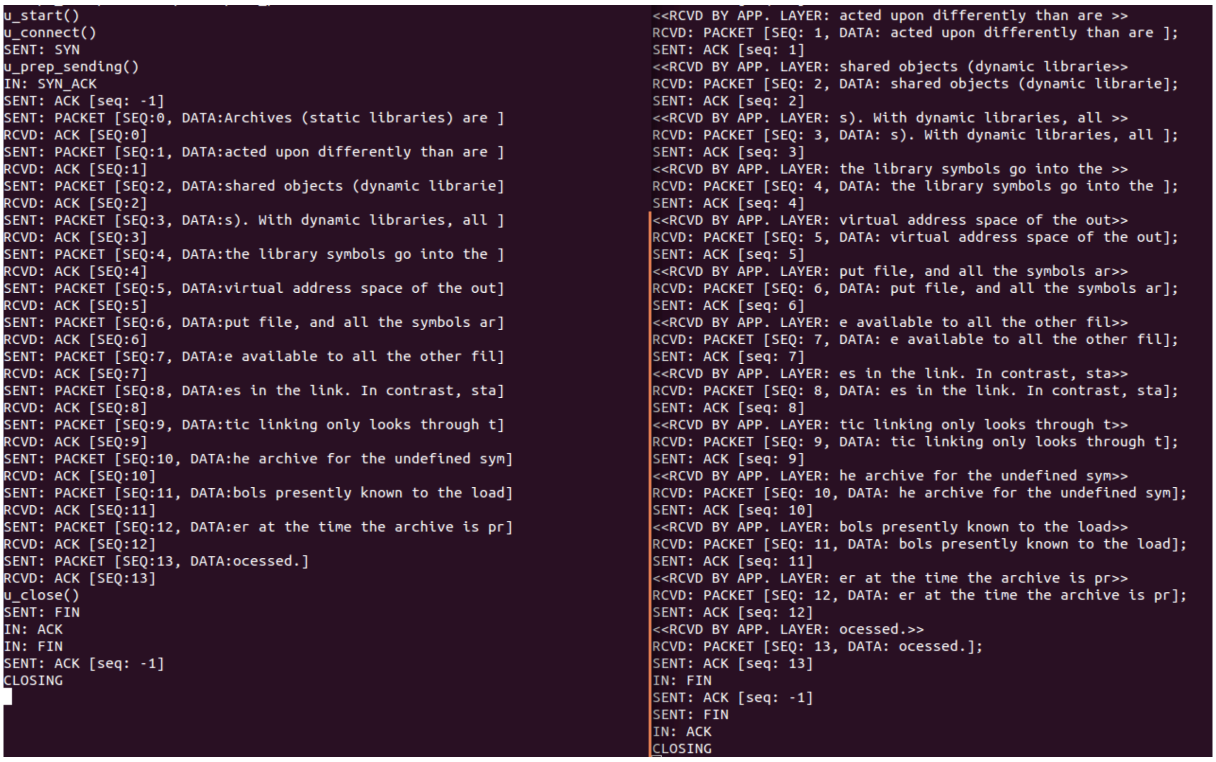
Figure 2: Server (left) and client (right) during data transfer with heavy error simulation



```
u_close()
SENT: FIN
ERROR SIMULATION: FLAG PACKET DROPPED [seq: -1, ack: 1, syn: 0, fin: 0]
ERROR SIMULATION: FLAG PACKET DROPPED [seq: -1, ack: 0, syn: 0, fin: 1]
SENT: FIN
SENT: FIN
SENT: FIN
IN: FIN
SENT: ACK [seq: -1]
IN: FIN
SENT: ACK [seq: -1]
CLOSING
```

```
RCVD: PACKET [SEQ: 13, DATA: ocessed.];
SENT: ACK [seq: 13]
RCVD: PACKET [Wrong order, seq: 13]
SENT: ACK [seq: 13]
IN: FIN
SENT: ACK [seq: -1]
SENT: FIN
SENT: FIN
ERROR SIMULATION: DROPPED ACK
SENT: FIN
IN: ACK
CLOSING
```

Figure 3: Client (left) and server (right) during connection teardown with heavy error simulation



```
u_start()
u_connect()
SENT: SYN
u_prep_sending()
IN: SYN_ACK
SENT: ACK [seq: -1]
SENT: PACKET [SEQ:0, DATA:Archives (static libraries) are ]
RCVD: ACK [SEQ:0]
SENT: PACKET [SEQ:1, DATA:acted upon differently than are ]
RCVD: ACK [SEQ:1]
SENT: PACKET [SEQ:2, DATA:shared objects (dynamic librarie]
RCVD: ACK [SEQ:2]
SENT: PACKET [SEQ:3, DATA:s). With dynamic libraries, all ]
RCVD: ACK [SEQ:3]
SENT: PACKET [SEQ:4, DATA:the library symbols go into the ]
RCVD: ACK [SEQ:4]
SENT: PACKET [SEQ:5, DATA:virtual address space of the out]
RCVD: ACK [SEQ:5]
SENT: PACKET [SEQ:6, DATA:put file, and all the symbols ar]
RCVD: ACK [SEQ:6]
SENT: PACKET [SEQ:7, DATA:e available to all the other fil]
RCVD: ACK [SEQ:7]
SENT: PACKET [SEQ:8, DATA:es in the link. In contrast, sta]
RCVD: ACK [SEQ:8]
SENT: PACKET [SEQ:9, DATA:tic linking only looks through t]
RCVD: ACK [SEQ:9]
SENT: PACKET [SEQ:10, DATA:he archive for the undefined sym]
RCVD: ACK [SEQ:10]
SENT: PACKET [SEQ:11, DATA:bols presently known to the load]
RCVD: ACK [SEQ:11]
SENT: PACKET [SEQ:12, DATA:er at the time the archive is pr]
RCVD: ACK [SEQ:12]
SENT: PACKET [SEQ:13, DATA:ocessed.]
RCVD: ACK [SEQ:13]
u_close()
SENT: FIN
IN: ACK
IN: FIN
SENT: ACK [seq: -1]
CLOSING
```

```
<<RCVD BY APP. LAYER: acted upon differently than are >>
RCVD: PACKET [SEQ: 1, DATA: acted upon differently than are ];
SENT: ACK [seq: 1]
<<RCVD BY APP. LAYER: shared objects (dynamic librarie>>
RCVD: PACKET [SEQ: 2, DATA: shared objects (dynamic librarie];
SENT: ACK [seq: 2]
<<RCVD BY APP. LAYER: s). With dynamic libraries, all >>
RCVD: PACKET [SEQ: 3, DATA: s). With dynamic libraries, all ];
SENT: ACK [seq: 3]
<<RCVD BY APP. LAYER: the library symbols go into the >>
RCVD: PACKET [SEQ: 4, DATA: the library symbols go into the ];
SENT: ACK [seq: 4]
<<RCVD BY APP. LAYER: virtual address space of the out>>
RCVD: PACKET [SEQ: 5, DATA: virtual address space of the out];
SENT: ACK [seq: 5]
<<RCVD BY APP. LAYER: put file, and all the symbols ar>>
RCVD: PACKET [SEQ: 6, DATA: put file, and all the symbols ar];
SENT: ACK [seq: 6]
<<RCVD BY APP. LAYER: e available to all the other fil>>
RCVD: PACKET [SEQ: 7, DATA: e available to all the other fil];
SENT: ACK [seq: 7]
<<RCVD BY APP. LAYER: es in the link. In contrast, sta>>
RCVD: PACKET [SEQ: 8, DATA: es in the link. In contrast, sta];
SENT: ACK [seq: 8]
<<RCVD BY APP. LAYER: tic linking only looks through t>>
RCVD: PACKET [SEQ: 9, DATA: tic linking only looks through t];
SENT: ACK [seq: 9]
<<RCVD BY APP. LAYER: he archive for the undefined sym>>
RCVD: PACKET [SEQ: 10, DATA: he archive for the undefined sym];
SENT: ACK [seq: 10]
<<RCVD BY APP. LAYER: bols presently known to the load>>
RCVD: PACKET [SEQ: 11, DATA: bols presently known to the load];
SENT: ACK [seq: 11]
<<RCVD BY APP. LAYER: er at the time the archive is pr>>
RCVD: PACKET [SEQ: 12, DATA: er at the time the archive is pr];
SENT: ACK [seq: 12]
<<RCVD BY APP. LAYER: ocessed.>>
RCVD: PACKET [SEQ: 13, DATA: ocessed.];
SENT: ACK [seq: 13]
IN: FIN
SENT: ACK [seq: -1]
SENT: FIN
IN: ACK
CLOSING
```

Figure 4: Client (left) and server (right) during normal error-free data transfer

Conclusion

Our solution works for reliable one-way communication between client and server over the internet. It can manage dropped packets, corrupted packets, packets out of order and duplicate packets.

Figures