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

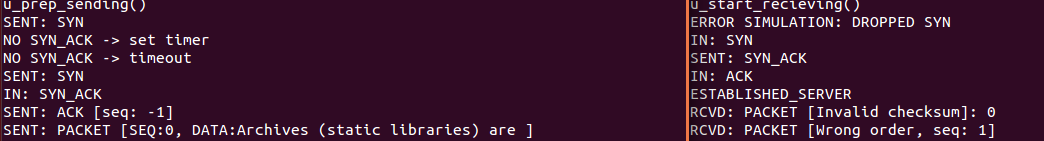


Figure : Client (left) and server (right) during the 3-way-handshake simulating dropped packets

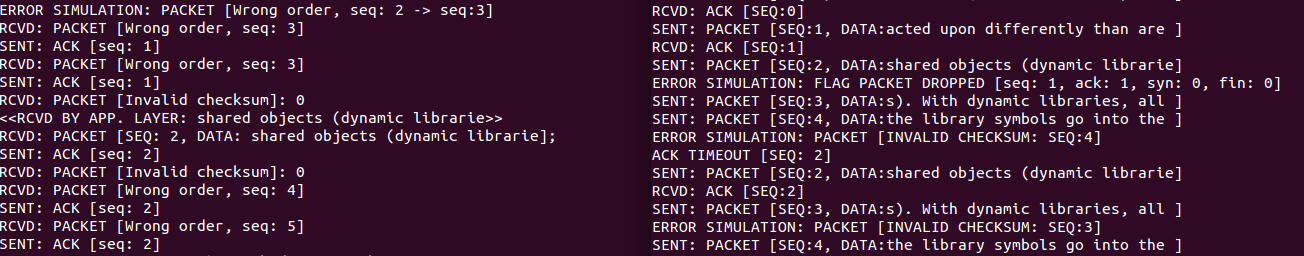


Figure : Server (left) and client (right) during data transfer with heavy error simulation

# Skärmavbild%202016-02-11%20kl.%2014.37.28.png

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

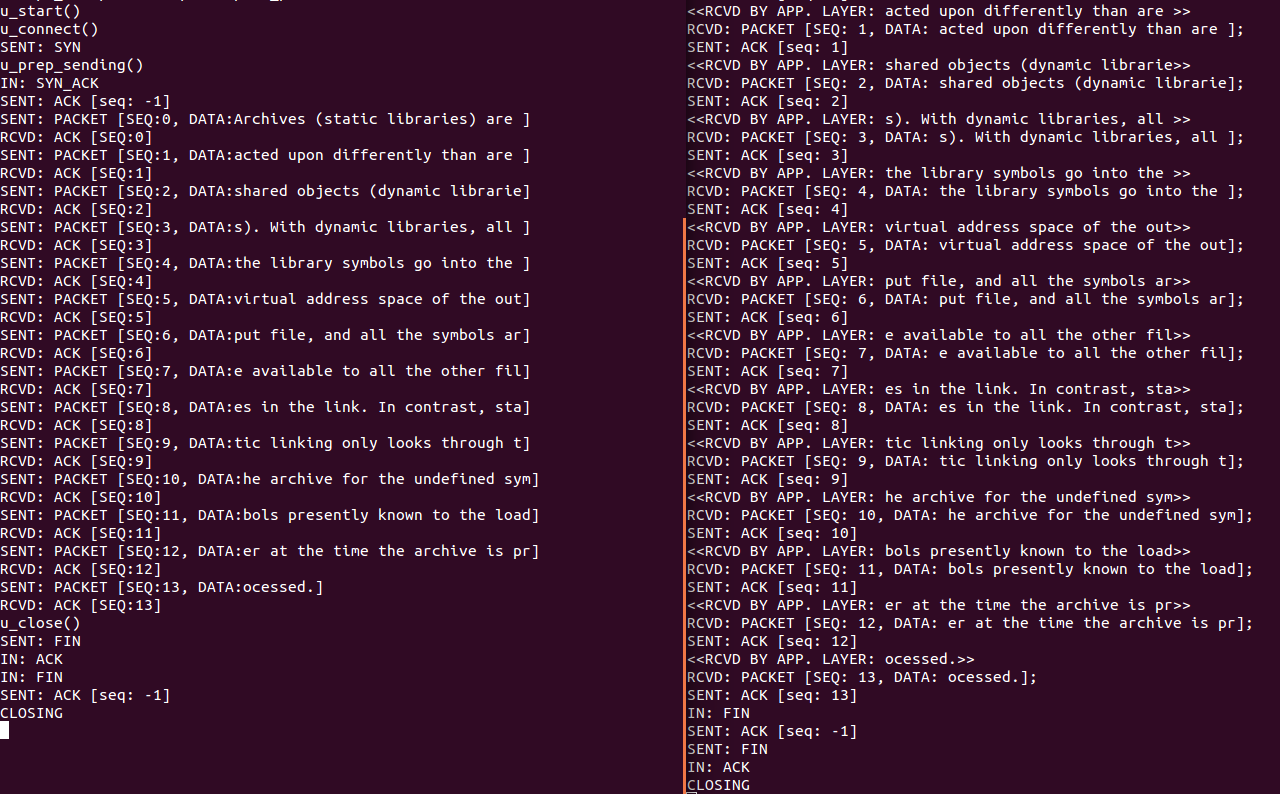


Figure : Client (left) and server (right) during normal error-free data transfer

# 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

Figure 1: Client (left) and server (right) during the 3-way-handshake simulating dropped packets 2

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

Figure 3: Client (left) and server (right) during connection teardown with heavy error simulation 2

Figure 4: Client (left) and server (right) during normal error-free data transfer 3