Datakommunikation DVA218
Lab 3B - Reliable and efficient transport protocol implementation

Fanny Danielsson, fdn16001@student.mdh.se

1. Introduction

In this lab you were supposed to implement a reliable transportation protocol on top of UDP.

At first, I started with modifying Lab 2 to make it send via a UDP socket instead of TCP. Then instead of only sending a char array, I started with sending my struct(*Figure 1*) containing the package header and data.

```
typedef struct
{
  int flags;
  int id;
  int seq;
  int windowsize;
  int crc;
  char data[MAXMSG];
}rtp;
```

Figure 1: Header and package data struct, rtp.

Then I implemented state machines for Connection setup, Connection Teardown and Sliding Window.

1.1 Special cases

The assignment also stated that a few special cases where supposed to be handled. These cases where lost packages, corrupted packages and packages out of order.

1.1.1 Lost packages

To handle lost packages, I used timeouts. To keep track of this I implemented a linked list that contained all the sent packages. This linked list has a struct that contains the information necessary for the list and timeout functionality.

When a package is sent it will be added to the linked list. Throughout the whole program a timeout thread will run to see if the first package (last package sent) in the list has timed out. If it has it will resend itself and all packages after. If we got an ACK before a timeout we can just simply remove the first package in the list and continue the process of sending and waiting for ACK/timeouts.

```
typedef struct pl
{
  rtp* header;
  time_t timestamp;
  struct pl *next;

}Packagelist;
```

Figure 2: Linked list struct, Packagelist.

1.1.2 Corrupted packages

For preventing corrupted packages, I used a Internet checksum. I choose it because it was easy to understand and implement. What it does is to divide the data of a package into a 16bit integer. If this integer is bigger than 0xffff(which means 16bits) it will subtract it with 0xffff. When the summation is done it will compute its bitwise complement and return it in network byte order.

1.1.3 Out of order packages

To handle packages out of order I used a variable in the server to keep track of the expected package sequence number, so if a package has the wrong expected package number it will just drop that package.

If we got an out of order ACK on the client side a few different things can happen depending on the sequence number. If the ACK is on a package with a higher sequence number, then the expected we know that the server has received the packages before but ACK's got lost. Then we can say that all packages before that ACK has been transmitted and remove them from the linked list.

If we got an ACK which is in the correct order, we know that the package has been transmitted and we can remove that.

If the ACK has a sequence number which is lower then expected we just drop it.

2. State machines

2.1 Connection Setup

For handling the connection setup I implemented a function which switches between different states. These states are in the client, WAIT_SYNACK and in the server WAIT_SYN. Then they both have the states INIT, WAIT_ACK, ESTABLISHED and WAIT_TIMEOUT.

The function also reads activity from the socket and if something comes in it will return the event of that incoming package. These events are in the client got_synack, and in the server got_syn and got_ack.

Depending on the state and the event it will do stuff such as send syn+ack, acks etc.

2.2 Connection Teardown

For handling the connection teardown I also implemented a function which switches between different states. These states are in the client, WAIT_FINACK and in the server WAIT_FIN. Then they both have the states INIT, WAIT_ACK, ESTABLISHED and WAIT_TIMEOUT.

The function also reads activity from the socket and if something comes in it will return the event of that incoming package. These events are in the client got_finack, and in the server got_fin and got_ack.

Depending on the state and the event it will do stuff such as send fin+ack, acks etc.

2.3 Sliding Window

For the sliding window I used the Go-Back-N technique. I thought that this was the easiest thing to implement and that is pretty much why I choose it.

2.3.1 Client

For sending packages I have a Sendmessage function that will send a choosed amount of packages. How many packages that is being sent at one time is depending on the window size.

To handle the ACK's I have a function which can receive incoming ACK's on packages that has been sent. When getting an ACK it will remove it from the Package list that I mentioned before.

2.3.2 Server

Since the server is passive in this it will just use its readmessage function to handle sending ACK's back on received packages. This function checks if a package has the flag DATA and the expected sequence number, if it has then it will send an ACK back. If not as mentioned before it will drop it.

And of course also check so the checksum is accurate.

3. User Manual

Ofcourse the first thing is to download all the needed files for this program which is the server.c, client.c, Linkedlist.c, Linkedlist.h and the Makefile.

After doing this use the command make when in the folder where these files are.

To start the server type ./server and to start the client type for example ./client localhost. If you do not have your server on your localhost type hostname -I on the server computer and use that to connect.

Since the server is all out passive you do not need to type or click anything there. On the client side although you need to answer if you want to connect to the server or not. If you type "y" and click enter it will start the connection setup, otherwise it will close the client.

After doing these steps it will just simulate the connection setup, sliding window and connection teardown. You do not need to do anything additional for it to connect, run or disconnect.

If you want to switch window size you can change the variable wsize in client.c.

Right now the client is set to send 10 packages. But if you want more you can change the defined variable packagelimit(in client.c) to something else.

Also now it randomizes between accurate package, lost/out of order package or corrupted package. 1/5 chance for it to randomize a corrupted package. Also 1/5 chance for it to send a lost/out of order package. 3/5 procent chance that it will send an accurate package. See the RandomError function for clarification.

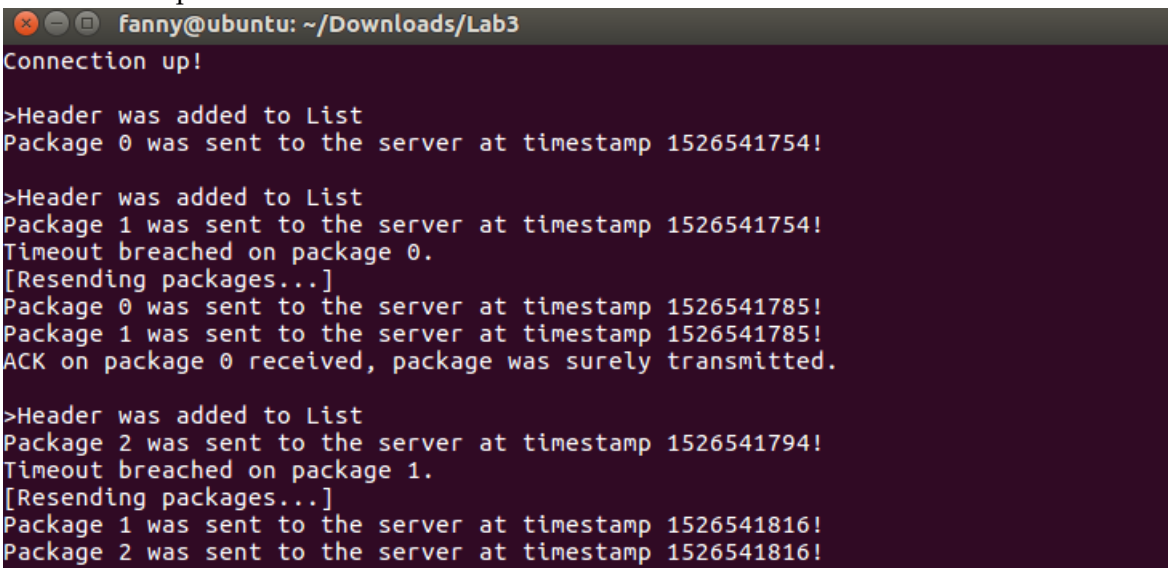
4. Test cases

Test 1: Testing the timeout functionality.

Test 2: Testing the out of order functionality.

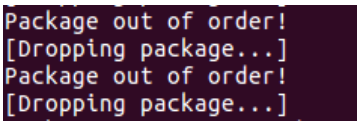
Test 3: Testing the corrupted package/check sum functionality.

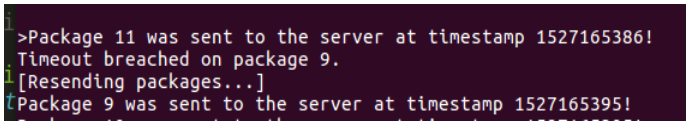
Test 1 Example:



Here we can see that the client has reached a **timeout** on package 0. It will then resend all packages that was sent after 0 and itself. Which in this case is 0 and 1.

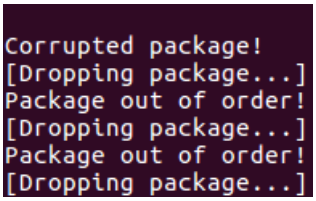
Test 2 Example:





Here we can see that the server on the left has received an out of order package and drops it. Then the client on the right will after 10 seconds reach its timeout on the packet which **got lost** or thrown because it was out of order. When it has done this it will resend all packages after that one.

Test 3 Example:



In this test we can see that the server receives **a corrupted package** and drops it. After that few more packages comes in but then they are in the wrong order and will also be dropped. The client will timeout and then resend them again.