Lab 1b) Binary UDP client

Introduction

The aim of this assignment is twofold, (1) introduce you to UDP as the transport protocol, and (2) how to handle binary data. When it comes to exchanging binary data, UDP has certain benefits. For instance, (usually) UDP delivers a complete datagram during a read operation, while during a TCP read operation, there is no guarantee that you get enough data. On the other hand, UDP will NOT let you know if some data has been lost in transit, TCP will. Similarly, if your datagram happens to exceed the minimum MTU between sender and receiver, thus causing fragmentation. Implementation wise, it may be a bit unclear what will happen at the receiver. Theoretically, the transport layer should reassemble the datagram before delivery. But as the datagram has 2 or more IP packets, each with the same loss probability, there might be problems. The easiest way is to ensure that you do not exceed the minimum MTU, even though it may require more datagrams to be sent.

This task is a variant of the first assignment, but you are now to use **UDP** and on the application-level, it is a **Binary** protocol between the client and server. Message and formatting are defined in *protocol.h*, found in the base repository [https://github.com/patrikarlos/np_assignment2 (Links to an external site.)](https://github.com/patrikarlos/np_assignment2).

Before you start the assignment, **READ** the entire assignment and reflect on it.

Requirements

You are to write a simple client. The protocol is **UDP,** and on the application-level, the protocol is a **binary-based** protocol, defined in *protocol.h*, found in the git repo associated with the assignment [1]. Within that file, you can also find information on messaging and formatting.

When the client starts, it should contact the server (you can reach it via 13.53.76.30, UDP port 5000, or XYZ:5000 (TBD)) by sending a 'calcMessage'. In that message, it should specify what type of message this is, the actual message, what transport protocol is used, and the version of the protocol it supports. In this case, the values that are to be used are shown in table 1. Please, note that these are just values when they are written to the structure you must convert them into network byte order before you transmit them.

Table 1: Client first message to server

type	22

message	0
protocol	17
major_version	1
minor_version	0

If the server supports the protocol, it will respond with a 'calcProtocol' message, see Table 2, where it includes the assignment that is to be solved. Please note in Table 2, that only the parts that have a grey highlight are actually included in the message that is to be transmitted.

Table 2: calcProtocol message

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	bits
0 1	2 3	bytes
uint16_t type	uint16_t major_version	
uint16_t minor_version	uint32_t id	
(cont uint32_t id)	uint32_t arith	
(cont uint32_t arith)	int32_t inValue1	
(cont int32_t inValue1)	int32_t inValue2	
(cont int32_t inValue2)	int32_t inResult	
(cont int32_t inResult)	double flValue1	
(cont double flValue1)	(cont double flValue1)	
(cont double flValue1)	double flValue2	
(cont double flValue2)	(cont double flValue2)	
(cont double flValue2)	double flResult	
(cont double flResult)	(cont double flResult)	
(cont double flResult)		

If the server does not support the protocol provides by the client, it responds with 'calcMessage', type=2, message=2, major_version=1,minor_version=0. If the client receives such a message it must abort, this applies all the time. The client should print a notification that the server sent a 'NOT OK' message, then terminate.

Otherwise (calcProtocol message received by the client), it should interpret the assignment and send the response back to the server. The operations are the same as before, but see how they are handled in the calcProtocol message. The server reads the client response, and compares it with its control calculation, and returns OK if it's correct, and NOT OK if there was an error, using calcMessage.

As we are using UDP as the transport protocol, which by now you should know that it's connectionless and nonreliable, we need to implement a simpler form of fault management. From the client's perspective, it means that it needs to handle timeouts. Every time a client sends a message, it needs to start a timer. If a does not arrive within 2 seconds, the message should be retransmitted. This is repeated two times (the message has then been sent 3 times, 1 ordinary and 2 retransmissions). After the third timeout, the client should print a notification that the server did not reply, and terminate. This applies to all messages the client sends, the first message (calcMessage), and the result (calcProtocol).

Let us show a couple of example:

Example 1, the client sends calcMessage. The message is lost, after 2 seconds, the client should retransmit the message. This is correctly received and handled by the server, thus the client received a reply (calcProtocol) before it timed-out. The client performs the calculation, and sends the result (calcProtocl) to the server, which replies with a calcMessage.

Example 2, the client has received calcProtocol, and executed the operation, and sent its reply. However, in this case, the server has died. So the client time-out, and does retransmission, which again time-out, this triggers the final and last retransmission, and subsequently time-out. At that time, the client terminates and prints an error message. This should have taken 3x2s = 6 seconds, since the client sent its initial calcProtocol message.

(more examples?)

Submission

Your submission **must** be in the form of a *tar.gz* file, containing your solution in a folder called np_assignment2.

The s

The solution should have a well-utilized git repository, source code, and a makefile that is used to build the solutions (client and server). Start from the repository above, then you will have the folder, Makefile, and templates for the solutions. Furthermore, the repository contains a library and a test solution, that comes handy when implementing. Make sure that you track the progression of the solution with commits to the repository, of course, you should not commit things that did not work, or that was just for testing.   I will not examine your repository handling, but I'll check that YOU made progress. I.e. it should not be a clone of my repository, and a commit 'solved'. Once you have completed and tested the solution, run make clean, then compress the entire folder (Links to an external site.), including the git repository. The compressed file (archive) should be named <bth_id>-assignment1.tar.gz. Then submit that archive here.

Testing

As I'll have several solutions to test, I've automated a large part of the testing. The automation requires that you follow the naming standard, and that 'make' and 'make clean' works. That the solutions are called client and server, nothing else. Both should accept IP:PORT as the only argument.

I'll test both of your solutions against other implementations, hence, protocol compliance is critical.

[1] https://github.com/patrikarlos/np_assignment2