ECE 151: Communication Networks **Second Semester 2018-2019**

Machine Problem 1: Implementation of Go-Back-N and Selective Repeat protocols

Deadline: Online in UVLE 11:59PM 12 April 2019

This project will require you to work in groups of 5-6 students assigned by the teacher. Aside from a deeper understanding of technical aspects of pipelined protocols, this project is also intended for you to learn how to work together in larger teams. You are encouraged to start early and plan early as well, and leave enough time to work on the write-up at the end.

There will be a peer-grading component. You have 3 weeks to finish the project. During the end of the second week, groups may elect to remove members who are not making any significant contribution to the group. Inform the teacher if this is the case.

Introduction

In this project you will be implementing the two pipelined protocols discussed in class, Go-Back-N (GBN) protocol and Selective Repeat. Your task is to make sure that data sent on one end appears on the other end exactly as it was sent, using UDP as the underlying transport protocol.

In this project, only one end sends data. The other end only sends acknowledgements back to the sender. You must however implement a simple congestion control mechanism. When packets are dropped, the protocol will start sending at a slower rate (window size=1), and speed up when it successfully receives an acknowledgement (window size=2).

The sender side (client) is the host which initiates the connection and will send a file to the receiver (the server), which in turn saves the data into a file as well. The receiving side is the one always waiting for a connection and will receive the file from the sender.

The sender and receiver programs should be called at the command prompt using the following parameters:
Sender $: `sender <dest_IP> <dest_port> <filename> <proto>`

dest_IP	= destination IP address (IP address of host to send file to)
dest_port	= destination port_number
filename	= name of file to send
proto	= pipelined protocol to use, either "`gbn`" or "`sr`", for *Go-Back-N* or *Selective Repeat* respectively

Receiver $: `receiver <src_port> <filename>`

src_port	= port number receiver will listen to
filename	= name of file to save received data from the sender

Note that for this project you can assume that there is only a maximum of ONE sender sending at any given time to the receiver. It is the sender which "contacts" the server to initiate a connection and is also the one which initiates the termination of the process upon successfully sending all the contents of the file.

You must build your protocol using the system calls provided by UDP to applications:

- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`
- `bind()`

It is highly recommended that you read the manual pages for these system calls. You can use Beej's guide (http://beej.us/guide/bgnet/) or online sources.

Important: You must use only the UDP system call functions mentioned above to perform network operations. For example, you are NOT allowed to use such system calls as `send()` in place of `sendto()`. However you are allowed to implement additional helper functions as necessary.

Specifications

You can find the description of the “Go-back-n” protocol and “Selective Repeat” in sections 3.4.3 and 3.4.4 from the textbook by Kurose and Ross "Computer Networking: A top-down approach". Copies of this book are available from the EnggLib II reserve section and I believe there is also a PDF version online. You may also checkout this visualization at http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

You need to read the textbook first as some of the details of the behavior will be specified there, eg what to do when an out-of -order packet is received, what segments to retransmit in the event of a time-out, depending on whether you are using Selective Repeat or Go-Back-N, etc.

The protocol you are to implement in this project is a simplified version of the protocols described in the textbook. You will only need to consider window sizes of 1 and 2 for your implementations.

You should design a simple state machine for your protocols and implement it. Your protocols should have a connection setup and a connection teardown. In the connection setup, a so-called SYN packet must be sent to the server to initiate the connection. The server then replies with a SYNACK packet to the client if it accepts the connection or with a RST packet if it rejects the connection. In the connection teardown, the party wanting to finish the connection should send a FIN packet to the other end host. This other host, in turn, replies with a FINACK packet to acknowledge the first FIN packet.

Your sender will have a basic congestion control mechanism. It will switch between two modes: fast mode (n = 2) and slow mode (n = 1). In slow mode, the next packet will not be sent until the sender receives the acknowledgement for the previous packet. In fast mode, a second packet may be sent before receiving an acknowledgment for the previous packet. The protocol will start in slow mode. Whenever it is in slow mode and receives an acknowledgment for the most recently sent packet, it will switch to fast mode. When it is in fast mode and a packet needs to be retransmitted because the sender did not receive acknowledgement from the receiver, the program switches to slow mode.

Your segment should have the following header fields (data type defined in stdint.h)

```
uint8_t  type;          /* packet type                    */
uint8_t  seqnum;        /* sequence number of the packet   */
uint16_t checksum;      /* header and payload checksum     */
```

The sequence number at the start of the connection is randomly generated (for debugging purposes you can set this to start with 1 first). The type field identifies the type of packet being sent:

- SYN,
- SYNACK
- ACK
- RST
- DATA
- FIN
- FINACK

The *seqnum* is the sequence number of the packet. If it is an ACK packet, the *seqnum* field contains the sequence number of the segment being acknowledged. The checksum field is used to check the integrity of the received packet. It is calculated by the packet sender and checked by the packet receiver. The checksum should take into account the entire header and the carried data. You will need to provide the function that will calculate the Internet checksum as discussed in the textbook.

The SYN, SYNACK, FIN, FINACK, and RST packets contain no DATA field. They are only composed of the type and checksum fields. It is up to you how to decide what to do with the sequence number in this case (as seqnum is part of the header field, it should always appear as part of the header).

Some helpful implementation aspects

In order to test if your protocol works with losses/corruption, you should not call the sendto() function directly. Although unreliable, in reality data transmission in local networks can have little loss and almost no corruption. Instead, you should write a function `perhaps_sendto()`. The `perhaps_sendto()` function selects packets at random to be dropped based on some probability (it is up to you to determine what pdf to use and their parameters and which functions to use to implement this). Aside from this, it has the same parameters as the regular `sendto()` function used to send UDP datagrams.

For this project, you can use fixed values for the for the re-transmission timer(s). It is a predetermined value that does not change throughout the connection. Aa value of 1-2 seconds should work well, but you can choose whatever value you prefer. You should also assume that if the same packet is sent five times without receiving any acknowledgment, the connection is broken and you can already terminate the connection.

Timeouts can be implemented using the `alarm()` or the `setitimer()` functions. Upon expiry of timer, a SIGALRM signal is sent to the calling process which runs a signal handler. You need to define and bind a function in your code to handle this SIGALRM signal, which can be done using the `signal()` or `sigaction()` functions (use only one, mixing the two is not recommended).

You will also need to determine whether a sleeping processs, eg `recvfrom()`, is woken up due to a timer expiring or a packet has arrived. If a timer expires while the process is waiting for a packet to arrive, the signal handler is executed first. After the signal handler returns, the function that caused the sleep (`recvfrom()`) returns and the regular execution of the process resumes. The return value of that function should be checked to see if it returned due to a timer expiring or due to actually receiving a UDP datagram.

Deliverables

Note that the programming language that you use should be C and your program should run on a Linux machine. I will be checking it on a Linux machine.

You need to submit all commented source files in a single zipped file in UVLE. You are also required to submit a short report with the following contents:

- high-level implementation details of your program,
- the aspects which you found especially difficult to design / implement
- comparison of the pros and cons of GBN and Selective Repeat, eg ease of implementation, performance, etc.

Deadline: Online in UVLE 11:59PM Fri 12 April 2019

Intellectual Dishonesty

As this is a group project, students may collaborate only with the members of their group. Any form of collaboration BETWEEN groups is considered as intellectual dishonesty already. Only the members of the group may contribute towards the design and implementation of the code to be submitted for the MP1. Copying code from the Internet or having someone else do the MP1 for you are also considered as cheating.

I will be running your submissions through a software similarity test and a plagiarism checker. Groups who submit substantially similar work with another group or similar to code available on the Internet will be investigated for intellectual dishonesty.

Credits: “Networked and Distributed Computing” course, Cornell University.