CMSC 417 Computer Networks **Fall 2015**

Programming Assignment 1

Assigned: September 10 *Due: September 17, 11:59:59 PM.*

1 Description

In this assignment, you will modify your TCP client and server from Assignment 0 to instead use UDP as your transport layer protocol. In contrast to TCP, UDP provides fewer properties and guarantees on top of IP. As in TCP, UDP supports 2^{16} ports that serve as communication endpoints on a given host (which is identified by the IP address). Unlike TCP, UDP is a connection-less protocol, meaning that a source can send data to a destination without first participating in a "handshaking" protocol. Additionally, UDP does not handle streams of data, but rather individual messages which are termed "datagrams". Finally, UDP does *not* provide the following guarantees: 1) that datagrams will be delivered, 2) that the datagrams will be delivered in order, 3) that the datagrams will be delivered without duplicates.

You can find information on the UDP socket API calls in the Donahoo/Calvert book. For more information on UDP, you can refer to Section 5.1 and the introduction to Section 5 in the Peterson/Davie book.

This document will only contain the differences between the two assignments, so please refer to Assignment 0 for any details you may not recall.

2 Protocol

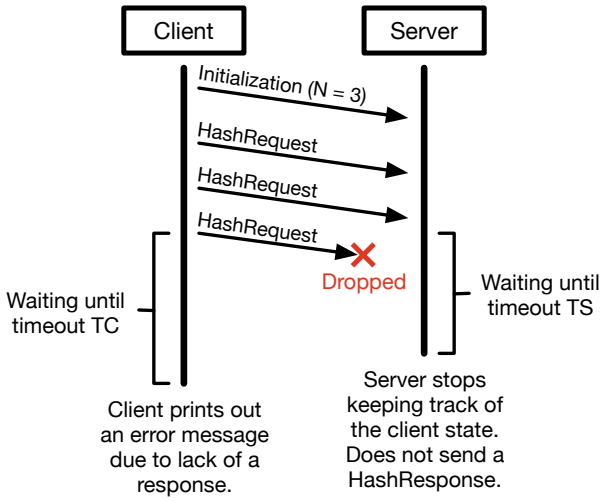

Figure 1: Example message sequence diagram for the assignment protocol. There is a dropped packet that forces the protocol to not successfully complete.

The protocol follows that of Assignment 0. However, due to the limited properties that UDP

provides on top of IP, there are some new cases to handle in the implementation of your client and server.

The first case is that of a dropped packet, as shown in Figure 1. The client sent an initialization message with $N = 3$ to the server, and followed up by sending the 3 HashRequests. The last HashRequest is not delivered in its entirety to the server, due to a packet being dropped in the network. Note that in the example we are assuming that an entire HashRequest was dropped; however, the HashRequest may be larger than the amount of data that a single UDP datagram payload can hold. The server handles these cases by waiting for a timeout (TS) for each HashRequest to come in. If the timeout triggers, then the server discards any state it kept on the client (e.g., I have received K of N messages). After the client sends its last HashRequest, it also waits for a timeout (TC) for the HashResponse to come in. If the timeout triggers, then the client prints out an error message.

Additionally, the packet loss could occur in an earlier HashRequest. The server may, depending on the sizes of the current and subsequent HashRequest(s), receive enough data to satisfy the current HashRequest *even though that data would contain parts of the subsequent HashRequest(s)* In this case, it would unknowingly compute an invalid hash. The server will later discover the packet loss occurrence most likely from parsing an incorrect ID (i.e., not equal to 0x0417) from the next HashRequest. Note that it is possible that the ID matches (1 in 2^{16} chance if the request contains bytes selected uniformly at random); in such cases, subsequent fields may contain invalid values, or a timeout may occur.

As mentioned in the description, packets may also be delivered out-of-order or as duplicates. You should also handle these cases, which have similar means of detection.

Datagrams Both the client and server will send UDP packets that are less than or equal to 4096 bytes in size. Note that UDP packets contain 28 bytes of header information (which includes IP and UDP headers). Thus, the data payload contained in a UDP packet should be less than or equal to 4068 bytes. This maximum payload length is important for creating appropriate buffers to pass into the `recvfrom` function.

A datagram must not contain multiple protocol messages; however, a single protocol message may be made up of multiple datagrams (due to the aforementioned size constraint).

3 Server Implementation

The server will be a command line utility, which takes the following *additional* arguments:

1. **-t <Number>** = Timeout value for the server (TS) in seconds. Represented as a base-10 integer. Must be specified, with a value in [0,30]. A 0 value means that the server will wait forever (i.e., no timeout).

An example usage is as follows:

server -p 41714 -s newsalt -t 5

If datagrams get lost, duplicated, or delivered out-of-order, the server should detect this issue at the earliest possible point. The first method is by checking the fields of the HashRequest for valid values, meaning that ID should be 0x0417 and the length is within the limit imposed on SMax (i.e., 2^{24}). If this method of detection finds the error, the server must print out a single line that contains the field name (ID or Length) and the infringing value. The field name and value must be

separated by a space. The value must be displayed in hexadecimal (with a leading "0x") for the ID, and in decimal for the Length.

The second method is via timeouts. For each HashRequest, the server should wait for a period of time specified by the TS command line argument. This timeout period is reset for each datagram the server receives as part of the HashRequest. If the timeout triggers, meaning that the complete HashRequest was not received, the server should drop all state information it holds on the client. If this method of detection finds the error, the server must print out a single line consisting of the text "Timeout".

The third method is via datagram lengths. Since a single datagram will never contain multiple protocol messages (either in part or in whole), then the length of received datagrams provides insight into potential errors when compared to what is expected. For example, the server may be waiting for the last datagram of a HashRequest to arrive, with a length less than 4068 remaining in the HashRequest payload. If the server sees a packet of length 4068 (i.e., from the next HashRequest), the server detects a length violation and knows that the missing packet must have been dropped (or out of order). If this method of detection finds the error, the server must print out a single line consisting of the text "Length", the expected length of the datagram, and the length of the received datagram. These values must be separated by a space, with the length value displayed in decimal. Note that this method, if not employed, would typically result in the errors being detected by the first method.

Throughout the course of the server running, the server should print out any hashes that it successfully computes for the client. The server must print these in the format as specified for the client.

All of these printed lines should be prefixed by the string representation of the IP address and port for the corresponding client (e.g., "10.0.2.15:4171"). An example of the printouts is as follows, where the server was able to compute two hashes prior to hitting an incorrect ID in the next HashRequest:

10.0.2.15:4171 0x5f70bf18a08607016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef

10.0.2.15:4171 0x5f70bf18a08607016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef

10.0.2.15:4171 ID 0xbeef

Once again, the server is memory limited to 1 MB. However, since the server is now witholding its HashResponse until the end of the protocol, this memory limit will not take into account the memory allocated to maintain complete HashResponses for all active clients. Not only should you maintain the proper buffer management practiced in Assignment 0, but you should also make sure that the server only maintains state on active clients.

4 Client Implementation

The client will be a command line utility, which takes the following *additional* arguments:

1. **-t <Number>** = Timeout value for the client (TC) in seconds. Represented as a base-10 integer. Must be specified, with a value in [0,30]. A 0 value means that the client will wait forever (i.e., no timeout).

An example usage is as follows:

client -a 128.8.126.63 -p 41714 -n 100 –smin=128 –smax=512 -f /dev/zero -t 15

After sending the last HashRequest to the server, the client will begin waiting for a period of time specified by the TC command line argument. This timeout period is reset for each datagram the client receives as part of the HashResponse. If the HashResponse is received, the client will print it out in the same manner as Assignment 0. If the timeout triggers, meaning that the complete HashResponse was not received, the client will print out a single line containing the text "Timeout" and exit with an error status.

5 Grading

Your project grade will depend on the parts of the project that you implement. Each letter grade also depends on successful completion of the parts mentioned for all lower letter grades. Assuming each part has a "good" implementation, the grades are as follows:

Grade	Parts Completed
C	Protocol completes using UDP when there are no errors
B	Handle timeouts appropriately with 1 Client
A	Handle timeouts appropriately with >1 Clients

6 Additional Requirements

1. Your code must be submitted as a series of commits that are pushed to the origin/master branch of your Git repository. We consider your latest commit prior to the due date/time to represent your submission.

2. The directory for your project must be called 'assignment1' and be located at the root of your Git repository.

3. You must provide a Makefile that is included along with the code that you commit. We will run 'make' inside the 'assignment1' directory, which must produce two binaries 'server' and 'client' also located in the 'assignment1' directory.

4. You must submit code that compiles in the provided VM, otherwise your assignment will not be graded.

5. Your code must be -Wall clean on gcc/g++ in the provided VM, otherwise your assignment will not be graded. Do not ask the TA for help on (or post to the forum) code that is not -Wall clean, unless getting rid of the warning is the actual problem.

6. You are not allowed to work in teams or to copy code from any source.