CS 39006: Networks Lab

Assignment 3: Basic Socket Programing (Working with A Single Threaded File Transfer Application)

Date: 25th January, 2018

Objective:

The objective of this assignment is to understand the basic functionalities of POSIX socket programming through a single threaded file transfer application.

Submission Instructions:

The code needs to be written in either C or C++ programming language. You need to prepare a report that will contain the followings.

1. **Documentation** of the code along with **compilation and running procedure**, **sample input** and **sample output**
2. **Observations** from the experiments.
3. Intuitive **justification** behind the observations

Also include a **makefile** to compile your code. You need to submit the report, code (only C/C++ source) and Makefile in a single compressed (tar.gz) file. Rename the compressed file as `Assignment_1_Roll1_Roll2.tar.gz`, where `Roll1` and `Roll2` are the roll numbers of the two members in the group. Submit the compressed file through Moodle by the submission deadline. The submission deadline is: **February 1, 2018 02:00 PM.** Please note that this is a strict deadline and no extension will be granted.

Please note that your submission will be awarded zero marks without further consideration, if it is find to be copied. In such cases, all the submissions will be treated equally, without any discrimination to figure out who has copied from whom.

In this assignment, you need to create a file transfer server and a file transfer client using sockets. For each server and client, you need to create a TCP and a UDP variant for the transport layer services. The basic file transfer application that you have to implement, is as follows.

1. The **file transfer client** sends a file to the server (file size should be at least 1 MB)
2. The **file transfer server** receives the file, and returns back an acknowledgement (after the file transfer is complete) with the MD5 checksum of the file.
3. The protocol works as follows.
    a. The client first Informs the file name and the file size to the server by sending a hello message.
    b. The server acknowledges the hello message.
    c. The client forwards file data over the stream/datagram socket to the server
    d. The server receives the data, reconstruct the file at the server side, creates the MD5 checksum of the entire file
    e. The server acknowledges the client with the MD5 checksum of the file.
    f. The client creates MD5 checksum of the original file before transfer, and matches it with the received MD5 checksum from the server. The client prints a message at the console “MD5 Matched” or “MD5 Not Matched”, and exists.

You need to implement two variants of the protocol - one using reliable data delivery through TCP, and another using unreliable data delivery through UDP. The details of each are as follows.

Client/Server Using TCP Socket

Here all the communications between the client and the server will be over TCP sockets. As TCP handles reliability and byte ordering by itself, so the implementation of the protocol is straightforward.

Client/Server Using UDP Socket

Here all the communications between the client and the server will be over UDP sockets. As UDP is unreliable, you need to implement reliability on top of transport layer, that is at the application layer itself. To ensure reliability, the client implements the following additional functionalities at the application layer.

1. Divide the file into chunks of 1 KB (last chunk can be less than 1 KB)
2. Inform the server about the file name, the total file size and the number of chunks to be sent
3. Add an application header to every chunk. Each header should contain following fields,
    a. Sequence number - 4 Bytes
    b. Length of the chunk - 4 Bytes

This additional information appended with the application header will help you to ensure the reliability.

4. Forward the chunks using a Stop and Wait protocol with `timeout=1 second,` to ensure the reliability of data transfer at the application layer. The details of the protocol is given later.

For both the TCP based and UDP based implementation of the file transfer application, capture the packets using Wireshark at the receiver side (**server**) and at the sender side (**client**), and compute the followings.

a. Total number of segments received for TCP and the segment size distribution
b. Total number of datagrams received for UDP and the datagram size distribution
c. Total number of retransmitted segments for TCP
g. Total number of retransmitted datagrams for UDP
h. Total time to receive the file for TCP and UDP. Give justifications if you see any time difference between the two protocols.

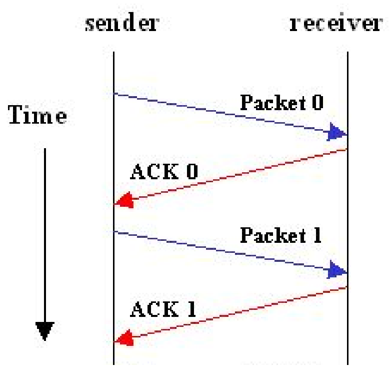
Note

1. It is better to run the client and server in separate machines - then you may be able to see the effect of unreliable data delivery through UDP.
2. Sample code for TCP/UDP server-client are provided.

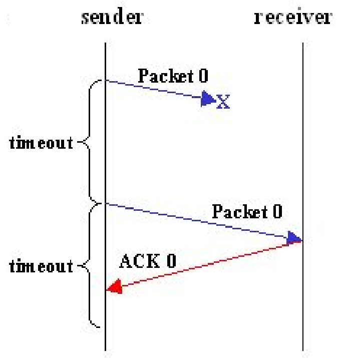
Stop and Wait Protocol for ensuring Reliability

This is a very simple protocol by which one can provide reliable data transfer over an unreliable channel. Let us see how it works. Here, we use the term "packet" in a more general way, which in this application scenario, is a data chunk from the file along with the application layer header.

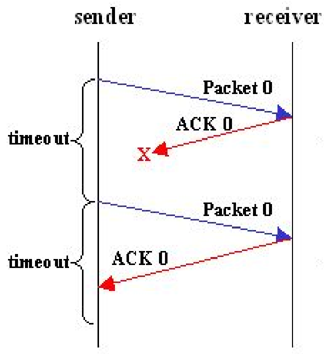
1. Say there is a **sender** and a **receiver**. The sender sends one packet to the receiver
2. The receiver, upon receiving the packet, sends an **acknowledgement**, or **ACK**, to the sender, informing that it has received the packet



3. The sender, only upon receiving the **ACK**, sends the next packet
4. In case the **data packet** is lost in transmission, the sender waits for an ACK till a **timeout** value, after which the sender re-sends the packet



5. In case the **ACK** for a corresponding data packet is lost in transmission, the sender waits for an ACK till a **timeout** value, after which the sender re-sends the packet. When the receiver gets the duplicate packet (you can check this from the sequence number field), it ignores the packet but sends a fresh **ACK**



6. In case of duplicate **ACKs**, which can happen due to network delays, the protocol employs **sequence numbers**, which is unique to each packet, so that **ACK** of packet 1 is not mistaken for the **ACK** of packet 2