Computer Networks

PROJECT FALL-2015

Due Date: Sunday, 15th November, 2015 @ 11:30 pm

GENERAL INSTRUCTIONS:-

- Programming can be done in either C or C++.
- Submissions should include the annotated source code.
- Programs that will not be executed will get a low grade.
- Make sure your programs do not crash when given bad input, but instead provide warning messages.
- Copy cases will get F grade.
- Marks will be allocated for code style. This includes appropriate use of comments and indentation for readability, plus good naming conventions for variables, constants, and functions. Your code should also be well structured (i.e. not all in the main function).

SUBMISSION INSTRUCTIONS:-

You should submit your complete code by the date posted on SLATE. You need to submit the following:

- Source code files
- Makefiles
- A README file describing your code and amount of implementation details you have successfully incorporated. Also clearly state any assumptions made during the implementation part.
- A report containing details of design decisions and implementation details.
- The compressed files (either rar or zip) are required to be submitted using SLATE in a timely manner.
- Submission Folder name should be Roll No1_Roll No2 e.g. 13i-1111_13i-1112

Project Statement

Motivation:

More or less similar project statement has been around from last 13 years in FAST ISB Computer Networks course, therefore it is a legacy of our university now and it is a wonderful one. All instructors of Computer Networks course made to do this project that made students to learn most about Networks during BS (CS) degree. The beauty of the project is that it can't be copied because only the original creator of the code and design understand the complex implementation and no one else can justify most of the parts of the implementation. This project has been an inspiration for many keen learners and I hope most of you will make the best out of it this time also.

Introduction:

In this project, you will design and implement UDP-based peer to peer applications for simple file transfer. You will manage the transfer of file data from peer to peer applications using only a UDP channel with unusually flaky delivery. The project allow you to better understand the common transport layer protocol mechanisms by applying your knowledge of fragmentation, error detection and/or correction, and retransmission to construct a Reliable Delivery Protocol (RDP) on top of UDP. Multiple POSIX threads in the server will allow logically concurrent operation between many clients. Because of the difficulty of managing concurrent accesses to data structures, you are strongly encouraged to use distinct UDP sockets for handling requests and to support each RDP stream in the server/peers. However, since you really want to learn as much as possible, therefore you are supposed to implement a two layer stack that will consist of Reliable Data transfer Protocol (RDP) layer Peer to Peer network architecture (Application Layer).

Peer to Peer Architecture:

At application layer you have to implement a very simple peer to peer architecture. You have to implement a centralized peer to peer architecture in which there is a central node with which all peer nodes should be register. Central node should be live all the time. Peer nodes will come and register with the central node. For registration each peer node has to register its IP address, Port number and all files and data to the Central node. Central Node can write this information into a file unless a peer explicitly asks to release this information at time of exit. Any peer can request Central node for this information to access data from other peers. Any peer can access any other peer for the data and files it held and use Reliable Data transfer Protocol (RDP) layer to perform a successful transaction.

RDP layer:

The primary goal of this layer is to improve your understanding of the issues and approaches utilized to build a reliable byte stream abstraction across an unreliable network with artificial segment boundaries. Your protocol should run on top of UDP, which is an unreliable best effort datagram communication service. UDP provides you with the necessary demultiplexing service for process-to-process communication, so you do not have to replicate this in your transport layer. However UDP may not deliver some messages, may deliver out of order, and may deliver duplicates of a message. The maximal segment size supported by UDP is 32768 bytes. Therefore you might have realized that use of UDP is asking application writers to design their own algorithm on top of an unreliable stream. This layer should ensure normal operations of a reliable layer. In this document only minimal design requirements are provided and you have the freedom to create your own design. So *be creative* but you should be able to justify your own design. Some of the alternatives that you should consider include:

- Packet size
- Header Format
- Sliding Window Protocol (GBN/Selective repeat/Hybrid)
- Flow Control
- ACK vs ACK/NACK
- Piggybacking or not
- RTT estimation
- Error control using CRC-16 ($x^{16} + x^{13} + x^{12} + x^{11} + x^{10} + x^{8} + x^{6} + x^{5} + x^{2} + 1$)
- Timer management i.e. timeouts using timer linked lists and doing retransmissions.

Please design appropriate calls so that functionality of this layer can be implemented. Since we are quite kind hearted instructors, hence you have been spared from taking care of delayed duplicates in your RDP layer; however, enthusiastic students are most welcomed to work in this area, hoping to get extra credit.

You have to provide us with a proper design document. Your design document must clearly answer following parts:

Part 1(a): Message Formats

Describe the message formats you will use for sending requests, indicating the nonexistence of a file, indicating the length of a file, sending data, and sending acknowledgements. Explain and justify your framing decisions for each format; remember that UDP provides length-based framing for you, but if you put more than one piece of information into a message, you must solve the framing problem again.

Part1 (b): Window Data Structures

Define data structures to keep track of send and receive window information. On the sender side, for example, this structure should include the last acknowledgement received (LAR) and an array of outstanding segments with timeout information. Provide versions of your data structures annotated with explanations of the purpose of each field.

Part1(c): Window Operations

Briefly outline the steps necessary for determining the minimum timeout value across outstanding packets in the send window (i.e., packets already sent at least once but not acknowledged). Also outline your strategy for processing acknowledgements. Do not turn in C code. Rather, provide descriptive paragraphs, supplemented if necessary by pseudo-code of the form used in algorithms textbooks.

Part1 (d): Error Correction

Discuss the conditions under which the server should use forward error correction. In particular, provide an Analysis of the minimum number of packets sent to transmit N bytes of data in packets no larger than L bytes for both an error detection scheme based on CRC-16 and an error correction scheme based on 2D parity. Assume that every packet has an independent probability p of suffering a single-bit error, and that only single-bit errors occur.

Implementation Details:

You need to configure your client and server with the run time configurations. Client configuration file should have at least following attributes:

- Central Server IP
- Central Server port
- Peer IP
- Peer Port
- File name
- Packet size
- Server Timeout
- ACK number to drop
- ACK number to corrupt

In the start you will read a simple text file in which you have saved these configurations. Further, Central node and peer side, you have to implement the RDP which will run according to the giving configurations. You have to show a well structured message order to support the readability of your program.

Messages:
To assist in debugging and in simplifying testing of your code, please prefix printed output with human-readable messages. For example,

ERROR: all error messages
REJECTED: bad request message received
NO FILE: requested file not found
ACCEPTED: valid request for existing file received
OBTAINED: client knows file length and is ready to receive data
SENT #: sent packet/ACK number #
RESENT #: resent packet/ACK number #
ACK #: server received ACK #
SEGMENT #: client-received segment #

And you are, of course, encouraged to design your own error message, if you feel they would help in debugging.

Testing Layer:

Some of intelligent students must have already realized that how the instructors are going to test the software. Ok in real life customers hardly write test code rather writing test code is the responsibility of software developer, customer only provides test cases. Hence an important part of the project could be to write test code that would make the instructor confident that your code is working properly. The purpose of test software would be to make your code crash at specific places or on specific errors. The ideal candidate would be the one who can write test code that can make software of another group crash. He may expect highest grade in project, OK!!. Hence the competition is between groups, make another groups code crash and enjoy the life, but of course best group would be the one whose code can't be crashed in any adverse cases. Please comply with the use of a few initialization and replacement functions. Please note that not using the functions will be treated as a deliberate attempt to disable the adversary code and will be graded as not addressing the assignment (zero credit).

```
ssize_t project_recvfrom (int s, void* buf, size_t len, int
flags, struct sockaddr* from, size_t* fromlen);

int project_poll (struct pollfd fds[], nfds_t nfds, int
timeout);
```

```
int project_pthread_create (pthread_t* thread, const
pthread_attr_t* attr, void* (*start_routine) (void*), void*
arg);

void project_init (int* argc, char*** argv);
```

Use the project_recvfrom, project_poll, and project_pthread_create routines just as you would use the corresponding C library calls. For function prototypes, include the header file project.h file into your client and server executables. project_recvfrom must be used to receive all file data and acknowledgements; recvfrom may be used directly only for accepting client request messages. The semantics of project_recvfrom differ slightly from recvfrom in that project_recvfrom will never return messages longer than 256 bytes. Such messages are dropped silently, implying that your RDP will need to fragment and reassemble the file. Data packets (and ACK's) may be dropped because of the server/client configuration file or other reasons as well; make sure that all members of different groups writing testing software should sit together and define similar interfaces so that one group's routines can be tested with another group's adversary code.

In both client and server, the main function must begin with a call to project_init to initialize the adversary code. Note that you must pass pointers to the argc and argv parameters to allow the routine to modify them.

We have full faith in you that if both members in the group work really hard,

1. Project is doable in the stipulated time
2. If you feel at any stage that you can't do this project in stipulated time, then please go to point 1

Note: This project, if done, could place you in line with students of TOP 10 US Universities because their students implement problems of similar difficulty level in their semester projects and I think that this reason alone is sufficient to put you on toes to complete this project.
Policy on Plagiarism

- All the projects will be checked by MOSS script.
- Any one resulting in 40% or more similarity will be graded F in the course, irrespective of being a donor or receiver.

Grading Criteria:

Your project will be graded out of 100 points, with the following criteria:

- Report (10)
- Correct Makefiles (5)
- Sensible and comprehensive Readme file (5)
- Errors free compilation and execution (5)
- Peer to Peer (20)
- Header Formats (5)
- Sliding Window Protocol (GBN/Selective repeat/Hybrid) (10)
- ACK vs ACK/NACK (5)
- Error control (corrupt packet, checksum, correction mechanism) (15)
- Testing layer(5)
- Timer management i.e. timeouts (10)
- Well written (good abstraction, error checking, and readability) and well commented code (5)