# Computer Science 467

**Computer Networks**

**Fall 2020**

**PEX 2 - 80 Points**

# Help Policy

|  |
| --- |
| **AUTHORIZED RESOURCES:**   Any, except another cadet’s/group’s program from this semester or past semesters’ offerings of this course.  **NOTES:**   * Never copy another person’s work and submit it as your own. * You must document all help received from sources other than your instructor, or instructor-provided course materials (including your textbook), or your partner. * DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others. |

# Documentation Policy

|  |
| --- |
| * You must document all help received from any source. **Include your documentation statement at the top of your programs**. Unless quoting directly or paraphrasing, you do not need to document your course text, lectures, or any other course materials provided by your instructor. * Each documentation statement must be specific enough that it explicitly describes **what** assistance was provided, **how** it was used in completing the assignment, and **who** provided the assistance. * If no help was received on this assignment, the documentation statement must state “NONE.” * If you checked answers/solutions with anyone, you must document with whom on which problems.  You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why. * Vague documentation statements must be corrected before the assignment will be graded, and will result in a 5% deduction on the assignment. |

# Learning Objectives:

* Demonstrate understanding of reliable data delivery by implementing a simplified version of TCP over a UDP socket.
* Function effectively as a member or leader of a team tasked with identifying solutions to computer network problems.
* Apply computer networking protocols and software development practices to produce client-server communication applications.

# Project Description

The goal of this project is to implement a reliable data-delivery protocol over UDP. Specifically, you will use your PEX 1 client (MP3 Streaming Client), and implement additional message headers and reliability functionality that resemble the TCP protocol.

You should implement this PEX using a C library. The reliable data functionality should be abstracted away from your MP3Client code, and as a result there should be **very few** changes to a working MP3Client from PEX1. For example, you will be replacing your ‘sendto’ and recvfrom’ functions in your MP3Client with ‘TCPSend’ and ‘TCPReceive’ that you implement in your library. **You are only required to have the ‘LIST\_REQUEST’ functionality from PEX1 working.** See additional details section for more information.

**The majority of this PEX will be graded on functionality.** The TCP\_MP3GraderServer.exe I provide will allow you to run in a ‘grader’ mode, and you will be able to test specific parts of the reliable data functionality. **You should develop your TCP library using these grader functions and implement one piece at a time.** You should have a good idea of what your grade will be when you turn in the assignment.

You will be provided a header file that will include the minimum functions that your library must export to your main program. However, I highly encourage you to decompose the problem further within the library.

**Your TCP header information must match EXACTLY the information provided in this assignment.** Failure to comply with the exact field names will result in severe point reductions due to a lack of functionality.

Included with the PEX 2 files is a Wireshark capture of what a correct communications stream looks like.

# Detailed Requirements

**Reliable Data Protocol Functionality**

* Your library must implement and export a ‘TCPConnect’ function that conducts a 3-way handshake with the server (see additional details).
  + Your client will immediately establish a connection to the server when it is started.
  + After each MP3Stream command is completed, your client will establish a new connection with the server. For example, after LIST\_REQUEST completes, a new connection (with new SEQ/ACK #s) will be immediately established before the START\_STREAM is sent.
* Your library must implement and export a TCPReceive function that replaces recvfrom in your MP3Client. It should receive a packet, parse, validate, and strip the TCP header, and return the underlying application data. If the packet contained application data, it must immediately send an ACK packet before returning data to the application.
* Your library must implement and export a TCPSend function that replaces sendto in your MP3Client. It should construct and prepend a valid header to the application data, send the packet, and wait for an acknowledgement from the server.
  + Flags, SEQ#, and ACK# must be updated based on the connection state, data sent, and data received.
  + TCPSend and TCPReceive must implement the sending and receiving rules articulated in the additional details.
* Protocol functionality must be implemented in a library (you should not have any TCP functionality in the original PEX 1 code).
* Client must correctly implement the LIST\_REQUEST functionality of PEX 1. This includes the client receiving and displaying the LIST\_REPLY response.

**Programming requirements**

* Your program must be written in C and can be compiled in VS Code using Cygwin (on Windows) or gcc (on Mac).
* You should not use any hardcoded paths (i.e., specifying a save directory that is unique to your computer). Instead, use relative paths.

# Suggestions / ROE

* It is perfectly legal to use the example code presented by your instructor in class. Get help if the example code is confusing.
* Wireshark will be an invaluable tool during this PEX. Using the ‘loopback adapter’ will allow you to observe traffic flowing between processes on your computer.

# Turn-in

You MUST submit all your code—this should include your client program, library, header file, and any supporting files/folders. Your instructor should be able to load this project file in VS Code, compile, and run it without any changes to your code.

# Additional Information

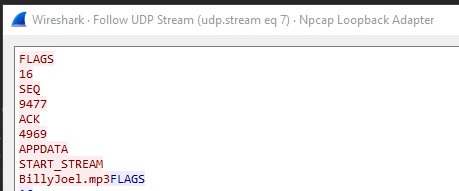
**Header Information.**

Every packet your client sends or receives will have a header prepended to the application data (the data the MP3 client/server are sending to each other). **This data is sent and received in ASCII.** The entire header can be viewed as a single string. Numbers are also converted into their ASCII representation before sending (use something like sprintf). While this if far from efficient, it makes it much easier to develop and troubleshoot your code when simply working with strings. The header data must follow the following format:

“FLAGS\n<flag number>\nSEQ\n<sequence number>\nACK\n<acknowledgement number>\nAPPDATA\n<application data>”

The <application data> is the exact information you used in PEX 1 and will start with something like “LIST\_REQUEST” and “START\_STREAM”.

Below is a screenshot of a Wireshark capture of a response from a LIST\_REQUEST command:

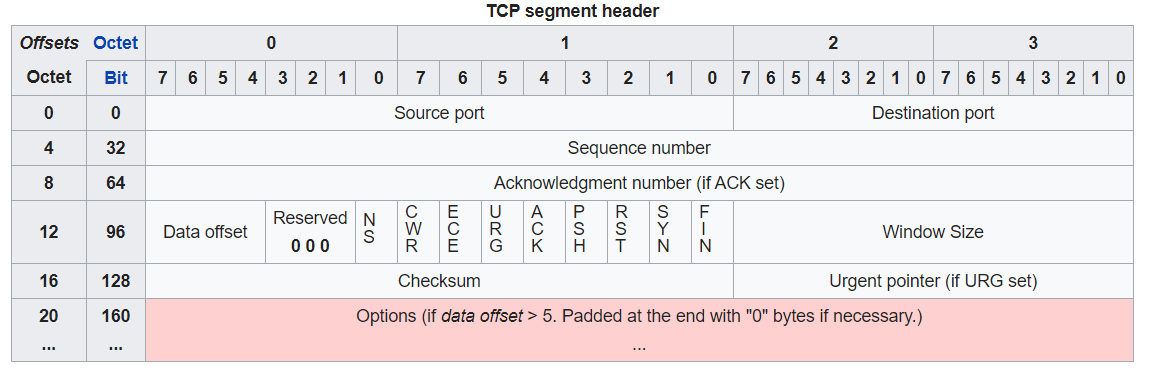


**Hint:** Use strtok() to parse the header based on newline characters.

We will now break down each part of this header.

**HEADER: FLAGS.**

The first field in the header provides flags for the packet. These flags identify what type of packet is being sent. There are only two flags you will use for this PEX: SYN and ACK. We will use the same structure that TCP uses.



A packet with only the SYN bit set would contain the bits 00000010. For this PEX, this flag is represented in the header as ASCII decimal ‘2’ and the header would look like “FLAGS\n2\n…”

ACK packets would be 16, and SYN+ACK packets would be 18.

Every packet will include an ACK flag after the first SYN packet is sent (see the TCPConnect details below).

**HEADER: SEQ.**

The sequence number, just like in TCP, tracks the amount of **application data** **sent**. It does NOT include TCP header data. It is helpful to think of the application data as one stream of data, and SEQ numbers identify where in the stream we are.

**SEQ numbers are calculated using a random starting number established during the TCPConnect handshake, added to the total number of application data bytes sent + 1**.

Example: Client’s initial sequence is 3000. Client sends the message “Hello World” with a sequence number of 3001 (initial sequence + 0 data + 1). The next packet that the client sends will have a sequence number of 3012 (initial sequence + 11 bytes of data + 1).

**HEADER: ACK.**

The acknowledgement number, just like in TCP, confirms what data has been received. Again, it only tracks application data, not the header data.

**ACK numbers are calculated using the remote application’s initial sequence number that was established during the TCPConnect handshake, added to the accumulation of the data received +1.** You can think of this as acknowledging the next expected byte of application data.

Example: Server’s initial sequence number is 5000. The client’s acknowledgement number will be 5001. Server sends “Hello World” with a sequence number of 5001. Client will acknowledge this message with an ACK number of 5012.

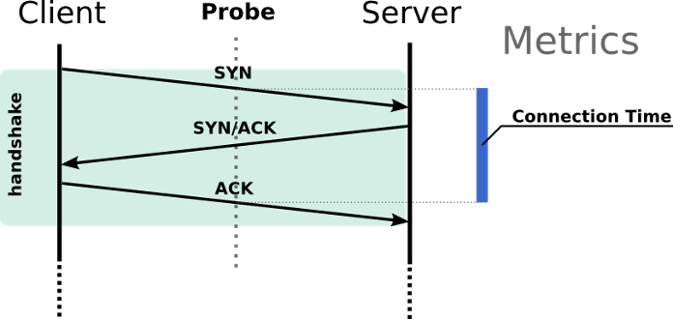
Make sure you are tracking the number of BYTES being sent and received (using strlen will not correctly count non-ascii data being sent).

**HEADER: APPDATA.**

This designates the start of the application data (LIST\_REQUEST, START\_STREAM, MP3 data, etc). This information should be passed to and from the library.

**TCPConnect.**

Your client must establish a connection to the server before sending application data. This is accomplished with a 3-way handshake.



1. Your client sends a SYN packet to the server (FLAGS = 2). Your initial sequence number is provided and should be a random number between 1 and 10,000. The ACK number should be 0 (we don’t know the server’s sequence number yet). There is no APPDATA (but the “APPDATA\n” field must still be included in the header.
2. Server will respond with a SYN+ACK packet. This includes the server’s initial sequence number, and an acknowledgement number (client’s SEQ + 1). Again, there will be no APPDATA.
3. Client will send an ACK packet. Your SEQ will be your initial sequence + 1. Your ACK number will be the server’s initial sequence + 1.

You are strongly encouraged to examine the attached Wireshark .pcap to see what this three-way handshake should look like. Packets 1 to 3 are the first 3-way handshake. Remember that after the LIST\_REQUEST is completed, another 3-way handshake occurs. Packets 8 to 10 are the second 3-way handshake.

**Sending Rules:**

1. Every packet sent must contain a valid header with FLAGS, SEQ, ACK, and APPDATA fields (again, see the .pcap file for what this looks like on the network).
2. Every packet after the first part of the 3-way handshake will contain the ACK bit checked in the flags. It will also provide the current SEQ and ACK numbers.
3. After sending a packet containing application data, your client will immediately wait for that data to be acknowledged from the server.
   1. Wait for 1s for a response (use socket timeouts).
   2. If an ACK is not received **OR has the wrong ACK number**, resend the packet and wait for an ACK again.
   3. Try repeat up to 3 times.

**Receiving Rules:**

1. After receiving a valid packet with application data, immediately send an ACK packet acknowledging the data. (These packets will have no data after the appdata field; packet 7 in the .pcap shows this ACK from the client.)
   1. Make sure you aren’t ACK’ing the server’s ACK packets
2. If your client receives a malformed packet (i.e., missing/incorrect header fields), it should ignore the packet.
3. If the client receives an unexpected sequence number (a duplicate packet, for example), immediately send an ACK packet telling the server what byte you are expecting.
4. If #2 or #3 occur, retry receiving the packet up to 3 times.

Remember, the application (MP3Client) expects all this to happen in the background (in the library). If the TCP protocol you implement detects an error or invalid/duplicate packet, it should try to handle using the rules above **before** returning execution to the application.

**Implementing as a Library.**

You have been provided a header file (tcp\_functions.h) that identifies what functionality your library must export. You will need to make a .c file (tcp\_functions.c) to implement these functions and include them in your source files.

There are only 3 functions and a structure that you should export. This corresponds to 3 key changes you should make in your MP3Client:

1. Call TCPConnect to establish your connection
2. Replace all instances of sendto with TCPSend
3. Replace all instances of recvfrom with TCPReceive

4. You may also need to pass the tcp\_info structure to any function that makes calls to   
 TCPSend and TCPReceive

You will also need to update your include statements. For reference, I added this line to my TCP\_MP3Client.c:

#include “tcp\_functions.h”

It’s very important you understand the **tcp\_info** struct that the header file exports. This keeps track of your current connection information and is used to build your packet headers as well as validate the data you’re receiving. The header file provides detailed comments, but the equations for the header fields would look something like this:

Outbound SEQ# or expected ACK# from server:

tcp\_info->my\_seq + tcp\_info->data\_sent + 1

Outbound ACK# or expected SEQ# from server:

tcp\_info->remote\_seq + tcp\_info->data\_received + 1

**This seems like a lot, where do I start?**

1. Your first task should be to make sure your LIST\_REQUEST part of PEX1 is working (see the rubric if you lost any points on it). It is the only part of PEX1 that is required to work for PEX2. This includes receiving and displaying the response from the server. Feel free to get EI to get this done quickly.
2. Use the provided server’s grader functionality and Wireshark to build your code from here.
3. Create your tcp\_functions.c file, and implement TCPConnect (the 3-way handshake). I suggest you create functions for building a packet header and parsing a packet header. This is 30+ points of the PEX.
4. Implement TCPSend and TCPReceive, using the tcp\_info struct, to add and parse headers for every packet sent and received.
5. Work on the TCP error scenarios provided in the grader server. Your client must still receive and display the contents of LIST\_REQUEST after errors occur. These are only worth 5pts each. Don’t get stuck trying to account for every possible scenario of multiple malformed packets, duplicate ACKs, etc. Build the minimum requirements of the rubric, get the grader test working, and move on. If your client has unhandled packets in the socket after displaying the list of songs, that’s okay.
6. Start early, get EI early.

**Names:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**CS467 – PEX 1 - Cut Sheet**

**80 Points**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Grading Criteria** | **Pts** | **Comments** |
|  | **Auto-Grader** | **65** |  |
| A | Client correctly conducts a 3-way handshake. | 20 |  |
| B | Client establishes new connection on completion of each MP3Stream command | 5 |  |
| C | Client calculates and sends correct SEQ #s | 10 |  |
| D | Client calculates and sends correct ACK #s (sends ACK every time appdata is received) | 10 |  |
| E | SENDING DATA:  If ACK not received, resends packet | 5 |  |
| F | SENDING DATA:  If wrong ACK# is received, resends packet | 5 |  |
| G | RECEIVING DATA:  If unexpected SEQ is received, resends ACK | 5 |  |
| H | RECEIVING DATA:  If malformed header received, ignore packet | 5 |  |
|  | **Code** | **15** |  |
|  | TCP Functionality implemented as library exporting the functions outlined in provided header | 10 |  |
|  | Code is well-organized and commented | 5 |  |
|  | **Total** | **80** |  |
|  |  |  |  |
|  | Poor or missing documentation (-5%) | -4 |  |
|  | Late Penalty Cap  (25% per 24 hour period) (60, 40, 20, 0) |  |  |
|  | **Final Grade** |  |  |