ICSI416/516 ICEN416 Fall 2021 -- Project 1

The `remapper`*: socket programming and reliable data transfer*

Due Friday, March 31st, 2022 at 11:59PM via Blackboard

ASSIGNMENT OVERVIEW

The goal of this project is to practice your Application and Transport Layer skills by implementing (i) the `remapper`, a client-server application that remaps characters in a user-specified file and (ii) two reliable data transport protocols. The remapper will function as follow: given a text file, it will replace each character in that text file with a letter that is N positions ahead in the alphabet. For example, if N=5 and we are looking to remap the letter "a" we will be replacing "a" with "f". The project will be completed in two phases. In the first phase, all students will implement two versions of the program: one that uses stock TCP[1] for reliable data transfer; and one that implements stop-and-wait reliability at the application layer and uses UDP for transport. In the second phase, graduate students will be asked to evaluate and compare the stock TCP and the stop-and-wait implementations using Wireshark. Undergraduate students can attempt the second phase of the assignment for extra credit.

Objectives: There are a number of objectives to this assignment. The first is to make sure you have some experience developing a network-based socket application. Second, because you are allowed to use any references you find on the Internet (including copies of existing code!), this assignment will help you see just how many network programming aids are available. Third, you will get a first-hand experience in comparative evaluation of protocol performance. Finally, having just a bit of practical experience will put a lot of the protocol concepts we learn into perspective.

Reading: Chapter 2.7 and 3.4 are the absolute minimum of reading you need to complete to confidently tackle this assignment. You can also use online resources on socket programming.

GRADING GUIDELINES

This assignment is to be completed individually. Any assignments flagged for group work will be considered for plagiarism. See my policy on cheating for more details.

Important NOTE on Formatting and Code Compliance:

Your code should run on our course VM `icsi416-sp22.its.albany.edu`. This is where your submissions will be automatically graded. You will lose points if your code does not comply with the assignment's formatting requirements and does not run on the VM.

Citing sources:

You may use code from the Internet to help you do this assignment (e.g. basic socket code). However, this is just like citing a passage from a book, so if you copy code, you must cite it. To

[1] I.e. the TCP protocol implemented in the operating system.

do this, put a comment in the beginning of your code that explains exactly what you have copied, who originally wrote it, and where it came from.

The breakdown of points for this assignment is below. In addition to correctness, part of the points count towards how well code is written and documented. Good code/documentation does not imply that more is better. The goal is to be efficient, elegant and succinct!

Item	**Points**	**Undergrads**	**Grads**
TCP implementation	40	Mandatory	Mandatory
Stop-and-wait implementation	50	Mandatory	Mandatory
Wireshark evaluation	10	Optional (Extra credit)	Mandatory
Report	20	Optional (Extra credit)	Mandatory
Code documentation	10	Mandatory	Mandatory

Procrastination Warning:
This is an assignment you definitely want to start on early. The design of the assignment is such that it is nearly impossible to provide all of the details you need. Instead of assuming things should be done a particular way, **ask questions!** Use every opportunity to meet with the instructor and TA and send them emails with questions. Answers that are relevant to everyone in class will be posted on the Blackboard discussion forum.

Language choice and file naming conventions:
You can choose to program your application in Python or Java. Pay attention to the file naming directions carefully. Make sure that you name your files and format your messages as specified in the assignment. An automated program will be used for grading and if there are any deviations, you will lose points!

Item	File naming in Python	File naming in java
Client using TCP	`client_tcp.py`	`client_tcp.java`
Server using TCP	`server_tcp.py`	`server_tcp.java`
Client using UDP	`client_udp.py`	`client_udp.java`
Sever using UDP	`server_udp.py`	`server_udp.java`

User prompt conventions:
This assignment specification gives you several examples that illustrate the user prompts and verbal feedback that your code should support. Please, study these examples closely and implement your programs to follow the same prompt conventions. Your code will be graded automatically, and we will be looking for these prompts in our auto-grading scripts. You will lose points if your prompts do not follow our examples.

Assignment Turn-in:
You must submit your source code, so we can assess your implementation. The assignment should be submitted using the course Blackboard. Because Blackboard only allows one file to be submitted, you should use zip to combine all your files into a single archive of the format <lastname_firstname>.zip and submit this archive for grading.

Undergraduate students must submit 4 files[2]:

- TCP server and client programs (2 files)
- Stop-and-wait (UDP) server and client programs (2 files)

Graduate students need to submit a total of 13 files as follows:

- TCP server and client programs (2 files)
- Stop-and-wait (UDP) server and client programs (2 files)
- Report (1 file).
- Your pcap traces from Wireshark (8 files).

Cheating Policy:
Cheating is not tolerated. Please, read the syllabus for my policies on cheating. Students caught cheating will receive 0 points for the assignment and will be reported. Of particular relevance to this assignment is the need to **properly cite material you have used** and **work individually**. Failure to do so constitutes plagiarism.

ASSIGNMENT DETAILS:

Phase 1: Implementing the `remapper`

The Application:
Your application will use a client-server architecture to provide a character remapping service. You will need to implement the remapper function from scratch[3].

- *Remapping functionality:* The application will allow a user to upload a text file of arbitrary size to the server. The file will then be loaded, read, and each character will be remapped. The redacted text will be stored in a new file. The remapper function will replace each character in the text file with a corresponding character that is N positions ahead in the English alphabet. For example, if the target string is "***clock***", and *N=5* the remapped string will be "***hqthp***". Once the server is done remapping, it will issue a message to the user indicating the output filename. The server will then allow the client

[2] Unless undergrads have attempted the extra credit components, in which case undergrads should submit 13 files, as detailed in the turn-in instructions for graduate students.
[3] If your code relies on external libraries your implementation will fail the grading on our server, which will not be running custom libraries, and you will lose points.

to download the output file. We will implement a limited remapping function that will only work on the lower-case letters of the English alphabet (no uppercase letters, numbers or special characters will have to be remapped).

- *Supported commands:* Your program should allow a user to upload and download text files, specify the remapping offset N, and quit the program. To this end, you will implement the following commands that the client can send to the server:
  - Copy a file from the client to the server (`put`), which takes as an input argument the full path to a file <`file`> on the client. Example execution:
    `put <file>`
  - Copy a file from a server to a client (`get`), which also takes as an argument the full path to a file <`file`> on the server. Example execution:
    `get <file>`
  - Remap command that will allow the user to specify a remapping offset N and a target file in which to perform the remapping. Example execution:
    `remap <int> <file>`
  - Quit the program per user request
    `quit`

Starting the client; accepting user-specified commands:

```
icsi416-sp22% client_udp.py <server_IP> <port>
              Enter command: put test.txt
              Awaiting server response.
              Server response: File uploaded.
              Enter command: remap 5 test.txt
              Server response: File test.txt remapped. Output file is
      test_remap.txt
              Enter command: get test_remap.txt
              File test_remap.txt downloaded.
              Enter command: quit
              Exiting program!
icsi416-sp22%
```

Your application should behave the same way with both stock TCP and stop-and-wait reliability.

The Transport:
You will practice your understanding of the Transport Layer by implementing two versions of reliable data transport: one that uses stock TCP (i.e. the TCP implementation that comes with your operating system) and another one that implements stop-and-wait reliability at the application layer and uses UDP for transport.

- *Reliability over stock TCP*: This version of your program will use stock TCP to implement reliable transmission of the text file. Below are examples of how to start the server and the client program, which will give you a sense of required input arguments and formatting[4].

[4] Input arguments are specified with <>. An actual run might look like: `client_tcp.py 169.226.65.98 2222`

Starting the server:
icsi416-sp22% server_tcp.py <port>

Starting the client:
icsi416-sp22% client_tcp.py <server_IP> <port>
Enter command:

- *Stop-and-wait reliability over UDP*: This version of your project will implement the text file exchange using stop-and-wait reliability over UDP. As a reminder, UDP provides best-effort packet delivery service; you will have to implement reliability checks on top of UDP to ensure that your data is successfully transmitted between the server and the client. Since we are working at the application level, we will implement our stop and wait reliability at the level of message chunks. We will discuss this functionality in terms of sender and receiver. Note that depending on whether you are performing `get` or `put` your sender and receiver will switch places in the client-server architecture (i.e. for `get`, your sender will be the server and your receiver will be the client, whereas for `put`, the sender will be the client whereas the receiver will be the server). The reliable data transfer should function identically for both `get` and `put`. Your reliable protocol will function as follows:
    - First, the sender calculates the amount of data to be transmitted and sends a "length" message to the receiver, letting them know how many bytes of data to expect. The length message should contain the string `LEN:Bytes`.
    - Second, the sender splits the data into equal chunks of 1000 bytes each, and proceeds to send the data one chunk at a time. Note that the last chunk might be smaller than 1000 Bytes and that is OK. Your programs should be able to handle arbitrary text file sizes. After transmitting each chunk, the sender stops and waits for an acknowledgement from the receiver. To this end, the receiver has to craft and send a special message containing the string `ACK`.
    - Finally, once the receiver receives all expected bytes (as per the `LEN` message), the receiver will craft a special message containing the string `FIN`. This message will trigger connection termination.
    - Timeouts. Note that there are a few points in the sender-receiver interaction where a timeout might occur. The below description specifies how your program should behave in a timeout.
        - Timeout after `LEN` message. If no data arrives at the receiver within one second from the reception of a `LEN` message, the receiver program should terminate, displaying "*`Did not receive data. Terminating.`*"
        - Timeout after a data packet. If no `ACK` is received by the sender within one second from transmitting a data packet, the sender will terminate, displaying "*`Did not receive ACK. Terminating.`*"
        - Timeout after `ACK`. If no data is received by the receiver within one second of issuing an `ACK`, the receiver will terminate, displaying "*`Data transmission terminated prematurely.`*".

You will want to test your system with large enough files to confirm that it works correctly. Ideally, you should test with the test files provided with this assignment: `file1.txt`, `file2.txt`, `file3.txt` and `file4.txt`.

Below are a few example executions of the server and the client program, which will give you a sense of required input arguments and formatting. Note that the user interface is identical between the TCP and the stop-and-wait programs, so you can reuse it. You have to devise the full list of client-server interactions for the `get` and `put` commands based on the description of stop-and-wait reliability over UDP above.

Starting the server:

```
icsi416-sp22% server_udp.py <port>
```

Starting the client; accepting user-specified command:

```
icsi416-sp22% client_udp.py <server_IP> <port>
              Enter command:
```

Phase 2: Evaluating the remapper using Wireshark

This phase is mandatory for graduate students and optional for undergraduate students. Undergrads who work on the evaluation phase will receive up to 30 points extra credit.

In this phase you will perform a comparative evaluation of your implementations in terms of overall delay and achieved throughput. We define overall delay as the relative time difference between the last and the first packet exchanged within a single program invocation. We define the achieved throughput as the total sum of bits exchanged within a single program invocation divided by the overall delay for that invocation. You will run your server and client implementations on different physical machines in order to account for a realistic Internet scenario. Specifically, you will run your server program on our course VM (`icsi416-sp22.its.albany.edu`) and your client program on your personal computer. You will also run Wireshark on your personal computer to be able to record a packet trace for each program invocation. You will need to record four packet traces for each of the TCP and UDP implementation (so eight altogether) for the provided with this assignment (`file1.txt`, `file2.txt`, `file3.txt` and `file4.txt`). Once you have collected the Wireshark traces, you need to process them offline and determine the overall delay and achieved throughput for each invocation. You need to fill out the results in the tables below.

Delay	File 1 (16KB)	File 2 (32KB)	File 3 (48KB)	File 4 (62KB)
TCP (sec)				
UDP (sec)				

Throughput	File 1 (16KB)	File 2 (32KB)	File 3 (48KB)	File 4 (62KB)
TCP (bps)				
UDP (bps)				

You will use your '`get <FILE>`' command implementation in order to transmit remapped files from the server to the client and capture Wireshark Traces in this evaluation phase. The four files against which you should evaluate are provided as a part of this assignment.

Note: You should run your experiments from the same network. For example, if you run your UDP experiments from campus and your TCP experiments from home, different delay characteristics of the campus and your home network will skew your results. In addition, when calculating your throughput and latency, you should consider only the part of the Wireshark trace that captures your active file transmission; not the duration of the entire Wireshark file.

Preparing your report:
You need to submit a brief report (not more than 4 pages) on your evaluation and findings. Your report should also include:

- *Your name and email address.*
- *A description of your methodology. How did you process the Wireshark traces to calculate the above metrics? Did you use a program, or did you do it manually?*
- *Two tables, using the same format as the ones above, with filled out values for overall delay and achieved throughput, calculated in your Wireshark analysis.*
- *A description of the trends you see in your results along with a justification of these trends.*