ICSI/ICEN 416/516 Fall 2023 -- Project 1
The cache: socket programming and reliable data transfer

Due Wednesday, November 1st, 2023, at 11:59PM via Brightspace

ASSIGNMENT OVERVIEW

The goal of this project is to practice your Application and Transport Layer skills by implementing (i) a cache service for file exchange, and (ii) two reliable data transport protocols. 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 (SNW) 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 network applications that use sockets. Second, because you are allowed to use any references you find on the Internet, 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: Chapters 2.2.5, 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 and ACADEMIC INTEGRITY GUIDELINES:

This assignment is to be completed individually. Any assignments deemed as group work will be flagged for plagiarism. We use automated tools to flag submissions with high similarity. Thus, to avoid plagiarism flags, you are highly advised not to copy code from the Internet or actively code together. You are encouraged to discuss your conceptual approach with fellow students; however, the project implementation should be individual original work. See my policy on cheating and the slide deck from Lecture 1 for more details on what constitutes plagiarism and how we perform plagiarism checks.

Important note on Formatting and Code Compliance:
You are advised to develop and debug your code on your personal machine. However, **you should note that your code will be automatically graded on our course virtual machine (VM).** Thus, it is highly advisable that you run your implementation on the VM to ensure that it works correctly before submitting. You will lose points if your code does not comply with the assignment's formatting requirements and does not run on the VM. The VM is being finalized and I will email with details on how to access it.

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

The breakdown of points for this assignment is below. In addition to correctness, part of the points count towards how well your 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 TAs and post questions on Piazza. Answers that are relevant to everyone in class will be posted on the Piazza 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	`client.py`	`client.java`
Server	`server.py`	`server.java`
Cache	`cache.py`	`cache.java`
Stock TCP transport	`tcp_transport.py`	`tcp_transport.java`
Stop-and-wait over UDP	`snw_transport.py`	`snw_transport.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 Brightspace. 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 5 files[2]:

- Server implementation (1 file)
- Client implementation (1 file)
- Proxy/cache implementation (1 file)
- TCP transport implementation (1 file)
- Stop-and-wait over UDP transport implementation (1 file)

Graduate students need to submit a total of 14 files as follows:

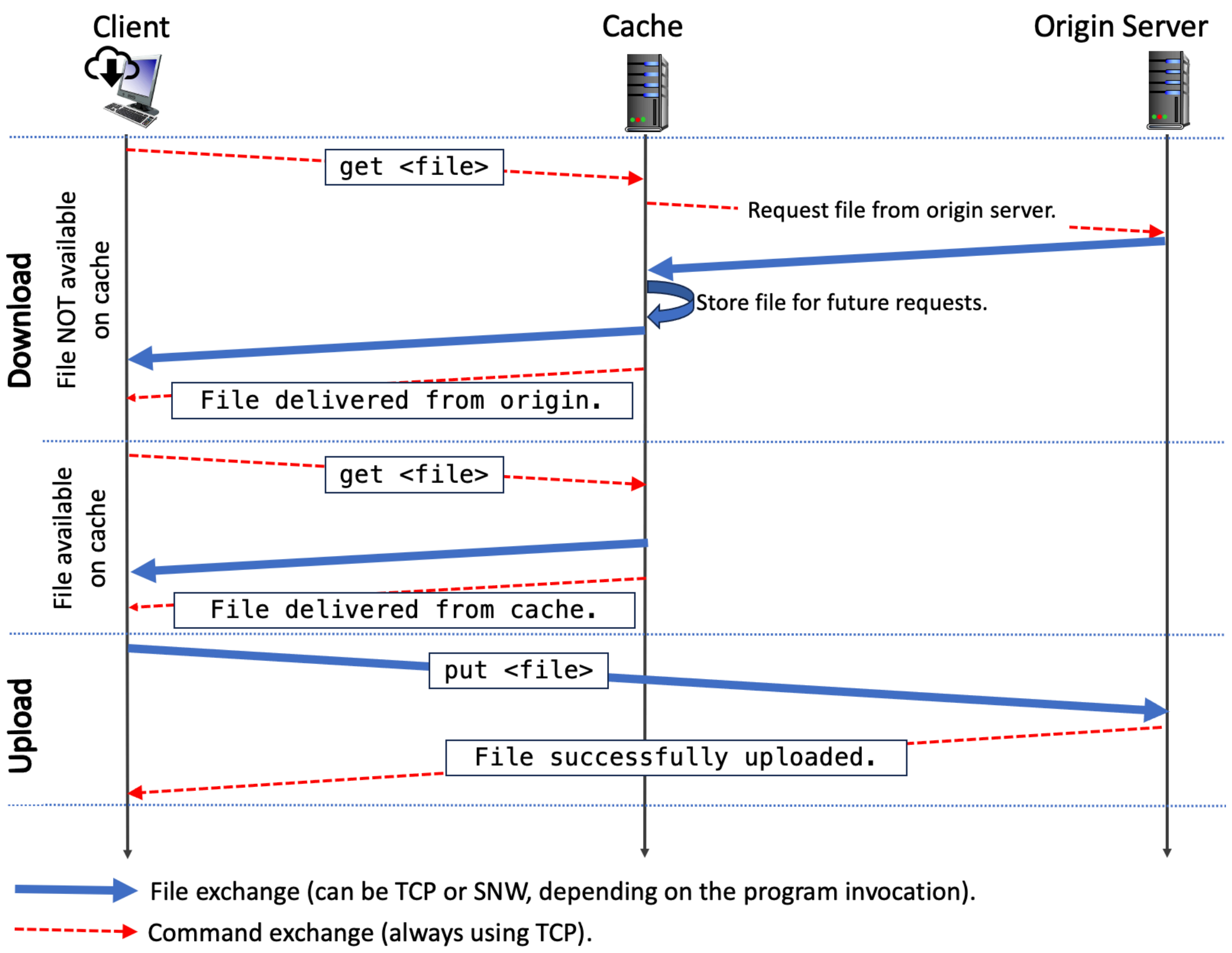
- Server implementation (1 file)
- Client implementation (1 file)
- Proxy/cache implementation (1 file)
- TCP transport implementation (1 file)
- Stop-and-wait over UDP transport implementation (1 file)
- Report (1 file).
- Your pcap traces from Wireshark (8 files).

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 might have used from online tutorials** and **work individually**. Failure to do so constitutes plagiarism.

ASSIGNMENT DETAILS:

The Application:
Your application will support file upload and download using a cache. There will be three components to your program: a client, a server, and a cache. All exchanges (upload or download) will be initiated by the client. Each attempt to download a file will first check with a cache to see if the file is available locally, and only if a target file is unavailable locally, it will be requested from the server. Each attempt to upload a file will store that file directly onto the server, not including the cache in the communication process. Figure 1 describes the expected interactions for file download and upload.

[2] Unless undergrads have attempted the extra credit components, in which case undergrads should submit 14 files, as detailed in the grad turn-in instructions below.



Client
Cache
Origin Server
Download
File NOT available on cache
get <file>
Request file from origin server.
Store file for future requests.
File delivered from origin.
File available on cache
get <file>
File delivered from cache.
Upload
put <file>
File successfully uploaded.
File exchange (can be TCP or SNW, depending on the program invocation).
Command exchange (always using TCP).

Figure 1: Illustration of interactions between the client, cache, and server programs.

- *Supported commands:* Your program should allow a user to upload and download files, and to quit the program. To this end, you will implement the following commands:
  - Upload: Copy a file from the client to the server using the `put` command, which takes as an input argument the full path to a file <`file`> on the client. Upon a successful receipt of a file, the origin server would send back "`File successfully uploaded.`" message and close the connection. This functionality should follow the Upload interactions illustrated in Figure 1. Example execution with prompts on the client:
    ```
    put <file>
    File successfully uploaded.
    ```
  - Download: Copy a file from the server to the client using the `get` command, which also takes as an argument the full path to a file <`file`> on the server. This functionality should follow the Download interactions illustrated in Figure 1. Specifically, the client should first connect to the cache to download the file locally. Upon successful receipt of the file from the cache, the client would close the connection and display "`File delivered from cache.`". Note that this message should come from the cache, rather than being generated locally on the

client. If the cache determines that the file is unavailable locally, the cache will request the file from the origin server on behalf of the client, store the file locally for further requests and deliver the file to the client. A control message sent from the cache to the client stating “`File delivered from origin.`” must be displayed to the user. Example execution with prompts on the client:

  - File available on cache:

```
get <file>
File delivered from cache.
```

  - File NOT available on cache:

```
get <file>
File delivered from origin.
```

- Quit the program per user request.

```
quit
```

- *Commandline inputs.* When starting your client, server, and cache, you will need to specify several commandline inputs, as detailed below:
  - Client – the client will take as commandline inputs (1) the server IP, (2) the server port, (3) the cache IP, (4) the cache port and (5) the transport protocol to be used. Input arguments should follow that specific order. Some example executions of your client are:

```
client.py localhost 10000 localhost 20000 tcp
    Enter command:

client.py localhost 10000 localhost 20000 snw
    Enter command:
```

  - Server – the server will take as commandline inputs (1) a port on which to run and (2) a transport protocol to be used, in that specific order. Sample runs of the server include:

```
server.py 10000 tcp
server.py 10000 snw
```

  - Cache – the cache will take as commandline inputs (1) a port number on which to run, (2) server IP, (3) server port, (4) transport protocol to be used, in that specific order. Sample runs of the cache include:

```
cache.py 20000 localhost 10000 tcp
cache.py 20000 localhost 10000 snw
```

All command exchanges (i.e. interactions designated with red dashed arrows in Figure 1) must be performed using TCP. File exchanges (designated with blue solid arrows in Figure 1) will use either TCP or stop-and-wait reliability depending on the invocation.

In what follows, we provide several examples of user interactions with the client program. Your submissions should follow the prompts exactly, or else, your program will fail our automatic grading.

Starting the client; accepting user-specified commands[3]:

```
icsi416-fa23% client.py localhost 10000 localhost 20000 tcp
              Enter command: put test.txt
              Awaiting server response.
              Server response: File successfully uploaded.
              Enter command: get test.txt
              Server response: File delivered from origin.
              Enter command: quit
              Exiting program!
icsi416-fa23%
```

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. You will implement these two as packages (`tcp_transport` and `snw_transport`) and will use them in your client, server and cache code, depending on the transport protocol specified by the user upon program execution. The Application specification above already gives you examples of how the user would start the server, client and cache while specifying a transport implementation to be used. You will use **`tcp`** as a commandline argument for TCP transport and **`snw`** as a commanline argument for stop-and-wait 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].
- *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 sender and the receiver. 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). Furthermore, depending on whether the server or the cache is delivering a file, the sender will change accordingly. The reliable data transfer should function identically for both `get` and `put`. Your reliable protocol will function as follows:

[3] Note that upon starting the client, your server and cache programs should already be running. Also note that if testing on your local machine, it is advisable that you use separate folders for the client, server and cache, not to overwrite the files that are being exchanged.

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

- 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 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 received 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`.

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.

Phase 2: Wireshark Evaluation:
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 transport layer 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 use your `get <file>` command implementation to download a file from the server (note that you should ensure the file is not cached). You will run your server and client implementations on different physical machines to account for a realistic Internet scenario. Specifically, you will run your server program on our course VM 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 SNW implementations (so eight altogether) for the files 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)				
SNW (sec)				

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

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.

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.*