Assignment 2: Writing Networked Applications

COL334/672 : Computer Networks, Diwali'25

Deadline: 8 September, 2025

Goal: The goal of this assignment is to familiarize you with how to write networked applications. In networks, sockets are the abstractions available to the application developer. Using socket programming, you will implement a basic client-server program along with a few scheduling algorithms.

The assignment is divided into four parts:

- Parts 1 and 2: TCP socket programming
- Parts 3 and 4: Scheduling algorithms. These will be released a week later.

Hints are provided throughout. If you still have questions, start a discussion on Piazza. *This assignment should be done in pairs.*

1 Word Counting Client

You will implement a client-server communication using TCP sockets. The client program downloads a file containing a list of words from the server and counts word frequencies. Here is a sketch of the file-reading protocol:

- The client establishes a TCP connection with the server, which is listening on a pre-defined IP address and port (from `config.json`) file. Once established, the connection remains persistent unless the client closes it.
- The server maintains a local file `words.txt` containing a comma-separated list of words. A sample file is provided.
- The client requests words by sending a message:

  `p,k\n`

  where

  - p = *offset*, the starting position, zero-indexed,
  - k = number of words to read.

  Example:

  `E.g., 10,5\n`

- The server responds with k words starting at offset p. If the end of the file is reached, it sends a special token, `EOF`. For simplicity, assume that this token does not appear in the file. For instance, if k=5, the server will respond:

```
word1,word2,word3,word4,word5\n
```

If there are only 4 words remaining, the server will respond:

```
word1,word2,word3,word4,EOF\n
```

If the offset is greater than the number of words, the server responds:

```
EOF\n
```

- After receiving words, the client closes the TCP connection and prints the word frequencies, one per line. Example:

```
bat, 1
cat, 2
```

- Ensure your code handles edge cases such as insufficient words, invalid offsets as described above.

1.1 Analysis

You will now test your code on Mininet, a lightweight network emulator that lets us create virtual networks of hosts, switches, and links on a single machine for rapid testing and prototyping. The platform will be used for later assignments as well, so please spend some time in understanding it. We have provided a network topology file that you should use for the analysis. It consists of two hosts (one client, one server) connected via a switch.

Experiment: Vary the value of k (the number of words per request) for the file download and log the completion time. Run the experiment 5 times for each value of k and compute the average and 95% confidence intervals. Plot the completion time (y-axis) along with confidence intervals vs k. We have provided a sample script called `runner.py`. Explain your observations. *Tip: For the analysis runs, calculate the average completion time for each value of k and store it in a file (e.g., CSV) for plotting rather than outputting it on terminal.*

1.2 Guidelines

You should write this part of the assignment using C++ (to expose you to low-level socket APIs). Later parts (including Part 2) will use Python as certain things such as multi-threading are easier in Python. You should also contrast the socket APIs between these two languages. The plotting code can be written in Python or any language of your choice. A sample plotting script is shared.

What to submit: You should submit the code in a separate folder called `part1` containing `client.cpp`, `server.cpp`, and `Makefile`. In addition, assume a common `config.json` file for both server and client containing parameters namely, server IP, server port, k, p (starting offset), filename (the word file name), num_repetitions. The Makefile should support the following:

- `make build`: compiles the server and client code
- `make run`: runs a single iteration of client-server code
- `make plot`: runs for varying `k`, `num_repetitions` times and generates `p1_plot.png`

2 Concurrent Word Counting Clients

In this part, you need to extend your server to handle multiple concurrent client connections. Note that the server is listening on the same port (you should appreciate the multiplexing allowed using TCP ports). The server processes requests one at a time (on a first come-first serve basis), while queuing the others.

2.1 Analysis

Run the word counter on mininet with different number of concurrent clients ranging from 1 to 32 (incremented by 4). For each value of *num_clients*, run the experiment 5 times. Record the average completion time per client. Plot average completion time per client (y-axis) vs number of clients (x-axis), including confidence intervals. Discuss your observations. In particular, does the completion time per client remain constant, or does it increase with more clients?

2.2 Submission

Submit a folder named `part2` containing `client.py`, `server.py`, and `Makefile`. Assume a `config.json` file containing parameters as Part 1 and an additional parameter, `num_clients` indicating the number of concurrent clients. Your Makefile should support the following:

- `Make run`: runs one experiment with the given configuration (1 server and `num_clients` clients)
- `Make plot`: runs experiments for varying `num_clients`, repeating each `num_repetitions` times, and generates `p2_plot.png`

3 When a Client Gets Greedy

So far, we have focused on socket programming. In the next two parts, we will explore a couple of basic scheduling policies and their motivation.

Consider the setup from Part 2 with one server and n clients. The server maintains multiple concurrent active connections but processes requests sequentially using a centralized scheduling algorithm. You previously implemented an FCFS scheduling policy, i.e., processing requests in order of arrival. While simple, FCFS can lead to fairness issues. In particular, a greedy client can monopolize the server by sending c requests back-to-back instead of sending one request and waiting for its response. In this part, you will implement such a greedy client along with an FCFS server.

3.1 Analysis

Emulate a scenario with 10 clients, one of which is greedy and sends c requests back-to-back, each trying to download the word file. Assume k is 5, i.e., each client (including the greedy client) is requesting 5 words in a single request. Log the time to download the file for each client. We will quantify fairness using the Jain's Fairness Index (JFI) applied to completion times[1]. Analyze how JFI varies as c increases from 1 to 10, and plot JFI (y-axis) vs. c (x-axis). Discuss your results: how does fairness change as c grows, and what would happen if c were increased further?

3.2 Submission

Your implementation should run on Mininet. Submit a folder named `part3` containing `client.py`, `server.py`, `Makefile`. Assume a `config.json` file that contains all parameters from Part 2, plus c (the number of parallel requests issued by the greedy client).

[1] You should read about JFI online. It is quite useful in quantifying fairness

Your Makefile should support:

- `make run-fcfs`: runs one experiment with FCFS scheduling using the parameters in the config file.
- `make plot`: runs experiments for FCFS with varying values of c (as specified above) and generates `p3_plot.png`.

4 When the Server Enforces Fairness

To address the fairness issues observed in Part 3, the server can implement a round-robin scheduling policy. In this policy, the server cycles through clients in a fixed order, serving one request from each client before moving to the next. In this part, you will implement a round-robin scheduling server.

4.1 Analysis

Emulate the same scenario as in Part 3. Log the completion time for each client and compute the Jain Fairness Index (JFI). Analyze how JFI varies as c increases from 1 to 10, and plot JFI (y-axis) vs. c (x-axis). Compare your results with FCFS: how does round-robin affect fairness? Discuss any limitations of your approach – for example, can a client still monopolize the server under certain conditions?

4.2 Submission

Your implementation should run on Mininet. Submit a folder named `part4` containing `client.py`, `server.py`, `Makefile`. Assume a `config.json` similar to Part 3.

Your Makefile should support:

- `make run-rr`: runs one experiment with round robin scheduling using the parameters in the config file.
- `make plot`: runs experiments for round robin with varying values of c (as specified above) and generates `p4_plot.png`.