COL 334/672 Computer Networks
Assignment 2: Writing Networked Applications

Saharsh Laud — 2024MCS2002
Shuvam Chakraborty — 2024MCS2004

Deadline: September 8, 2025

Contents

1 Word Counting Client

1.1 Analysis

Problem Statement: 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.

1.2 Guidelines

Problem Statement: 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.

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

Answer:

We implemented a TCP client-server application in C++ where the client downloads words from a server in chunks of size k. Our experiments on Mininet varied k from 1 to 200 words per request over 5 runs each.

Key Observations:

- **Inverse relationship:** As k increased, completion time decreased significantly. For k=1, average completion time was 182ms, while for k=200 it dropped to 21ms.
- **Diminishing returns:** The improvement plateaus beyond k=50, suggesting optimal request sizing around 50-100 words.
- **Network efficiency:** Larger requests reduce the number of TCP round trips, minimizing protocol overhead and improving throughput.
- **Variability:** Standard deviations were generally small (1-5ms), indicating consistent network behavior in Mininet.

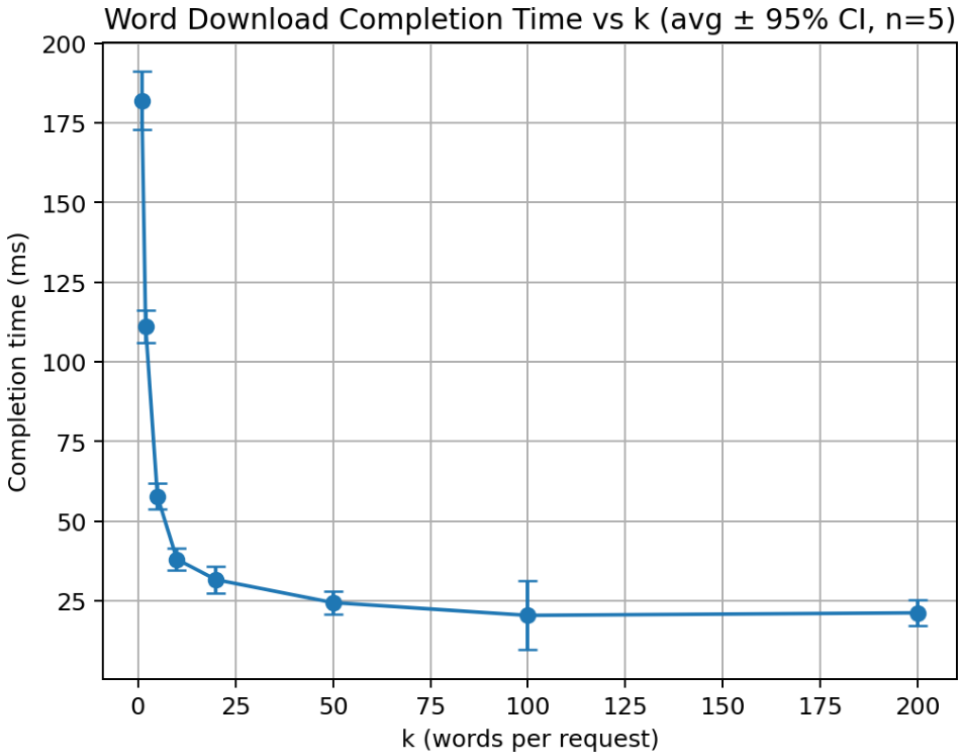

Figure 1: Completion time vs request size k showing inverse relationship

The C++ implementation provided fine-grained control over socket operations, allowing us to optimize buffer management and error handling. This low-level approach contrasts with Python's higher-level abstractions used in later parts.

2 Concurrent Word Counting Clients

2.1 Analysis

Problem Statement: 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

Problem Statement: 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

Answer:

We extended the server to handle multiple concurrent clients using Python, testing with 1, 5, 9, 13, 17, 21, 25, 29, and 32 clients. Each experiment ran 5 times to ensure statistical reliability.

Key Findings:

- **Linear scaling:** Average completion time per client increases approximately linearly with the number of clients, from 52ms (1 client) to 948ms (32 clients).
- **FCFS queueing:** The server processes requests sequentially, causing later-arriving clients to wait longer as queue depth increases.
- **Confidence intervals:** Variability increased with client count, particularly noticeable at 13 clients ($\pm$126ms), indicating occasional network jitter or scheduling delays.
- **Server bottleneck:** The single-threaded request processor becomes the limiting factor, demonstrating classic queueing theory in practice.

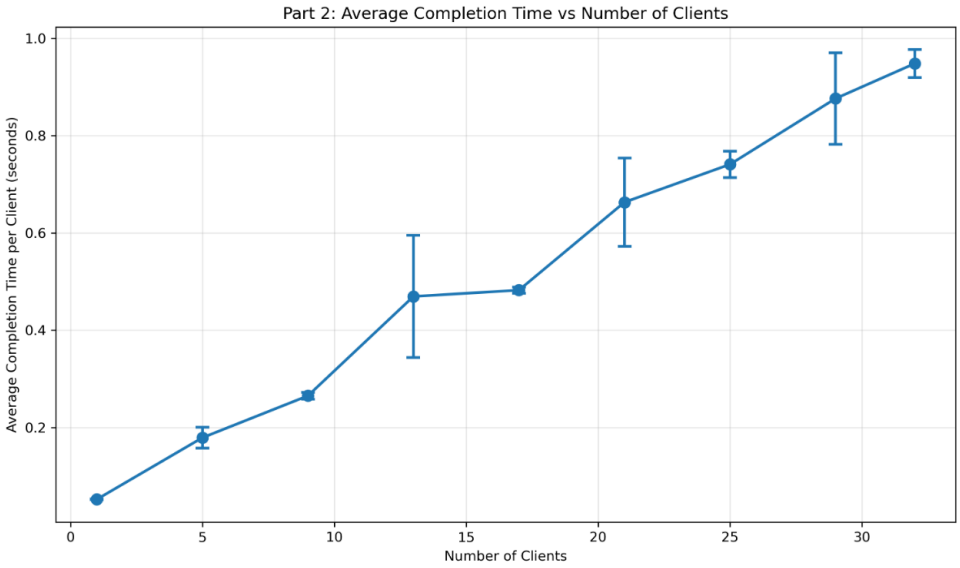

Figure 2: Average completion time per client vs number of concurrent clients

The results confirm that completion time does NOT remain constant but increases with more clients, validating our FCFS server implementation and highlighting scalability challenges.

3 When a Client Gets Greedy

3.1 Analysis

Problem Statement: 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. 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

Problem Statement: 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). 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.

Answer:

We implemented an FCFS server with 10 clients where one "greedy" client sends c requests back-to-back while others send single requests. We measured fairness using Jain's Fairness Index (JFI).

FCFS Scheduling Results:

- **Fairness degradation:** JFI decreases as c increases, dropping from 0.9999 (c=1) to 0.7672 (c=10) and further to 0.5342 (c=20).
- **Greedy advantage:** The greedy client completes downloads much faster (365ms at c=10) compared to normal clients (1200ms), demonstrating clear unfairness.
- **Monopolization effect:** Higher c values allow the greedy client to flood the server queue, starving normal clients of processing time.
- **Non-linear degradation:** Fairness loss accelerates beyond c=6, indicating a threshold where the greedy behavior becomes dominant.

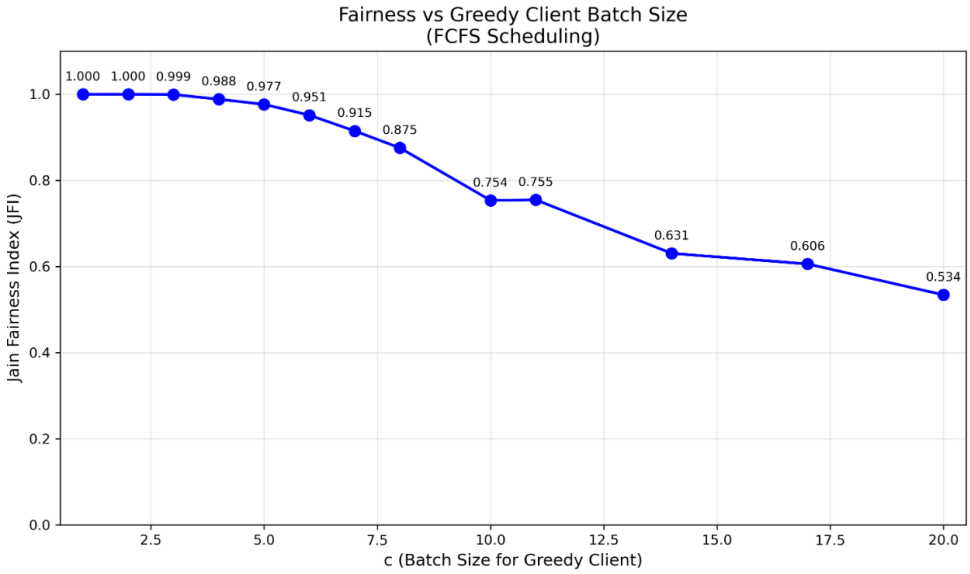

Figure 3: Jain's Fairness Index vs greedy client batch size (FCFS scheduling)

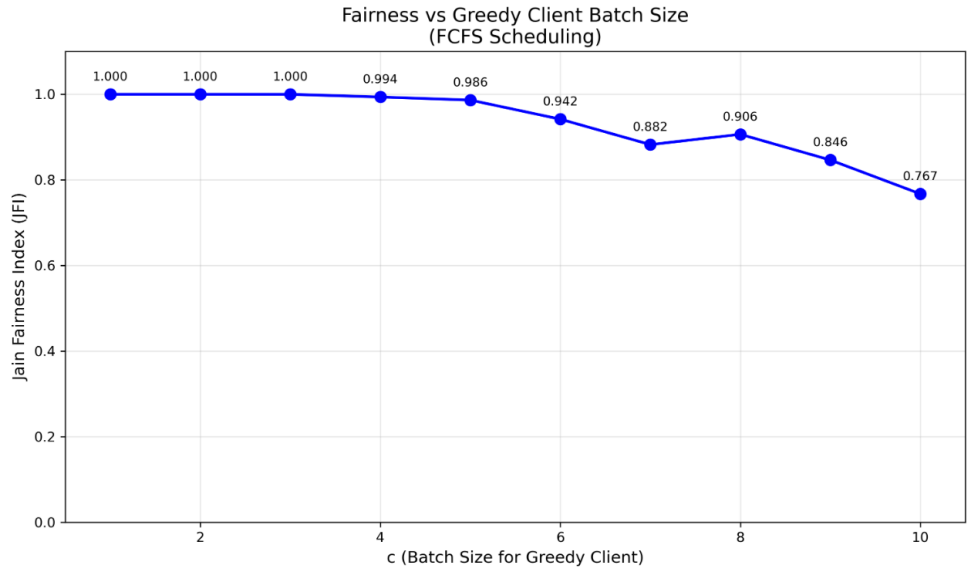

Figure 4: JFI vs greedy client batch size for c till 10

The JFI formula $JFI = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2}$ where x_i represents utility (inverse completion time), effectively captures the fairness degradation. As c increases further, we would expect JFI to approach even lower values, potentially causing complete starvation of normal clients.

4 When the Server Enforces Fairness

4.1 Analysis

Problem Statement: 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

Problem Statement: 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.

Answer:

We implemented a Round-Robin scheduler to address the fairness issues observed in Part 3, cycling through clients to serve one request from each before moving to the next.

Round-Robin Scheduling Results:

- **Fairness restoration:** JFI remains consistently high (0.9997-1.0000) across all c values, demonstrating effective fairness enforcement.
- **Greedy mitigation:** The greedy client's completion time (10.6-11.5 seconds) is now comparable to normal clients (11.1-11.6 seconds), eliminating unfair advantage.
- **Stable performance:** Unlike FCFS, completion times show minimal variation regardless of the greedy client's batch size.
- **Overhead trade-off:** All clients now take longer to complete (11 seconds vs 1.2 seconds in FCFS) due to round-robin scheduling overhead.

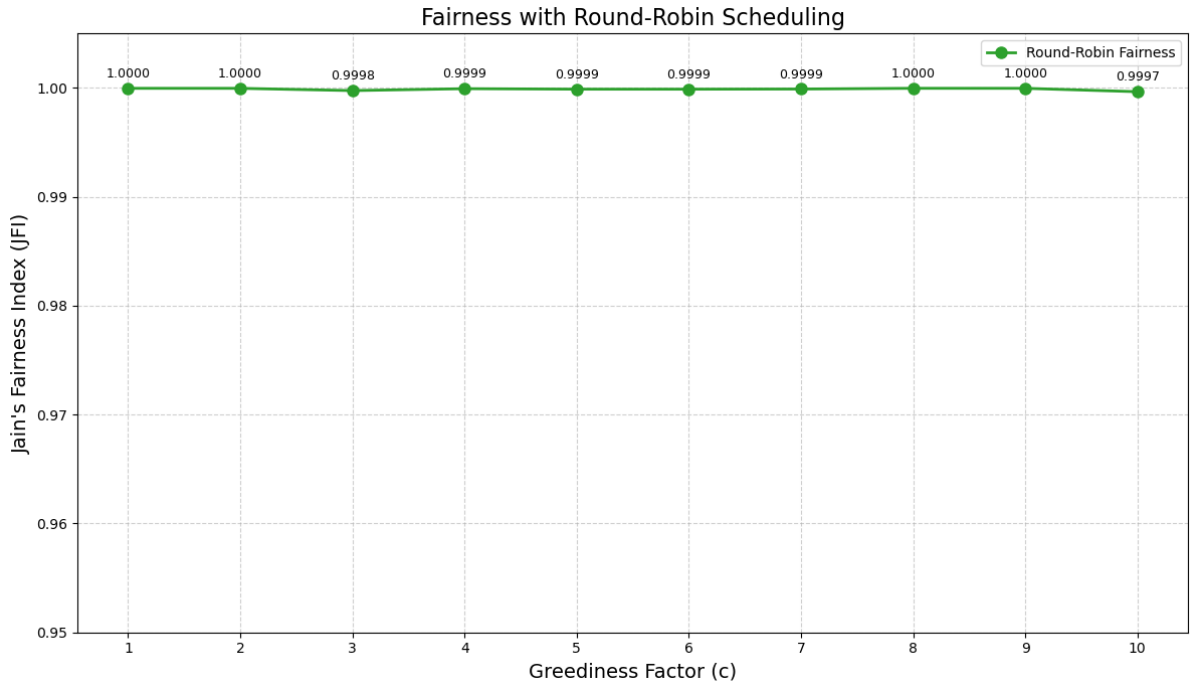

Figure 5: Jain's Fairness Index vs greedy client batch size (Round-Robin scheduling)

Comparison with FCFS: Round-Robin successfully prevents client monopolization by ensuring equal access to server resources. However, it comes with performance costs - all clients experience higher latency due to scheduling overhead.

Limitations: While Round-Robin improves fairness, a client could still attempt monopolization by opening multiple connections or sending larger individual requests. Additionally, the current implementation assumes equal priority for all clients, which may not suit all applications requiring differentiated service levels.

References

1. COL 334/672 Course Materials and Lecture Notes
2. Kurose, J.F. and Ross, K.W., "Computer Networking: A Top-Down Approach"
3. Stevens, W.R., "Unix Network Programming"
4. Piazza Discussions for COL 334/672 Assignment 2
5. Python and C++ Socket Programming Documentation
6. Mininet Documentation and Tutorials
7. Jain, R., "A Quantitative Measure of Fairness and Discrimination"
8. Perplexity. AI assistance for Round-Robin server and performance analysis. Available at: `https://www.perplexity.ai/search/i-have-to-work-on-this-assignm-b4DIDuVsQcep9J7QaOt4iw`
9. Perplexity. AI assistance for Assignment Part 2 experiments. Available at: `https://www.perplexity.ai/search/cn-assignment-part-2-i-am-work-h4yTEBfARuiOVWaFJPoNvw`
10. Perplexity. AI assistance for Assignment Parts 3 and 4. Available at: `https://www.perplexity.ai/search/cn-assignment-part-3-and-4-i-n-6M7H3sfmTC2UuS7CBzaIGg`
11. Perplexity. AI assistance for general guidance. Available at: `https://www.perplexity.ai/search/hi-8zR0SYkIQEmRERxRXwo5qA`