Assignment 2: Writing Networked Applications

COL334

Laksh Goel (2023CS10848)

Adit Jindal (2023CS50626)

IIT Delhi

September 9, 2025

Contents

Goal

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

1 Word Counting Client

1.1 Experiment

This experiment investigates the performance of a TCP-based client-server application designed to retrieve and count word frequencies from a server-hosted file. The client requests segments of words from the server using a custom protocol (p,k
n), where p is the offset and k is the number of words to retrieve. The server responds with the requested words or a special EOF token if the end of the file is reached.

The setup is deployed in a simulated network environment using Mininet, consisting of two hosts (a client and a server) connected via a switch. The primary focus of the experiment is to analyze how varying the chunk size k affects the total completion time of the file download and word counting process.

By systematically changing the value of k and recording the completion time across multiple runs, the experiment aims to determine the optimal request size and understand the trade-offs involved in batching data over a TCP connection. The results are used to compute average completion times and 95% confidence intervals, which are then visualized to support the analysis.

1.2 Implementation

1. client.cpp : This file controls the client. It sends a request (p and k) to the server, and waits for a response. It does this till the response contains EOF. Then it finds the count of each word in the received response.

2. server.cpp : This has the server logic. It establishes a connection with the client. Then it receives a request from the client, parses it, and sends a suitable response.

3. Runner file : It calls functions to create the network (mininet), and connects the client and server.

1.3 Observations and Analysis

We observe that as k increases, the average completion time drops significantly. This is expected, because as the packet size increases (for a constant total message length), the number of round trips required for the entire file transmission decreases. This reduces the effect of network latency, thus reducing the time per response.

2 Concurrent Word Counting Clients

2.1 Experiment

In the second part of the experiment, the server was extended to handle multiple concurrent client connections.

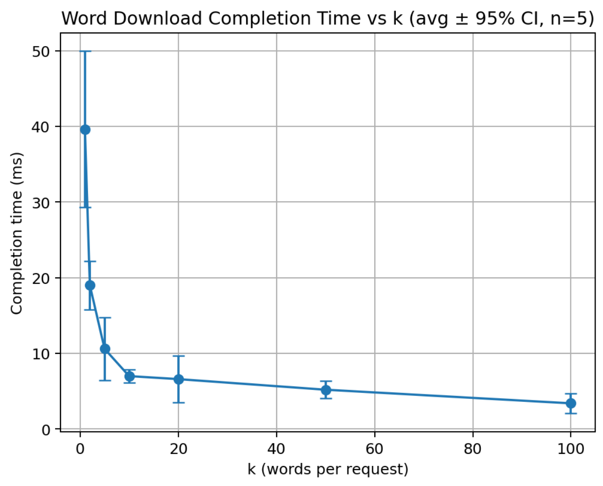

Figure 1: Single client

Server Behavior: The server continues to listen on a single predefined TCP port (as specified in config.json). When multiple clients attempt to connect, the server uses the accept() call to handle one client at a time in a first-come-first-serve (FCFS) manner. Each incoming connection is placed in a connection queue, and handled sequentially. This design demonstrates how a single-port TCP server can serve multiple clients without requiring multiple ports, leveraging TCP's use of (IP address, port number) tuples for each unique connection.

Concurrency Model: To manage multiple concurrent client connections without relying on multi-threading or multiprocessing, the server uses I/O multiplexing via Python's select module. This allows the server to monitor multiple socket descriptors simultaneously, handling I/O operations (read/write) only when the corresponding socket is ready. After binding and listening on the designated port, the server adds its listening socket to a list of monitored sockets. When select.select() is called, it blocks until one or more sockets become ready for reading. If the listening socket is ready, a new client connection is accepted. If an existing client socket is ready, it means a request has arrived and can be processed. This event-driven model ensures that the server can handle many clients concurrently using a single-threaded loop

2.2 Implementation

1. client.py : This file controls the client. It sends a request (p and k) to the server, and waits for a response. It does this till the response contains EOF. Then it finds the count of each word in the received response. It remains unchanged from the first part.

2. server.py : This has the server logic. It establishes a connection with the client. Then it receives a request from the client, parses it, and sends a suitable response. The change here is the use of SELECT (in python), which picks the read-ready sockets to create connections with or to process.

3. Runner file : It calls functions to create the network (mininet), and connects the client and server. It opens the different clients and the server.

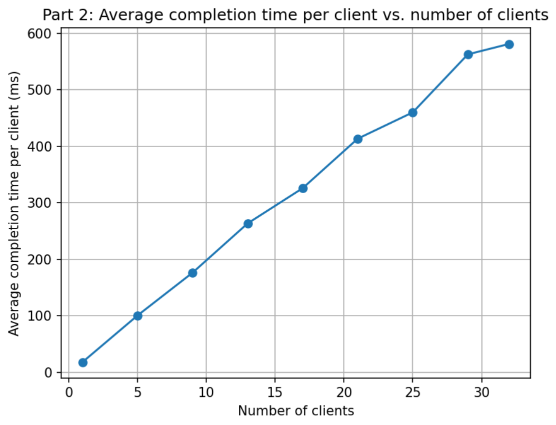

Figure 2: Multiple clients

2.3 Observations and Analysis

We observe that as the number of clients increases, the average completion time increases too. This is expected, because as the clients increase, the queuing delay gets added up, and is observed in the average time for each client.

3 When a Client Gets Greedy

3.1 Experiment

Although FCFS is a straightforward and fair policy under balanced loads, it does not handle client-side unfairness well. Specifically, it becomes vulnerable to greedy clients — those that continuously issue multiple back-to-back requests without waiting for responses. This can result in starvation of other well-behaved clients and poor overall fairness.

To explore this, a greedy client was implemented. Instead of sending one request and waiting for the server's response (as in a typical request-response model), the greedy client sends c consecutive requests (e.g., c = 5) without waiting in between. These requests are placed into the server's centralized request queue in rapid succession, occupying the front of the queue.

The server processes requests strictly in arrival order, regardless of the source. Since select() detects socket readiness but does not impose fairness across connections, all requests from the greedy client are likely queued before any requests from other clients are even noticed. As a result, the greedy client's requests are handled first.

As c is increased further, the entire file might get processed in a single request, givin gmaximum unfairness possible.

3.2 Implementation

1. client.py : This file controls the client. It sends c requests (p and k) to the server, and waits for the response. It does this till the response contains EOF. Then it finds the count of each word in the received response. Here, the difference introduced is that instead of sending a request and waiting for a response, the client sends c requests together and waits for their response collectively.

2. server.py : This has the server logic. It establishes a connection with the client. Then it receives a

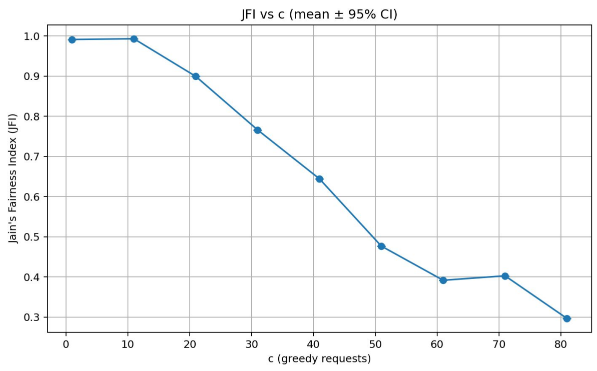

Figure 3: Multiple clients

request from the client, parses it, and sends a suitable response. It remains unchanged from part 2. The only (implementational) difference added is that to cater to the case when multiple requests arrive together (as one large string), the server is now able to separate them and process them separately.

3. Runner file : It calls functions to create the network (mininet), and connects the client and server. It opens the different clients and the server, opening one greedy client first.

3.3 Observations and Analysis

We observe that as the batch size increases, the fairness decreases. This is expected, because as c increases, the bias towards the greedy client increases, and the chances of its requests being picked from the queue get increased over the 'polite' clients. This results in the lower JFI values. This also shows that FCFS is clearly not fair in the presence of greedy clients.

4 When the Server Enforces Fairness

4.1 Experiment

To mitigate the fairness issues identified with the FCFS approach, the server was modified to implement a Round-Robin (RR) scheduling policy. Unlike FCFS, which serves requests in strict arrival order, RR ensures that each connected client gets an equal opportunity to have its requests processed, regardless of how many requests it sends or how aggressive it is.

In the RR model, the server maintains a list of all active client sockets. During each cycle, it iterates through this list in a fixed order and processes at most one request per client per round. This prevents any single client—greedy or otherwise—from monopolizing the server's processing time.

Round robin ensures fairness in this case, since the client can no longer have multiple requests processed together.

A client can still influence the fairness, by opening multiple connections with the server, thus occupying a larger fraction of the server's attention than fair. Also, it can repeatedly disconnect and connect, which can bring it to the front of the queue each time.

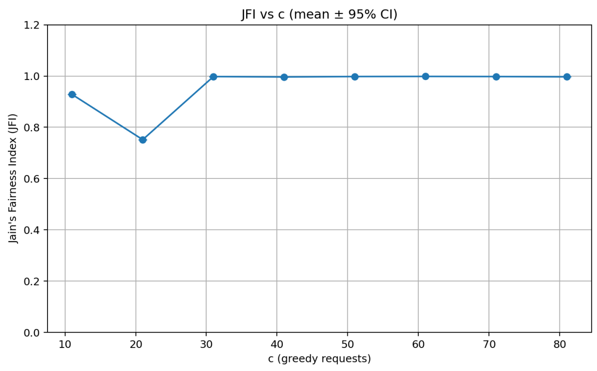

Figure 4: Multiple clients

4.2 Implementation

1. client.py : This file controls the client. It sends c requests (p and k) to the server, and waits for the response. It does this till the response contains EOF. Then it finds the count of each word in the received response. Instead of sending a request and waiting for a response, the client sends c requests together and waits for their response collectively. This remains unchanged from part 3.

2. server.py : This has the server logic. It establishes a connection with the client. Then it receives a request from the client, parses it, and sends a suitable response. The only (implementational) difference added is that to cater to the case when multiple requests arrive together (as one large string), the server is now able to separate them and process them separately.

3. Runner file : It calls functions to create the network (mininet), and connects the client and server. It opens the different clients and the server, opening one greedy client first.

4.3 Observations and Analysis

We observe that as the batch size increases, the fairness remains constant. This is expected, because even as c increases, the sever still picks only one requests from the greedy client, ignoring the rest. So the presence of more requests from the same client becomes immaterial in this case.