COL334 - Assignment 2 - Socket Programming

Nikhil Zawar 2022CS11106
Yash Bansal 2022CS51133

Contents

1 Word Counting Client **1**
1.1 Working of the Client-Server Pair . . . 1
1.1.1 Functional Overview . . . 1
1.2 Explanation of $$ and EOF . . . 1
1.3 Plot Analysis . . . 1
1.3.1 Explanation for the observed behavior at p=k . . . 2

2 Concurrent Word Counting Clients **2**
2.1 Functional Overview . . . 3
2.2 Plot Analysis . . . 3

3 Grumpy Server - ALOHA, BEB, SBEB Protocol **4**
3.1 Functional Overview . . . 5
3.2 Conclusion and Future Work . . . 5

4 Friendly Server and Scheduling Algorithms **6**
4.1 FIFO Scheduling . . . 6
4.2 Round-Robin Scheduling algorithm . . . 7
4.3 Plot Analysis . . . 8
4.4 Jain's Fairness index . . . 9

1 Word Counting Client

1.1 Working of the Client-Server Pair

In client-server communication using socket programming in C++, the key functions utilized for data transmission are `send` and `recv`. These functions are part of the `sys/socket.h` library. The `send` function is used to send data(both from client to server; and from server to client), while `recv` is employed to receive this sent data. The `send` and `recv` functions store the data in buffers associated with the sockets, which can be read at any time.

1.1.1 Functional Overview

1. `bool send_server_to_client_msgs(data_socket, k, p, client_number,offset_value)`
   This is the core function responsible for sending data packets from the server to the client. It has a job of sending `p` words at a time, up to `k` words in total, in multiple packets detecting and sending the `EOF` marker when the server has reached the end of the file. Returns `true` if `EOF` is reached, otherwise `false`.

2. `void handle_client(data_socket, k, p, client_number)`
   Receives offset requests from the client; Calls `send_server_to_client_msgs()` to send word data to the client.

1.2 Explanation of $$ and EOF

- `$$\n`: This special server message is sent by the server when the requested offset by the client exceeds the number of words available. No more requests after this will be sent by the client due to an out-of-bounds offset.

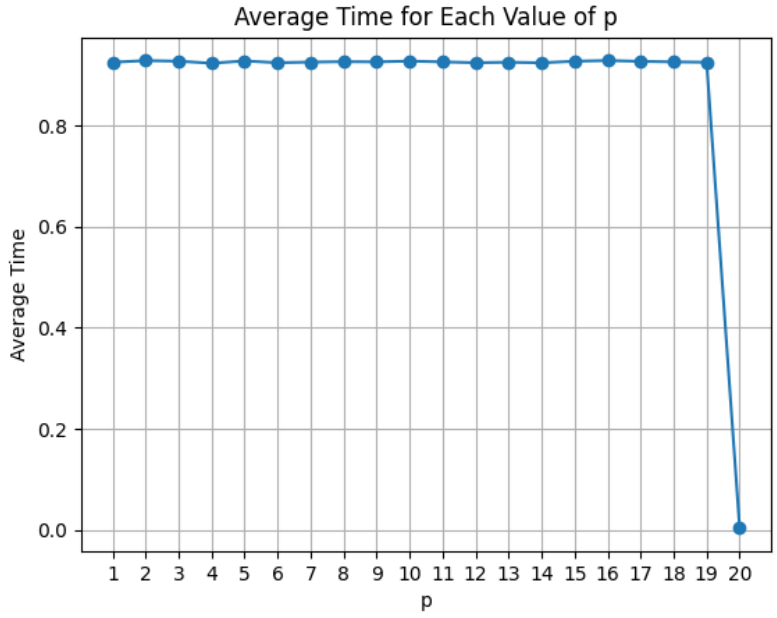
- `EOF\n`: The "End of File" (EOF) is a keyword in the server's file that marks the end of the word list. Once the client receives this keyword, it recognizes that this is the end of the frame and it has to stop the frequency computation at this word.

1.3 Plot Analysis

Experiment - for each value of p(size of a packet) (from 1 to 10), the completion times are logged, keeping k=10 constant. The plotting script takes these results and computes the average time across 10 runs. This helps visualize the impact of p on the performance of the client-server communication. Results Analysis

- For p = 1 to 9, the completion time remains relatively constant.

- For p = 10, the completion time drops drastically to 0.0085 seconds, indicating a substantial performance improvement when sending the maximum number of words in one packet.



1.3.1 Explanation for the observed behavior at p=k

In the case of p = 10, the completion time is significantly lower becaise:

1. **Single Request for Full Data:** Since p = k = 10, the client only needs to send one request to the server. The client receives all the required words in a single transmission. With less number of round trips between the client and server, the time spent on the request-response cycle is reduced significantly.

2. **Packet Division on server-side:** When p is smaller than k, the server has to split the words into multiple packets. This requires additional pre-computation on the server to properly chunk the data, maintain the correct offsets, and ensure the order is preserved across multiple transmissions. Wtih p=k, the pre-computation and the state-management of the server is reduced significantly, that is why the time is reduced.

2 Concurrent Word Counting Clients

Objective - There are multiple clients and a single server. There are different buffers on the server side for each client. The requests are handled concurrently, meaning the server can send packets to two or more clients at the same time.

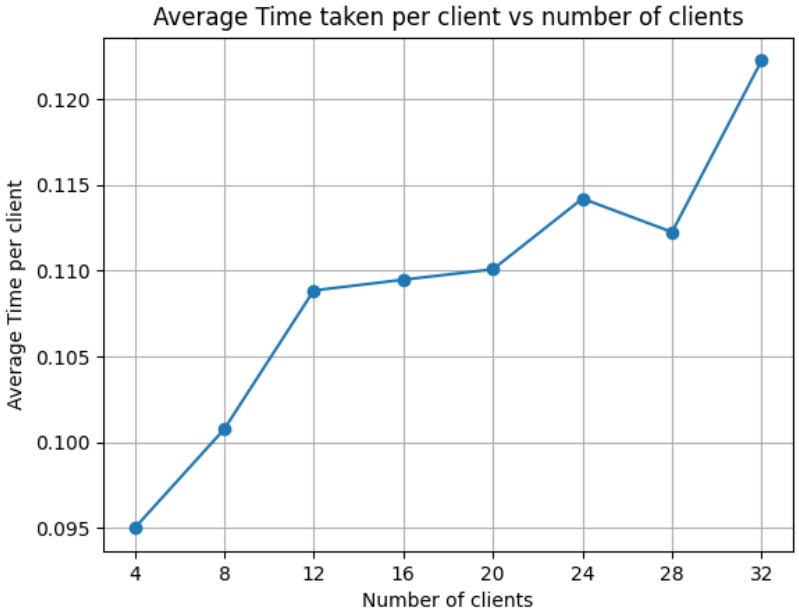
2.1 Functional Overview

- `void* client_thread(void* args)`
  A thread function that handles client requests concurrently.It defines the description of the socket and calls `handle_client()` to handle the interaction with the client. Each client connection runs in a separate thread created by this function, ensuring multiple clients can be served concurrently.

- `struct ThreadArgs`
  This structure is used to pass arguments to the client threads. Since `pthread_create` requires a single argument of type void*, the program will put multiple inputs (such as server address, k, p, and client number) into this ThreadArgs structure. Each thread will receive a pointer to its own copy of ThreadArgs.

- `run_client`
  This function is the entry point for each client thread. The function receives a pointer to a ThreadArgs structure. Each client thread creates its own socket. The client attempts to connect to the server. After successfully connecting to the server, the client engages in word requests by calling the `chat()` function.

2.2 Plot Analysis



On logging the average time per client and changing the number of clients, by incrementing by 4 every time, we get the above plot. The plot clearly tells that as the number of

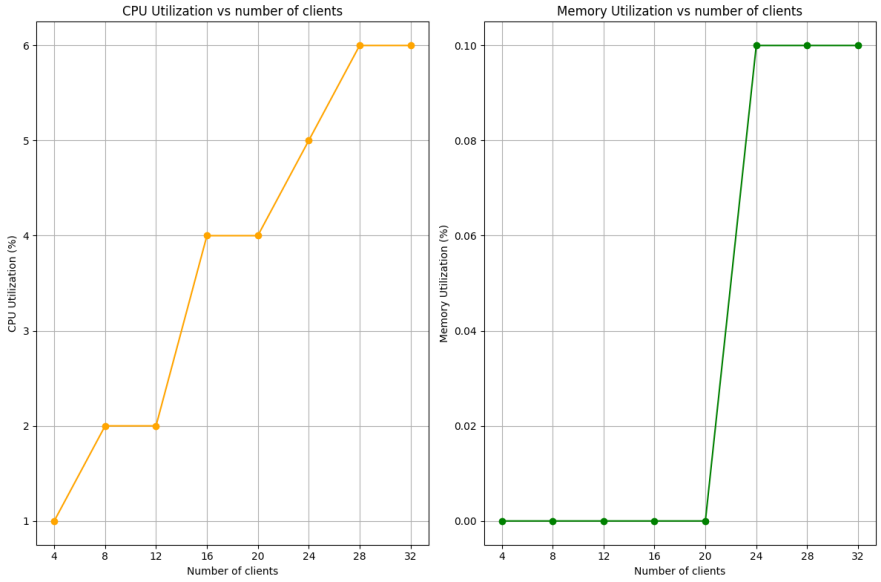
concurrent clients increases, the average time per client also shows a consistent increasing trend. There can be multiple reasons for this increase:

1. **Resource contention:** Multiple threads use some shared resources, in this part it is the memory containing words, and in other parts it can be some other shared variables. This leads to delays as the threads might compete for access.

2. **Thread Management Overhead:** Creating and managing a higher number of threads increases the overhead thread management which includes - creating, destroying, but more importantly scheduling threads. This can increase the response times.

3. **Context Switching:** With more number of threads, the operating system has to use the CPU multiple times to perform context switches. This leas to inefficiency and increased execution time since multiple threads are active simultanoeusly.

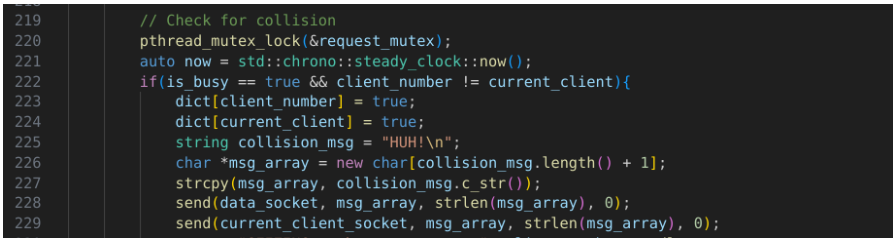
The increase in the number of clients essentially increases the work done by the server, keeping it busy by multiple threads at the same time, to handle more messages and process them concurrently. The additional analysis supporting the above statements is as follows:



3 Grumpy Server - ALOHA, BEB, SBEB Protocol

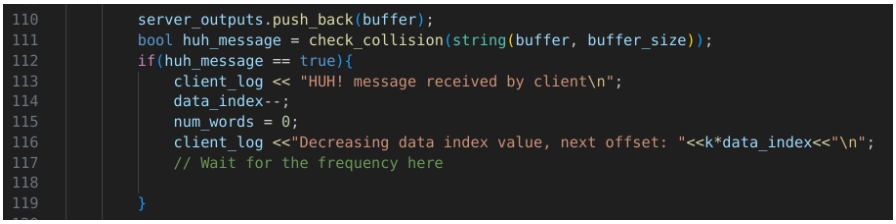
One server interacts with multiple clients simultaneously, but with a restriction that the server prefers serving only one client at a time. In such scenarios, the server sends a special message, `HUH!`, to both the existing and the new client, indicating that both should halt their communication and retry after a specific duration.

3.1 Functional Overview



```
// Check for collision
pthread_mutex_lock(&request_mutex);
auto now = std::chrono::steady_clock::now();
if(is_busy == true && client_number != current_client){
    dict[client_number] = true;
    dict[current_client] = true;
    string collision_msg = "HUH!\n";
    char *msg_array = new char[collision_msg.length() + 1];
    strcpy(msg_array, collision_msg.c_str());
    send(data_socket, msg_array, strlen(msg_array), 0);
    send(current_client_socket, msg_array, strlen(msg_array), 0);
```

Figure 1: Server side changes for the ALOHA Protocol



```
server_outputs.push_back(buffer);
bool huh_message = check_collision(string(buffer, buffer_size));
if(huh_message == true){
    client_log << "HUH! message received by client\n";
    data_index--;
    num_words = 0;
    client_log <<"Decreasing data index value, next offset: "<<k*data_index<<"\n";
    // Wait for the frequency here

}
```

Figure 2: Client side changes for the ALOHA Protocol

1. The primary challenge in this communication model is handling collisions. When a client sends a request while the server is already busy, both clients involved in the collision are notified with the `HUH!` message.

2. A key point is that the server must discard any partial communication with the current client when a collision occurs. Both clients independently retry communication using a distributed backoff protocol. This prevents frequent collisions and ensures that each client will eventually get served.

3. To manage concurrency and prevent race conditions, a mutex lock (`request_mutex`) is used. The server uses this lock to check if it is busy with a client. If a new client tries to request service during this period, the lock is used to trigger the `HUH!` response.

3.2 Conclusion and Future Work

- We developed a mechanism to detect these collisions and implemented logic to calculate the remaining time in the communication slot before allowing the server to become available again.

- However, we faced challenges in synchronizing the clients' retry attempts after a collision. While we were able to successfully detect collisions and track the time left in the communication slots, our attempts at implementing different synchronization mechanisms have produced inconsistent results.

- We tried multiple approaches, including using mutexes and conditional variables to control access and synchronization between threads. Despite consulting various resources, having discussions with colleagues, and experimenting with different design paradigms, we found it difficult to achieve reliable synchronization.
- These inconsistencies in results are likely due to gaps in our understanding of parallel programming concepts, particularly in the context of distributed systems. As a result, we were unable to fully synchronize client retry attempts after collisions, though we are confident in our ability to detect and manage collisions as they occur.

4 Friendly Server and Scheduling Algorithms

In this case the server can manage how the data has to be sent to each and every client. The server has allows collisions and stores all the requests received. It then has a scheduling algorithm, to prioritize some requests over the other. The scheduling algorithms implemented by us are explained as follows.

4.1 FIFO Scheduling

Algorithm Design: In this policy, the data requests are served in the order of their arrival. So, the idea is to maintain a common buffer across all the clients. Since the clients are running on parallel threads, the only way they can communicate across threads is through shared variables. For this we use mutex - mutual exclusion.
Shared variables: The `pending_requests` is the common buffer across all the clients

```
35  vector<pair<int, int>> pending_requests;
36  std::mutex pending_requests_mutex;
37  condition_variable cv;
```

Mutex: By locking all the threads, except the present. Changing the variables, and then unlocking all the threads if there is no collision

```
221         std::unique_lock<std::mutex> lock(pending_requests_mutex);
222         pending_requests.push_back({client_number, offset_value});
223
224         cv.wait(lock, [client_number]()
225                 { return pending_requests.front().first == client_number; });
226     }
```

Initialize the shared variables with values corresponding to no collision. As soon as a client sends a request to the server and the server is not busy, accept this request and keep it in the central buffer. During the execution of this request, if there is any other request sent, then append it to the buffer. As soon as the execution of the present request is complete, remove the first element from this common buffer. Important point here is that, whenever the buffer or the shared variables are changed, it is important to

lock the threads and after the changes are made, it is important to notify all the threads about this change.

4.2 Round-Robin Scheduling algorithm

Algorithm Design: In this policy, the data requests are served in a round-robin fashion. Each client has its own buffer, and the server processes requests from each client's buffer in a cyclic manner, ensuring fairness. Similar to the FIFO policy, the clients communicate across threads through shared variables, which are protected using mutex locks to ensure safety. **Shared Variables:** The `client_buffers` is a vector of buffers where each element corresponds to a particular client. Each buffer stores the pending requests from that client.

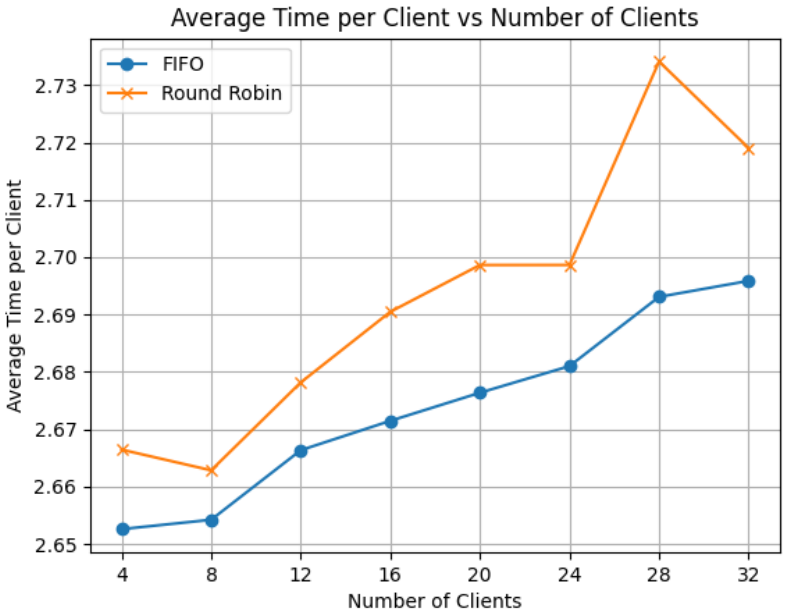
```
35  vector<vector<int>> pending_requests;
36  std::mutex pending_requests_mutex;
37  int curr_turn = 1;
38  std::mutex curr_turn_mutex;
39
40  condition_variable cv;
41  bool is_turn = false;
```

Mutex: To ensure that the buffers are accessed and modified safely, we use mutex locks. When the server processes a request from any client's buffer, it locks the shared variables, processes the request, and then unlocks them. The server checks each client's buffer in a round-robin order, and only non-empty buffers are processed.

```
251     {
252         std::lock_guard<std::mutex> lock(pending_requests_mutex);
253         std::lock_guard<std::mutex> lock1(curr_turn_mutex);
254
255         if (pending_requests[client_number].size() > 0)
256         {
257             pending_requests[client_number].erase(pending_requests[client_number].begin());
258         }
259         curr_turn++;
260         if (curr_turn > n_cl)
261         {
262             curr_turn = 1;
263         }
264
265         cv.notify_all();
266     }
```

The shared variables are initialized with empty buffers for each client. The server starts by checking the buffer of the first client. If the buffer is not empty, the request is processed and removed from the buffer. If it is empty, the server moves on to the next client's buffer. This continues cyclically across all clients. If a new request arrives while the server is processing another request, it is added to the corresponding client's buffer. Each time a buffer is accessed or modified, the corresponding thread locks the mutex. Once the changes are made, the mutex is unlocked, and all threads are notified of the changes. This maintains a kind of fairness among the clients.

4.3 Plot Analysis



This plot compares the performance of two scheduling policies: FIFO and Round Robin, in terms of average time taken per client for a varying number of clients (from 4 to 32, incrementing in units of 4).

1. **FIFO Performance:** The FIFO policy shows a more gradual increase in average time per client as the number of clients increases. The increase in average time is relatively linear. This miggt suggests that as the number of clients increases, it handles the load more predictably. This is because FIFO processes requests in the order they arrive, avoiding context-switching overheads and focusing on one request at a time.

2. **Round Robin Performance:** The Round Robin policy exhibits a little more erratic behavior. This changing regularity can be answered by context-switching overhead introduced by this policy. In Round Robin, the server cycles through all clients' requests, even if the client buffers are empty, which leads to additional overhead as the number of clients grows. This behavior is reflected in the sharp rise in average time with an increasing number of clients, like the peak at 28 clients indicates that this algo may struggle to handle many clients and cause significant delays. But since this is a fair algorithm, this might be sometimes good for service providers if they dont prioritize anyone.

3. **Conclusion:** FIFO tends to perform better overall as the number of clients increases, because it avoids the context-switching overhead present in Round Robin.

But, for applications where fairness is important, Round Robin might be useful

4.4 Jain's Fairness index

Jain's Fairness Index is a metric used to quantify the fairness of resource allocation among multiple clients. It takes values between 0 and 1, where 0 indicates complete unfairness (one client gets all resources) and 1 indicates perfect fairness (resources are equally distributed among all clients).
Calculation: Jain's Fairness Index J for n clients is calculated using the formula:

$$J = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

where x_i represents the share of resources allocated to the i-th client.
In this code, the x_i used to calculate Jain's Fairness Index corresponds to the time taken by each client to complete its task, stored in the time_clients vector.
Fairness Index Values:

- **FIFO:** Jain's Fairness Index: 0.907934
- **Round Robin:** Jain's Fairness Index: 0.997417

This pattern is consistently observed accross multiple test case files that the fairness index of round robin is larger than that of FIFO scheduling algorithm. This is what is ideally expected, that the round-robin implementation be more fair.
Factors Affecting Jain's Fairness Index:

- **Algorithm Choice:** Different scheduling algorithms can affect fairness. For example, FIFO tends to favor earlier arriving clients, while Round Robin aims for equal treatment.
- **System Load and Resource Constraints::** Higher system load can lead to decreased fairness due to increased contention for resources. Limited resources can impact fairness as clients compete for available resources.