Student Details

- **uid**: 'u7614074'
- **name**: 'Jugraj Singh'

References

- [1] Peter Reiher. Website Url - https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

Report

Overview

Our air traffic control (ATC) network implementation requires the following extensions to support concurrency and efficiency requirements, enabling it to handle multiple clients and airport nodes simultaneously. The primary extension tasks we implemented are:

- Thread Pooling and Connection Queue: The controller node calls the Accept method in a new thread to manage multiple clients at a time, but to avoid the creation of a new thread for every connection, the controller uses a thread pool. A synchronized connection queue take the client requests and organize them for the worker threads, allowing for the simultaneous execution.
- Fine-Grained Error Handling: A good number of validations have to be performed for different kinds of client mistakes, for example, improper airport IDs or wrong request parameters. When a client enters incorrect data the system produces comprehensible and precise error messages. The controller node acts more of an interface by communicating with clients and airport nodes. When a client connects, the controller parses and processes each incoming request in three main steps:

1. Parsing and Validation: When the controller receives a request, the controller breaks down the command and gets some values like airport ID and more. The controller regulates the data enter format and carries out an error check. In the situation where the request is for example, a malformed request or it is a request related to a wrong airport code the controller returns an error message to the client.
2. Forwarding Requests to the Correct Airport Node: In response to valid requests, the controller determines the airport node based on the airport ID in the request information. It creates a socket connection (local) to the listening port of the airport node and transfer the request over the network. The airport node processes the forwarded request such as a flight timetable or gate information and formulates a reply.
3. Returning the Correct Response: Once the controller gets a response from the airport node, the controller will send this response to the client. This ensures that the client receives timely and accurate information about their request.

Request format used

The format selected for forwarding requests from the controller node to particular airport nodes matches the format of the received requests from the clients. This way, the structure of each request remains unambiguous and let the sender’s intention become obvious. This format provides both efficiency and debug-ability benefits:

1. Efficiency: There is less preprocessing time required in a controller who follows a similar format which is very useful while working in the multi-threaded system. The airport node can directly understand the forwarded request without undergoing cumbersome tasks of converting data between various formats. Secondly, since the controller doesn’t have to remove or append any other metadata to the request, fewer processing steps are required which adds to time taken in request response cycle.
2. Debug-ability: It is easy to debug if the request format remains close to the format that the client used to input their request. Given that both, the controller and the airport node, process the request in a similar structure, the content of the request can be logged to facilitate the identification of problems in the communication chain. This format also corresponds to the expected responses and errors, allowing for a straightforward check to validate that the cores-of-competence reaching the airport nodes contain the correct parameters.

Extensions

Multithreading within the Controller Node In our implementation, the primary goal of adding multithreading to the controller node was to handle multiple client connections concurrently. We achieved this by establishing a thread pool where a set number of worker threads remain idle until a new connection request arrives. Each worker thread then dequeues a connection, processes it, and returns to an idle state. This approach allows for parallel handling of incoming client requests, avoiding the bottleneck of a single-threaded server.

However, the impact on performance remains somewhat limited. This design mainly optimizes for concurrent connection handling rather than speeding up request processing itself. The thread pool operates efficiently for basic concurrency, but fine-grained control over individual request processing is not implemented. Since each worker thread takes an entire request from the queue and processes it independently, there is no intricate or high-performance locking strategy applied to protect specific resources. Thus, while our approach successfully distributes connections to threads, it does not yield substantial performance benefits beyond the initial concurrency.

Multithreading within Airport Nodes The airport nodes use a similar thread pool mechanism to handle incoming forwarded requests concurrently. Each thread in the airport node's pool is responsible for processing a complete client request, allowing multiple requests to be handled in parallel. However, we opted for a straightforward concurrency model without fine-grained locking at the level of individual schedule time slots. Instead, we implemented basic locking on the queue level to prevent race conditions when accessing or modifying the list of client connections.

This approach enables safe and isolated processing of each client's request but does not maximize performance. The choice to avoid more complex locking mechanisms prevents deadlocks and reduces the risk of synchronization errors but sacrifices the efficiency of a more granular locking scheme. This makes the implementation straightforward and reliable but limits its scalability, as it lacks the finer locking necessary for highly optimized concurrent processing.

Testing

One significant implementation challenge encountered during this assignment was managing client requests consistently in a multi-threaded environment, specifically ensuring that each request was processed only once without repetition. In particular, controller considered the response by an airport as another request by a client. Also, the system exhibited unexpected behavior where it would repeatedly read the same request, leading to duplicate processing and incorrect responses.

Upon investigation, the issue was traced to the initialization of the rio (Robust I/O) structure in the handle_client_request function. Originally, we had placed the rio_readinitb call within the while loop that processed incoming client requests. This meant that each time the loop iterated, the rio structure would be re-initialized, causing it to reset its internal buffer and effectively "re-read" the same data from the beginning. Consequently, the same request appeared to be processed repeatedly, as if it were new each time.

The solution involved moving the rio_readinitb call outside the while loop so that the rio structure was initialized only once when the client connection was first established. This allowed rio to retain its position in the input stream across loop iterations, ensuring that each new line read by rio_readlineb reflected the next incoming request from the client. By making this change, we eliminated the redundant re-reading issue and achieved consistent, one-time processing of each request.

This challenge was particularly difficult to diagnose because the repeated behavior appeared similar to a client-side error or network issue. However, careful debugging and a closer inspection of the rio initialization logic revealed the root cause. Addressing this issue improved the robustness of our request-handling implementation and provided valuable insight into how buffered reads should be managed in a multi-threaded environment.

Test Results

Test basic-1

```
Running make all:                ok!
Running test basic-1:            passed!
Total: 1/1 tests passed.
```

Test basic-2

```
Running make all:                ok!
Running test basic-2:            passed!
```

```
Total: 1/1 tests passed.
```

Test basic-3

```
Running make all:               ok!
Running test basic-3:           passed!
Total: 1/1 tests passed.
```

Test basic-4

```
Running make all:               ok!
Running test basic-4:           passed!
Total: 1/1 tests passed.
```

Test basic-5

```
Running make all:               ok!
Running test basic-5:           passed!
Total: 1/1 tests passed.
```

Test basic-6

```
Running make all:               ok!
Running test basic-6:           passed!
Total: 1/1 tests passed.
```

Test multi-1

```
Running make all:               ok!
Running test multi-1:           passed!
Total: 1/1 tests passed.
```

Test multi-2

```
Running make all:               ok!
Running test multi-2:           passed!
Total: 1/1 tests passed.
```

Test concurrent-1

```
Running make all:               ok!
Running test concurrent-1:      passed!
Total: 1/1 tests passed.
```

Test concurrent-2

```
Running make all:                ok!
Running test concurrent-2:       passed!
Total: 1/1 tests passed.
```

Test concurrent-3

```
Running make all:                ok!
Running test concurrent-3:       passed!
Total: 1/1 tests passed.
```