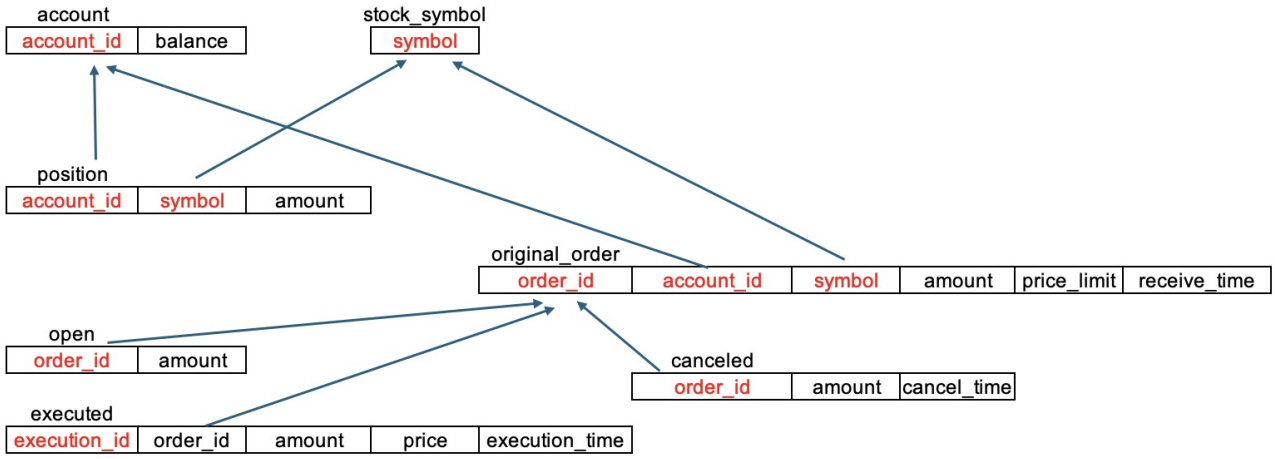
a. Project design

a..1 Database design



a..2 Multi-thread design

Strategy Overview: The server application is designed to spawn a new thread for each incoming client request. This approach enables simultaneous and independent processing of multiple requests.
Implementation: Utilizing the std::thread class from the C++ standard library, our server creates a thread that executes a designated function, "*handleClient*", which manages the request-specific operations.

```
thread clientThread(handleClient, client_fd, client_ip);
clientThread.detach();
```

1 Scalability

a. Test Infrastructure

The primary aim is to assess scalability by limiting the server and the client's operation to different numbers of CPU cores and observing the impact on handling a set number of requests.
CPU Core Limitation: The server and client's CPU core usage is restricted in varying scenarios to 1, 2, 4, and 8 cores. This is achieved using the taskset command, which effectively constrains the server to operate within the defined core limit.

From server's side,we modify the command line to limit server run on specific core:

```
taskset -c 0-7 ./server
```

From client's side,we modify the run_test.bh to limit server run on specific core:

```
#!/bin/bash

fileNum=30
maxThread=50
# Loop over different CPU core configurations
for cores in 1 2 4 8
do
    start_time=$(date +%s.%N)
    echo "Running clients with $cores cores"
    total_requests=0

    for ((j=1; j<=maxThread; j++))
    do
        # Loop from 1 to 100 for testing
        for ((i=1; i<=fileNum; i++))
        do
            filename="test${i}.xml"

            if [ -f "$filename" ]; then
                # Run the client with CPU affinity
                ./client "$filename" &
                client_pid=$!
                taskset -cpa 0-$(($cores-1)) $client_pid > /dev/null
                wait $client_pid
                ((total_requests++))
            fi
        done
    done
```

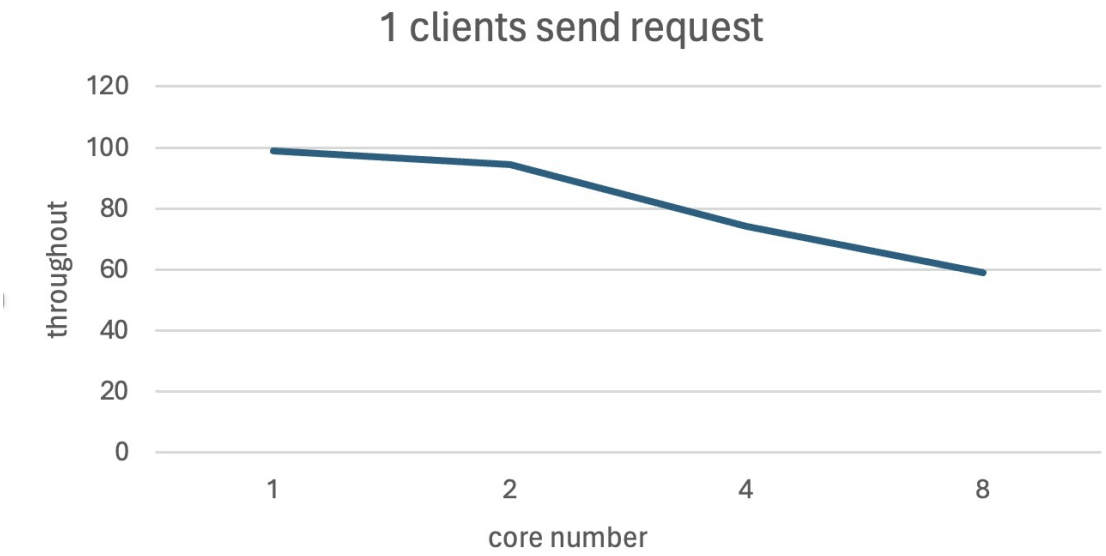
Test Case Execution: A total of 30 distinct test cases are executed, each run 50 times, culminating in 1500 requests. This approach ensures a comprehensive analysis over multiple iterations, providing a robust dataset for evaluating performance across different CPU core allocations.
In addition to limiting the server to run on different numbers of CPU cores, the testing strategy includes simulating varying levels of client-side concurrency. This is achieved by altering the number of client instances simultaneously sending requests to the server.
Adjusting Client Load: The number of concurrent client instances, referred to as **"instances"**, is systematically varied during tests. This approach allows the evaluation of server performance under different levels of client request loads.
Simultaneous Request Dispatch: The method involves initiating multiple clients concurrently, each sending requests to the server at the same time. This setup mimics real-world scenarios where a server may face simultaneous incoming traffic from numerous sources.

```
#!/bin/bash

instances=1

for ((n=1; n<=instances; n++))
do
    ./run_test.sh > "output_$n.txt" 2>&1
done

wait
echo "finished all"
```

b. Experiment

b..1 One client

The test results reflect the server's performance in handling requests from a single client while being restricted to run on 1, 2, 4, and 8 cores, respectively. The throughout is shown as below:



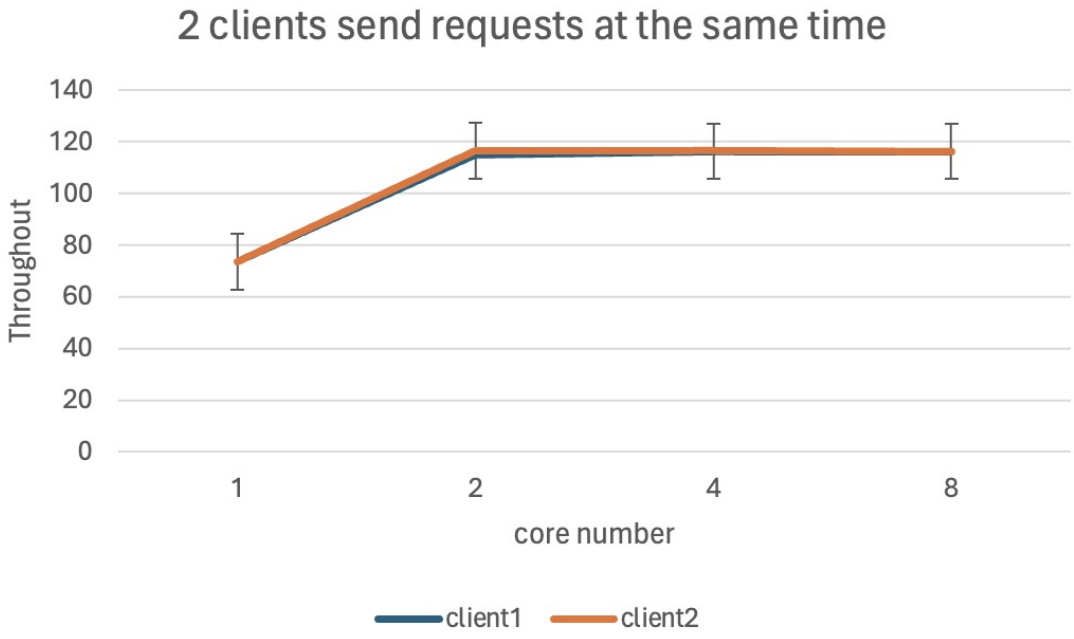
Interpretation of Results: When operating on a single core, the server displays a baseline throughput of 99. This is indicative of the maximum load or the number of requests it can handle under the most restricted computational resource scenario. As more cores become available, we expect an increase in throughput. However, the observed trend is a decrease in throughput with additional cores. Possible explanations include:

- Overhead of Parallelism: Increasing cores might introduce additional overhead, especially if the server or the application isn't optimized for parallel processing. This can include context switching, synchronization, or other parallel computing inefficiencies.

- Single Client Limitation: With only one client sending requests, there might not be enough concurrent tasks to fully utilize the additional cores. Hence, the server might not exhibit significant throughput improvement.
- Resource Management and Scheduling: The operating system's task scheduling and resource management can also impact how effectively additional cores are utilized. Inefficient scheduling could lead to underutilization of added cores.

b..2 Two client

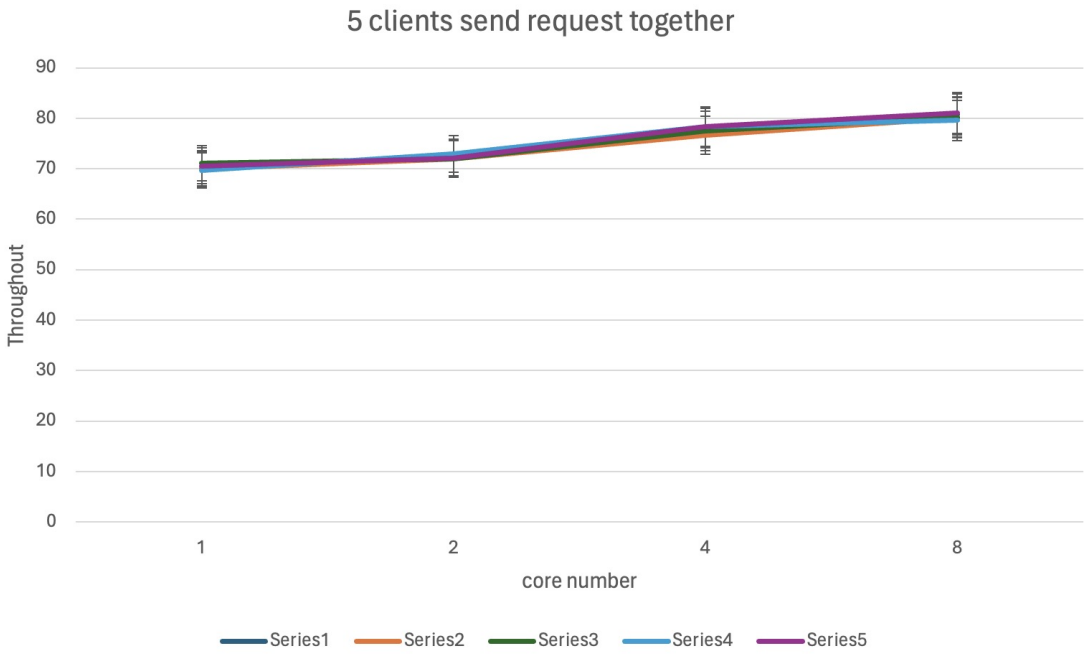
The results display the throughput achieved by the server when handling requests from two clients concurrently, with CPU core limitations set to 1, 2, 4, and 8 respectively. The throughout is shown as below:



Interpretation of Results:
With just one core, the server's throughput is at its lowest, indicating that a single core might be a bottleneck when trying to manage dual client requests concurrently.
Doubling the cores to two results in a marked improvement in throughput. This indicates the server's capability to effectively utilize additional CPU resources to handle parallel tasks, significantly reducing the wait time for each request to be processed.
The minimal increases observed when cores are further doubled to four and eight may imply that after a certain point, additional cores do not contribute to performance for two concurrent client requests. This could be due to that with more cores at play, the overhead of synchronizing data between threads might offset the benefits of parallel execution, or there are potential bottlenecks outside of CPU processing, such as network or disk I/O, which could be limiting performance.

b..3 Five clients

The results display the throughput achieved by the server when handling requests from five clients concurrently. The throughout is shown as below:



Interpretation of Results:
Initial throughputs for all clients when restricted to a single core are similar, which is expected as they share the limited computational resource.
As the number of available cores increases, The server exhibits increased throughput with each doubling of CPU cores. This indicates a positive scalability trend, showcasing our server's capability to leverage additional computational resources.
However, there is a trend of diminishing returns on throughput with higher core counts is evident, particularly when moving from 4 to 8 cores. This suggests that there are limits to the server's ability to scale with additional cores, which could be investigated further for optimization.