# Exchange Server Scalability Report

## Introduction

We implemented an exchange server using per-request threading and optimistic concurrency control. Then we built the test infrastructure using multi-thread and performed load testing on the server.

## 1. Server Implementation

- **Create a thread per request**

For the server-side implementation, we use the **per-request** threading strategy, with some optimization to limit the total amount of threads created. This implementation has drawn on the idea of a thread pool but is a rough imitation with overhead in creating new threads.

- **Main thread**

There is one main thread running endlessly, accepting new TCP connections from new clients, and receiving requests from the clients. In order to constrain the total amount of threads, it continuously pushes new requests into a request queue.

- **Thread managing thread**

The thread managing thread is responsible for creating a new thread to handle each request. When the amount of threads has not exceeded the specified limit, it will pop a request out of the queue and assign a new thread to handle it.

- **Request queue & thread limit**

The managing thread would only create a new thread when the queue is not empty and the amount of threads has not exceeded the specified limit. This constraint is forced because of the limitation of the PostgreSQL database. For every thread, we have to create a new connection to our database, because pqxx::connection is not thread-safe, and we cannot use a shared connection between multiple threads. However, there is a maximum number of connections (which is 100 in default). In order to prevent database overload, we need to add the thread limit.

- **Optimistic Concurrency Control**

We use optimistic locking to handle the concurrency in our database. In each table, we have a version column which is 1 by default. Every time we modify a row, we increase the version of that row, and all UPDATE operations will check for the version to ensure it hasn't changed since the row was read. We use the **affected_rows()** method in pqxx::result, if none of the rows were affected by UPDATE, the transaction is aborted, an exception is thrown and the outsider code will handle it (rollback and reread).

## 2. Test Infrastructure

- **Client as a server**

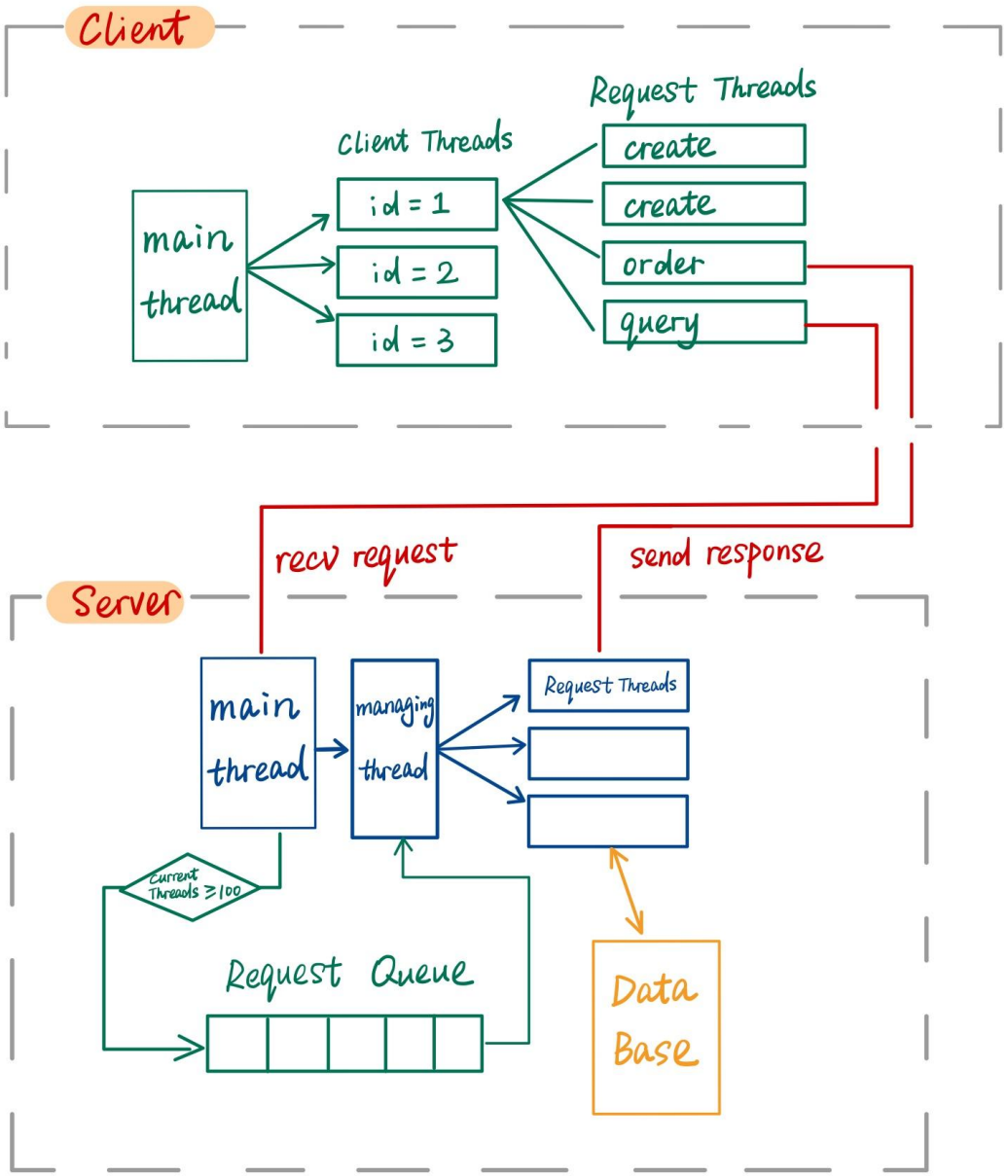
For testing, we implemented a client-side based on the same logic as the server. It is also **multi-threaded**. The main thread creates multiple "client" threads, and each "client" is assigned a certain id(representing the account id of this client). Then each "client" generates multiple "request" threads, each request thread will send one XML request to the server, then receive the response from the server and save it as an XML file for future reference.

- **Randomly generated requests with mix/match operations**

Each client will send randomly-generated requests which are a mix of different request types. In our experiment, each client will send 5 "create" requests, in each request create 1 account and 2 symbols(randomly chosen from our preset symbols), and 5 "transaction" requests, in each request, create a buy/sell order, the amount and price for the order is randomly generated within a certain range, such that there are many different buy/sell orders in the database to be matched every second.

- **Performance Metrics**

We tested the **throughput** and **latency** of our exchange server running under a **different number of processor cores**.


Client
Request Threads
Client Threads
create
id = 1
create
main thread
id = 2
order
id = 3
query
recv request
send response
Server
main thread
managing thread
Request Threads
current Threads ≥ 100
Request Queue
Data Base

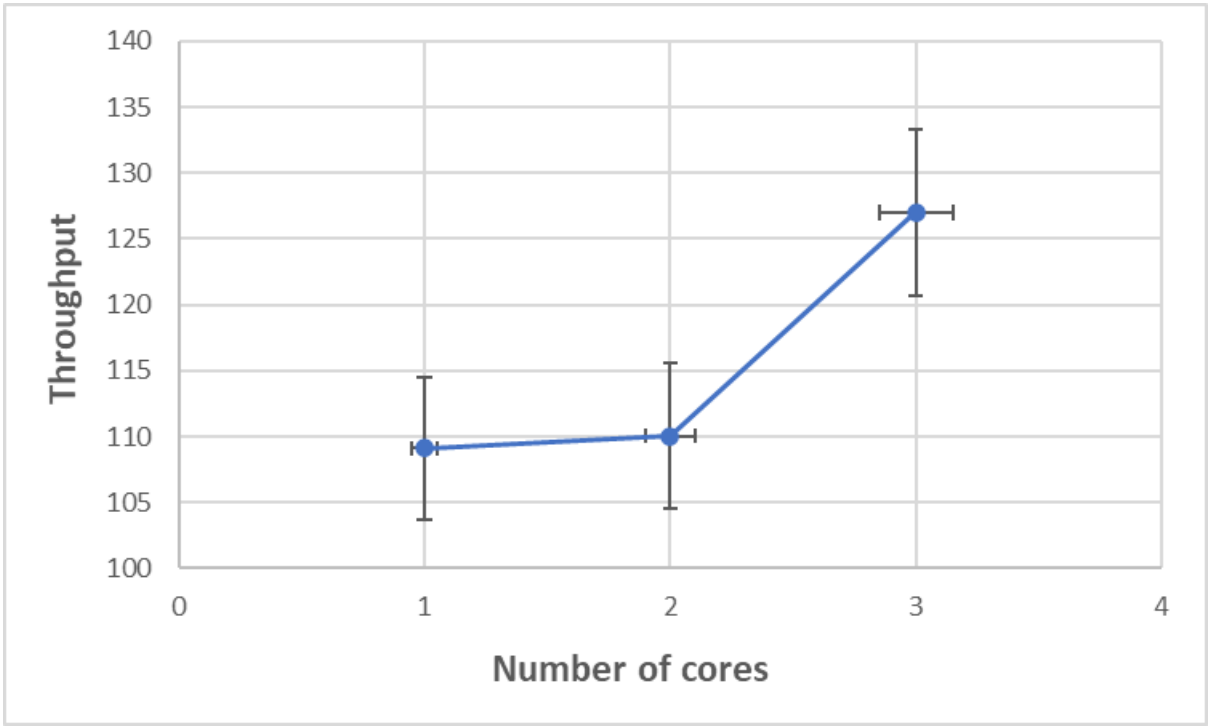
# 3. Scalability Analysis

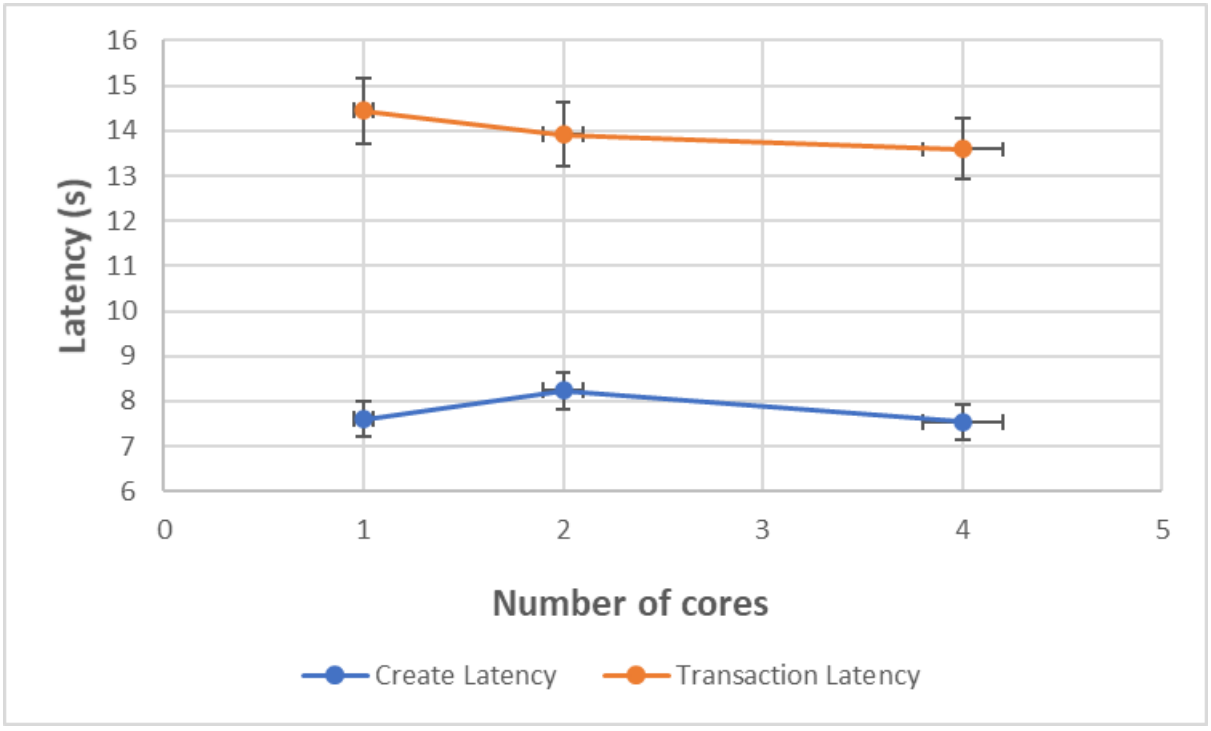
## 3.1 Concurrency with one client machine

| Num of cores | Num of requests | Total run time | Throughput |
|---|---|---|---|
| 1 | 1500 | 14.283 | 105.02 |
| 2 | 1500 | 14.193 | 105.69 |
| 4 | 1500 | 13.6789 | 109.65 |
| 1 | 2000 | 14.1398 | 141.44 |
| 2 | 2000 | 14.3152 | 139.71 |
| 4 | 2000 | 13.6156 | 146.89 |

## 3.2 Concurrency with two client machines:

| Num of Cores | Num of Requests | Total Run Time | | Avg Latency | | Throughput |
|---|---|---|---|---|---|---|
| | | machine #0 | machine #1 | Create | Transaction | |
| 1 | 4000 | 36.6644 | 38.099 | 7.6044 | 14.4489 | 109.10 |
| 2 | 4000 | 36.3520 | 37.1501 | 8.2391 | 13.9219 | 110.04 |
| 4 | 4000 | 31.4907 | 36.4071 | 7.5378 | 13.5908 | 127.02 |




We can see that the throughput is improving tremendously as the number of cores increases. The hardware resource contention decreases, while the communication latency goes up, as the former one is dominant, we end up with higher throughput.

The latency, on the other hand, does not show much difference as the number of cores changes. We analyze the latency of “Create” and “Transaction” type requests separately. As shown in the results, the transaction requests have a longer latency, nearly twice as much as the create requests because they involve modification operations in the database. While our throughput improves, the latency does not decrease a lot, possibly because the communication cost between clients and the server is dominating.

## 4. Further Optimization

Due to time constraints, we didn’t realize all our visions. We had designed a prototype that employs pre-create threading and pipeline parallelism.

- **Pre-create threading with thread pool**

Instead of creating a new thread per request, we can pre-create a certain number of threads, and use a thread pool to manage all the unoccupied threads. This implementation would benefit a lot in load balancing.

- **Pipeline Parallelism**

Instead of using one thread to handle the whole request (parse, execute, interact with the database, generate response, send back response), we can separate different stages of request handling and use pipeline parallelism. Generally, we would design three stages: receive and parse the requests -> read and write in database to execute the requests -> generate responses and send them back. There will be many threads in the second stage, performing operations in the database. It is nice to have many dedicated database-related threads, for load-balancing and easy control of the database.