Exchange Matching Engine Scalability Analysis

1. Introduction

This report analyzes the scalability of our exchange matching engine implementation, focusing on how the system's throughput changes with different numbers of CPU cores. The exchange matching engine is a critical component of any trading platform, responsible for matching buy and sell orders efficiently while maintaining the integrity of the market.

1.1 Project Overview

Our exchange matching engine is designed to:

- Process buy and sell orders for various symbols (stocks/commodities)
- Match compatible orders based on price and time priority
- Maintain account balances and positions
- Support order querying and cancellation
- Handle all operations atomically to ensure data consistency

The implementation uses **Golang** for its concurrency features and PostgreSQL for persistent storage of accounts, positions, orders, and executions.

1.2 Scalability

To evaluate the scalability and performance of our system under high load, we conducted stress testing using **Locust**, a Python-based load testing tool designed to simulate realistic user behavior. Locust allowed us to define user scenarios programmatically and scale the number of concurrent virtual users to test the limits of our application.

In addition to Locust, we used the `top` command to monitor CPU and memory utilization in real time during the tests, providing a quick and reliable way to detect resource saturation and potential bottlenecks.

For deployment, we leveraged **Docker Swarm** to orchestrate multiple service instances and distribute load across containers. To manage CPU resource allocation more precisely, we utilized Docker's built-in CPU control features:

- `cpus`: to limit the number of CPU cores available to each container (e.g., `--cpus=1.5`)
- `cpuset-cpus`: to pin containers to specific cores (e.g., `--cpuset-cpus="0,2"`)

These configurations enabled us to simulate various scaling scenarios, test container-level isolation, and evaluate the system's performance under constrained CPU conditions.

Overall, the combination of Locust, system monitoring tools, and Docker's resource control mechanisms allowed us to perform detailed and reproducible scalability tests.

1.3 Usage

```
### simply compose up would trigger the load test
### this would automatically open the loadtest with the server
sudo docker-compose up

### However We provide a server only entry
sudo docker-compose up app
```

2. Architecture Overview

2.1 System Components

Our exchange matching engine consists of the following key components:

- **Server**: Handles client connections, parses XML requests, and coordinates responses
- **Database**: PostgreSQL database for persistent storage of accounts, positions, orders, and executions
- **Exchange Engine**: Core matching logic that pairs compatible buy and sell orders
- **Order Heaps**: Priority queues for maintaining order books, optimized for buy-side (highest price first) and sell-side (lowest price first), in server allocation using LRU mechanism
- **Stock Pool**: LRU cache for frequently traded symbols to reduce database load
- **Account Management**: Handles balance and position updates

2.2 Concurrency Model

The system employs Go's concurrency primitives:

- Goroutines for handling multiple client connections
- Mutexes for thread-safe access to shared resources
- Read/write locks for optimizing concurrent read operations
- Database transactions for ensuring atomicity and rollback ability

2.3 Data Flow

1. Client sends XML request over TCP
2. Server parses the request and identifies the operation type (create, order, query, cancel)
3. Operation is processed, potentially triggering the matching engine
4. Matching engine attempts to pair compatible orders
5. Database is updated with results

6. Updating the in memory LRU heap pool
7. Response is formatted as XML and sent back to the client

3. Experimental Methodology

3.1 Test Environment

- **Hardware**: ECE 568 Multi-Core VM
- **Docker**:
- **Network**: [Network specifications to be added]

3.2 Measurement Methodology

For each test scenario, we:

- Vary the number of available CPU cores
- Run each test multiple times to account for experimental variance
- Measure the throughput (operations per second)
- Calculate average throughput
- Plot the results with error bars

3.3 Tools Used

- Custom load generator (included in the testing directory)
- PostgreSQL monitoring tools
- System resource monitoring (CPU, I/O, taskset)

4. Results and Analysis

4.1 Test Command Usage

Interactive Tester:

```
### erss-hwk4-xt66-yy465/docker-deploy/test/script/tester.py
python3 tester.py
### This command sets up a single client instance from localhost to send input interacted ###
xml data to the server, interaction i/o like below:
Create which XML (create/transaction)? transaction
Account ID for transaction: xuantang
Add 'order', 'query', 'cancel' or 'done'? query
Query ID: 1
Add 'order', 'query', 'cancel' or 'done'? done
<?xml version="1.0" encoding="UTF-8"?>
```

```
<transactions id="xuantang">
  <query id="1"/>
</transactions>

Sent 101 bytes to 127.0.0.1:12345
Response from server:
102
<?xml version="1.0" encoding="UTF-8"?>
<results>
  <error id="1">order not found: 1</error>
</results>
Create which XML (create/transaction)?
```

Funtinality Test:

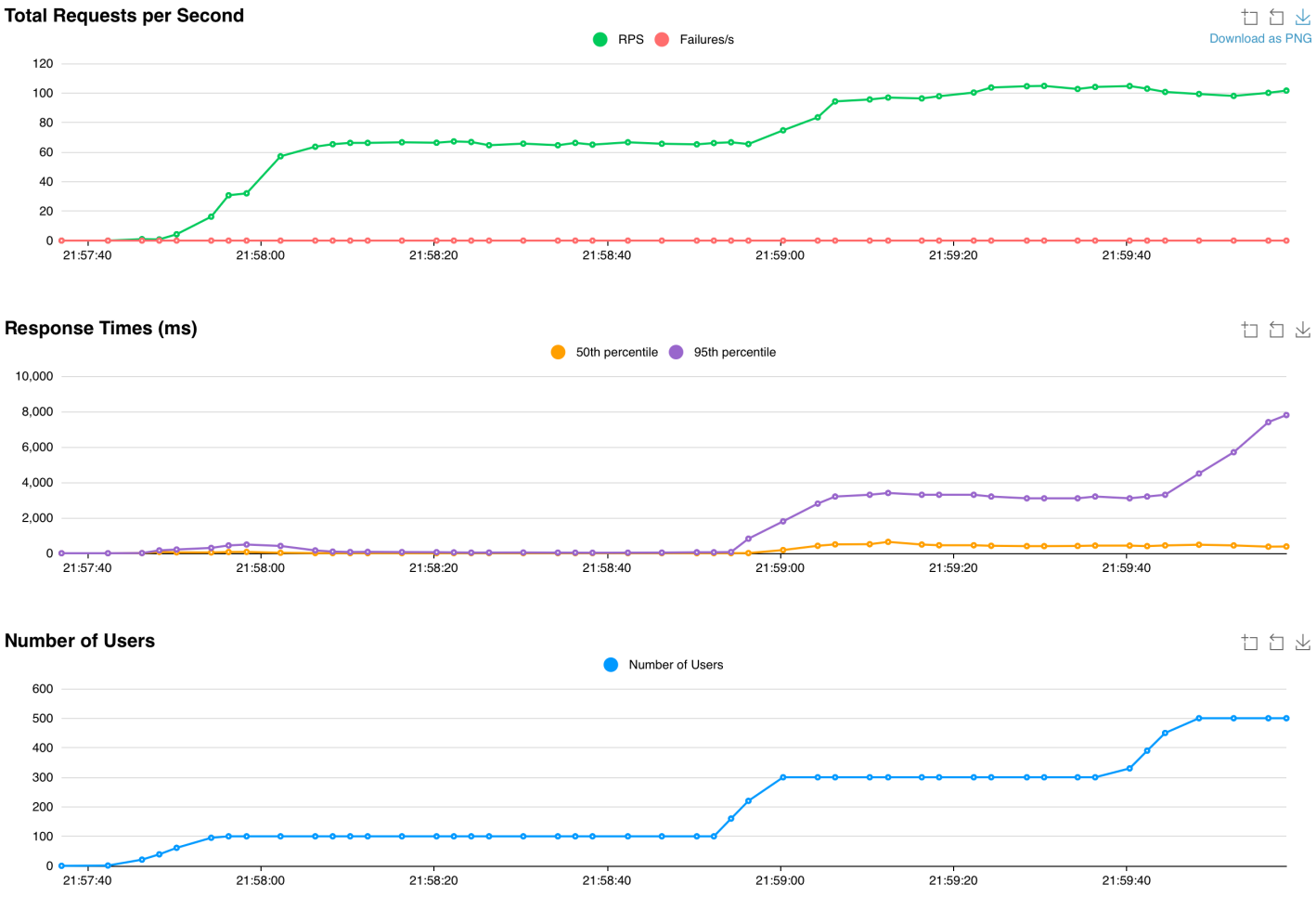
```
### docker-deploy/test/resource/*_test/run_test.sh
send_to_server() {
    local file=$1
    local length=$(wc -c < "$file")
    (echo "$length"; cat "$file") | nc -N localhost 12345
    echo "\n"
    sleep 2
}
check_files() {
    for file in 1_create.xml 2_buyorder1.xml 3_sellorder1.xml 4_multiorder.xml 5_query.xml
6_cancel.xml 7_mix.xml; do
        if [ ! -f "$file" ]; then
            echo "Error: File $file not found!"
            exit 1
        fi
    done
}
main() {
    echo "Starting test sequence...\n"
    check_files
    send_to_server "1_create.xml"
    send_to_server "2_buyorder1.xml"
    send_to_server "3_sellorder1.xml"
    send_to_server "4_multiorder.xml"
    send_to_server "5_query.xml"
    send_to_server "6_cancel.xml"
    send_to_server "7_mix.xml"

    echo "All files have been sent to the server."
}
main
### This is where you can run multiple selected functionality tests, done by
cat <file> | nc -N localhost 12345
```

Load Tests:

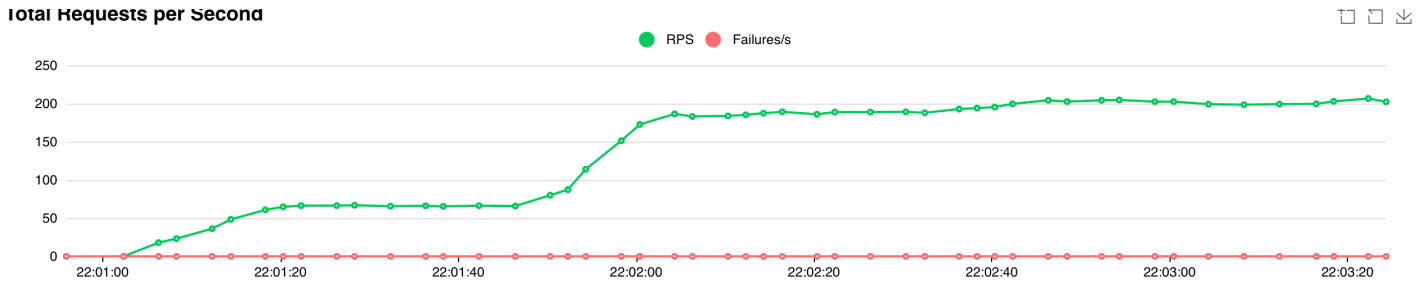
```
### inside docker-deploy/testing/cores_*_test.sh
chmod 777 *.sh
### To reproduce the test results
./cores_*_test.sh
### forward vm port to connected machine's 8089 port
### Or you can view on http://vcm-xxxxx.vm.duke.edu:8089/
### So you can do browser views for graphs
### You can adjust the max user and user spawn rate according to the website
```

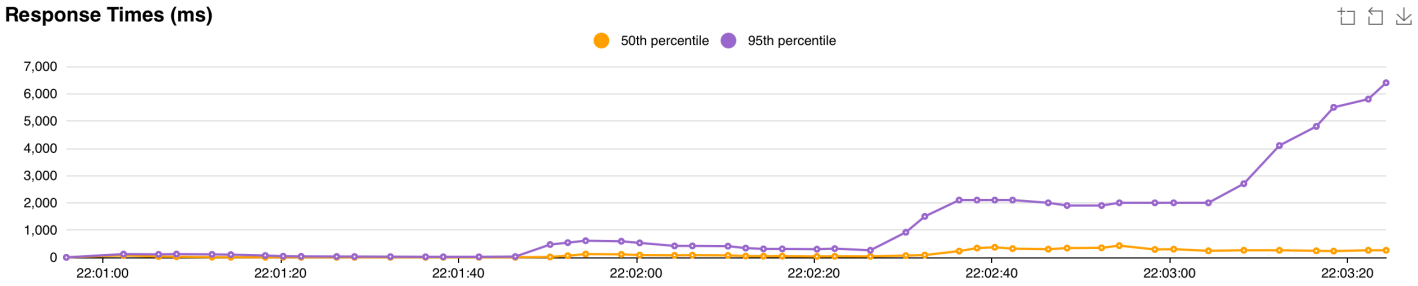
4.2 Result Graph

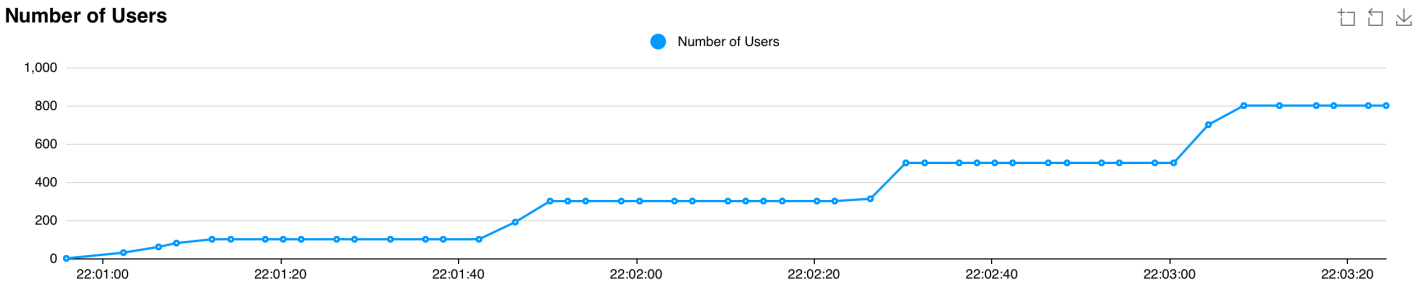
1-core: 100 rps, takes 100% of the cpu, mainly occupied by DatabaseIO



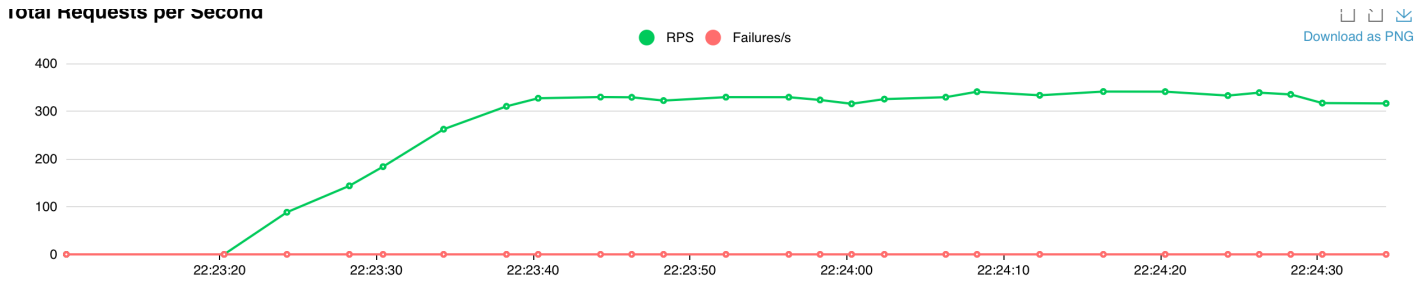
2-cores: 200rps takes 100% of cpus, mainly occupied by DatabaseIO

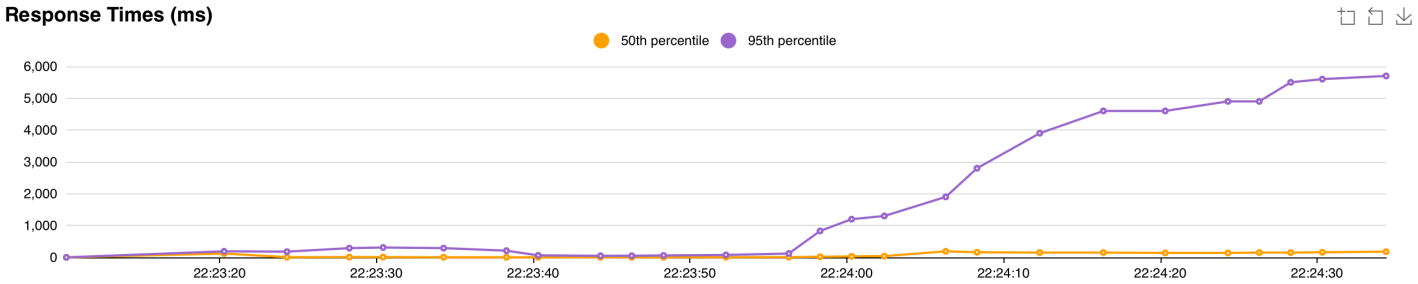


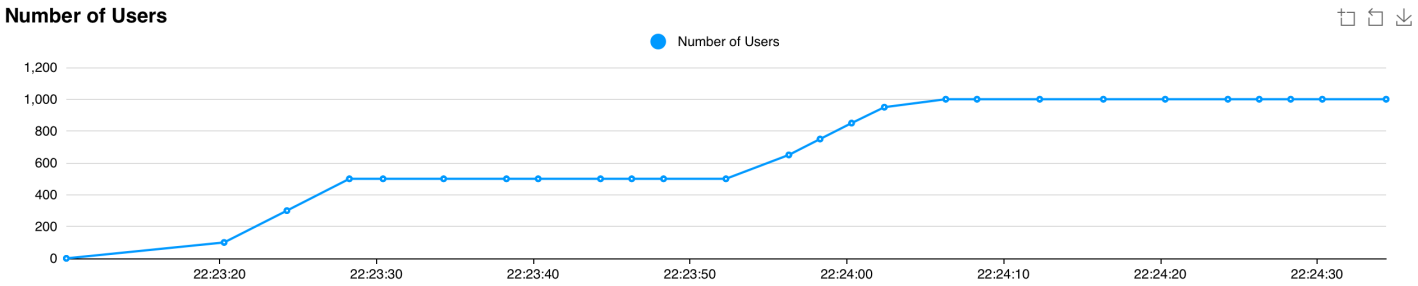




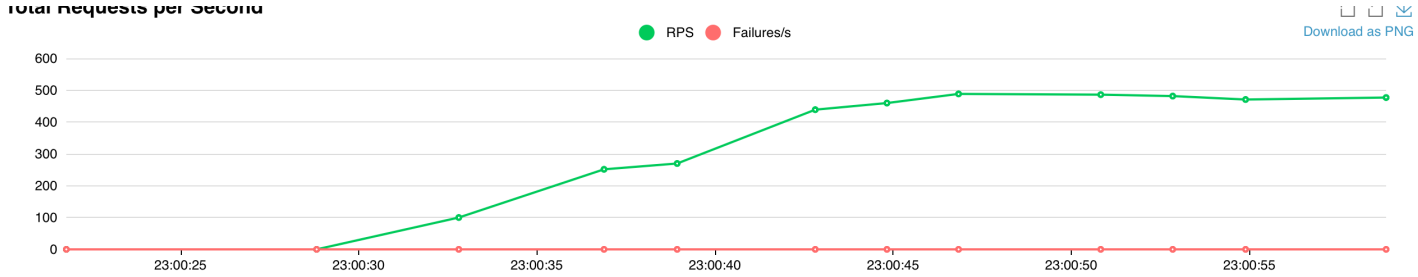
3-cores: 315rps takes 100% of cpus, mainly occupied by DatabaseIO

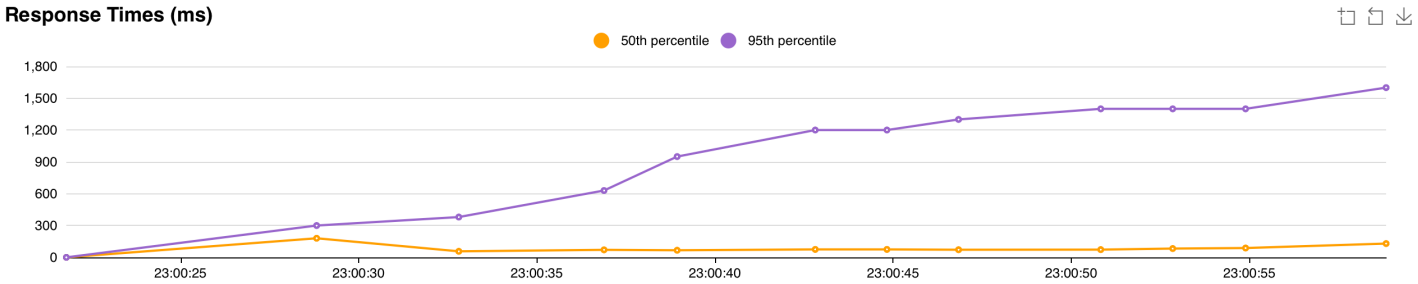


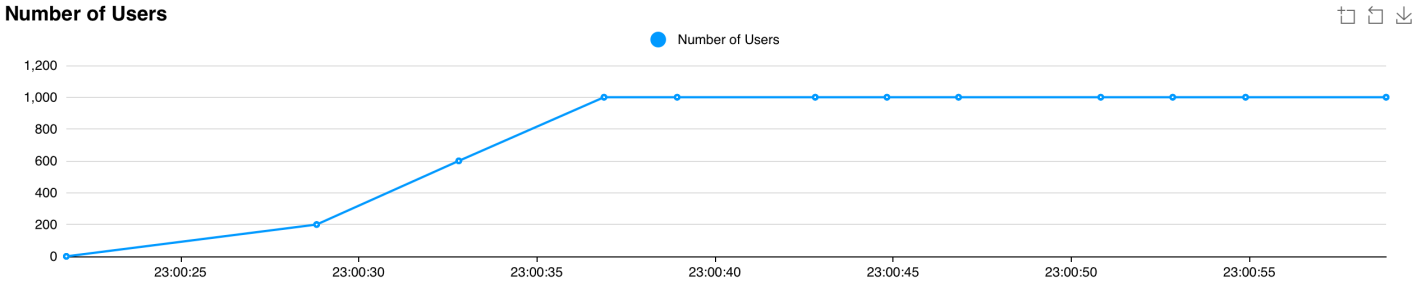




4-cores: 450rps takes 100% of cpus, mainly occupied by DatabaseIO







4.3 Bottleneck Analysis

During our load testing, we intentionally designed a high collision rate for order placements, meaning that a large number of incoming orders are likely to interact with overlapping price points or quantities. As a result, the primary bottleneck observed was within the **database I/O**, particularly in read and write operations under high concurrency.

We implemented a **heap-based LRU optimization strategy** specifically targeting the **order matching process**. This approach was designed to improve the performance of matching frequently accessed orders under real trading scenarios. However, in our synthetic load test environment, the rate of actual successful matches triggered during test execution was relatively low compared to the volume of incoming requests.

Because of this imbalance, the optimization did **not yield significant performance improvements in the test dataset**, even though the system was functioning correctly. We anticipate that in **real-world scenarios**, where matches are more meaningful and frequent, this optimization will have a much more **pronounced impact** on reducing latency and improving throughput.

In summary, while our bottleneck during load testing was mainly at the storage layer due to the artificially high volume of conflicting orders, the matching-layer optimizations are expected to show greater benefits under realistic market conditions.

5. Optimization Strategies

Based on our scalability analysis, we have identified several potential optimization strategies:

5.1 Database Optimizations

- Connection pooling tuning
- Transaction isolation level adjustments
- Partitioning strategies for orders table

5.2 Concurrency Optimizations

- Fine-grained locking strategies
- Lock-free data structures where appropriate
- Goroutine pool management
- Goroutine Channel Pipeline Strategy

5.3 Memory Management

- Cache for Order and Executions, to eliminate database IO for query
- Pre-allocation strategies
- Memory usage monitoring and optimization