Mini Stock Exchange

Engineering Case — Arthur Lobo

The Challenge

- Build a mini stock exchange
- Accept orders, match buyers to sellers, execute trades
- **Write:** submit orders (limit + market), cancel orders
- **Read:** order status, order book, stock price, broker balance

API Endpoints

Method	Endpoint	Description
POST	`/register`	Register a new broker (admin only)
POST	`/orders`	Submit a limit or market order
GET	`/orders/{id}`	Order status + trade history
POST	`/orders/{id}/cancel`	Cancel an open limit order
GET	`/stocks/{symbol}/price`	Last trade price + moving average
GET	`/stocks/{symbol}/book`	Order book (aggregated by price level)
GET	`/balance`	Broker's net cash balance

B3 Reference Numbers

B3 Metric	Value	Source
Daily trades	~4 million (~140/sec)	B3 daily market bulletin
Daily orders	~20 million (~700/sec)	Estimate — 5:1 order-to-trade ratio
Symbols	~450	B3 listed stocks
Brokers	~100	B3 registered brokers
Traffic mix	~53% orders, ~19% cancel, ~28% read	Estimate from other exchanges

How I Tested It

Full Realistic Simulation

- Sends requests at B3's per-second rates for a 60-second window
- Randomly spaced requests to simulate bursts.
- A few symbols get most of the orders, just like real markets.
- Controls **what % of B3 traffic** to send (e.g., 100% = full B3 load)
- Outputs a full report: latency percentiles, time series, and error rates

Correctness Tests: Verifies trading logic is correct, untimed.

Micro-Benchmarks: Times individual operations.

V1: Everything in the Database

```
Client → FastAPI → PostgreSQL
                   (matching + storage)
```

- Stateless API + PostgreSQL
- Every order = one database transaction
- Row-level locking keeps things consistent
- Orders for the same symbol processed sequentially

V1: Results

~40% of B3 (280 orders/second)

Bottleneck: Database Locks

- Multiple brokers trading the same stock → requests wait in line → latency spikes
- The DB was doing two jobs: **storage** AND **matching**
- Matching is the expensive one — more complex and frequent.

V2: Match in Memory, Read from Database

```
Client → FastAPI → In–Memory Engine → background flush → PostgreSQL
                   (sorted dict + queue)                   (storage)
         ↓
         Reads still go to DB
```

- Matching happens in memory: **sorted dict** for price, **deque** per price for FIFO
- Background task flushes to DB every **~30ms**
- Individual writes batched into bulk operations
- Read endpoints (order status, order book, prices, balances) still hit PostgreSQL

V2: Results

~90% of B3 (630 orders/second)

Bottleneck: Database Reads

- Read endpoints are **28% of traffic** and they all still hit PostgreSQL
- Under load, DB queries slow down — reads pile up
- Reads become the bottleneck, not writes

V3: Everything in Memory, Database for Durability

```
Client → FastAPI → Full In-Memory State
                   (matching + reads + balances + prices)
                          ↓
                   background flush every ~30ms
                          ↓
                   PostgreSQL (only for recovery)
```

- All state lives in memory: order book, prices, balances, trade history
- Background task flushes changes to DB every **~30ms**
- DB only used on startup (to reload state) and as fallback for old closed orders
- Open orders + recent closed orders in memory

V3: Results

~300% of B3 (2100 orders/second)

Bottleneck: CPU (Python)

- No database in the hot path — all operations happen in memory
- Every order still goes through one Python process, one at a time
- A single CPU core and the language Python are now the bottleneck

Trade-offs

Trade-off	What it means	How to fix it
Crash risk	Lose ~30ms of data on crash (~21 orders, ~4 trades)	Write-ahead log
Memory	Full B3 day (~20M orders + 4M trades) ≈ ~14 GB	Evict completed and expired orders after flush to DB
Single core	One Python process handles all symbols	Split symbols across multiple servers

What I Would Do Next

1. **Add a write-ahead log** — Log every order to disk before confirming
2. **Rewrite in Rust** — Architecture is right, language is the bottleneck
3. **Split by symbol** — Distribute symbols across servers, balanced by trading volume

Summary

Version	Architecture	Capacity	Bottleneck
V1	Everything in the Database	**~40%**	Database locks
V2	Match in Memory, Read from Database	**~90%**	Database reads
V3	Everything in Memory, Database for Durability	**~300%**	CPU (Python)