The making of an Ultra Low Latency Trading System With Java and Golang

Yucong Sun
Staff Software Engineer

Jonathan Ting
Senior Software Engineer
Exchange @ Coinbase

- Takeaways
  - General architecture of an Exchange
  - State of the Art
  - Learnings from optimizing the legacy system
Planetary view of an Exchange

Most users would/should not interact with an Exchange directly

User → Brokerage Product → API → Market Makers / Trading Firm → Market Data → Exchange
Orbital view of an Exchange

Order Management System:
Balance, Risk, Margin/Liquidations

Matching Engine: Order book

API: FIX, HTTP
MarketData: FIX, Websocket

Hot path: Balance check, Order Matching
Warm path: Settlement
Auxiliary: Market Data Feed
Assembly Lines of a Exchange

Trading System Process 1

Trading System Process 2

Order Order Order

Order Order Order

Trade Event Trade Event Trade Event

Trade Event Trade Event Trade Event
Exhibit A: Coinbase Derivatives Exchange

https://www.coinbase.com/derivatives
Trading System Logic Isn’t Complex

Hot Path

Submit & match incoming orders against resting orders (‘book’)

Public - no complex trading relationships

Other logic (timers, admin requests, state)

Affect trading logic, so want to be sequenced with any other events

Trading system assigns IDs to state

Single threaded
Trading System as **Deterministic State Machine**

\[ \text{State}_0 + \text{Input}_0 \Rightarrow \text{State}_1 + \text{Output}_1 \, \text{ALWAYS} \]

Can snapshot/restore/replay to get to live state

**Determinism is Tricky!**
- Data Structure Iteration
- No randomness
- Behavior changes
  - Old input ⇒ Old behavior
  - Feature flagging

Equivalent
Fault Tolerance with RAFT

Aeron Cluster
High performance RAFT implementation

App has to be deterministic & single threaded

Consensus batched & pipelined with application

System throughput = 1 / App processing time

What is RAFT? [Visualize it here](#)
Persisted RAFT Log

Cluster persists RAFT log (input) to disk, as per protocol

Aeron Archive API allows for replicating the RAFT log for backup
Replicated RAFT Log

Audit - Upload to cold storage

Logging - Replay & Send to ELK outside hot path

Debugging - Reproduce bugs locally

Fixing - Backfill missing events

Testing - CI/CD replay to avoid regressions
Replicating For Replay

Replicate *Input*, not *Output*

- Hot Path - Multicast output
- Other - Replicate input & fan out

Output larger & unbounded

1 order => potentially cascading set of events
Replicating For Scalability

Binary tree replication

Network Latency bound by log(n)

Bandwidth usage bounded
Entire Hot Path

RTT outliers < 100 μs
RTT medians < 50 μs
Trading System Processing Times ~ 1 μs
300k/s Peak Throughput

1) Parse & validate Order Submit
2) Send request to trading system
3) RAFT Consensus
4) Matching Algorithm
5) Send order events to gateway
6) Translate Order Ack

= 4 Network Hops (~20μs) + Processing
Hardware Environment for CDE

Colocated in datacenter with customers

Commodity hardware

- Intel Optane Drives
  Faster than enterprise SSDs
  We can fsync if needed without too much penalty

- Low Latency Switches
  350ns cut-through forwarding
  Real-time packet capture without latency hit

Isolated NICs for low latency & bulk traffic
Exhibit B:
Onto the (AWS) Cloud
Cloud

Cons
- Less control over hardware environment
- Need to maintain both DC/AWS deployment, toolchain, configs...

Pros
- Codification, Collaboration
- Good enough performance
- Personal environment
Challenge with Compute/Storage

Machine family: t, m, c, r, z , suffixes N, D
- Recommend: https://instances.vantage.sh/

Storage
- EBS vs Instance Storage

Orchestration
- Recommendation: Nomad
Challenge with AWS networking

Is there a good switch on AWS?
- Cut-through: <0.5us
- Store & forward: 5us - 50us
Secrets with AWS Networks

- Understand spine-leaf networking architecture
  - Region, AZ, sub-azs, racks
  - Avoid load balancers
- cluster placement group
  - capacity reservations
- bad apples

Numbers On AWS

RTT outliers < 1 ms

RTT medians < 300 μs

10 x Network Hops (~250μs)

Trading System Processing ~ 1 μs
Exhibit C: Deep Dive on Performance Tuning
Fast Memory Access

Memory Local Data Structures
   Cache locality outweighs O(n)

Primitive Friendly Data Structures
   No Map<Integer>, avoid Boxing/Unboxing

Deserialize from memory directly into primitives

Represent Strings as 2 Longs
   128 bits => 18 7-bit (ascii) | 21 6-bit (alphanumeric) | 25 5-bit (alphabetic) | 32 4-bit (hex)

No Allocation on Hot Path
   Object Pooling
Small Messages

**Simple Binary Encoding**

Byte Alignment Matters
FPGA Deserialization

Order Fields By Size

VarData / Enum / Bitsets at End

Add Padding If Necessary

```xml
<types>
  <enum name="Side" encodingType="uint8">
    <validValue name="BUY">0</validValue>
    <validValue name="SELL">1</validValue>
  </enum>
  <type name="ClientOrderId" primitiveType="char" length="32">
  </type>
</types>

<sbe:message name="Order" id="1">
  <field name="orderId" id="1" type="int64"/>
  <field name="price" id="2" type="int64"/>
  <field name="quantity" id="3" type="int32"/>
  <field name="side" id="4" type="Side"/>
  <field name="clientOrderId" id="5" type="ClientOrderId"/>
</sbe:message>
```
Java Challenges - Warmup

10k function invocations => JIT compilation
Regulated Exchange - Cannot “warm up” our code

Azul Zulu Prime JVM - ReadyNow!
Cache and Persist JIT Profile + Optimizations
Pre-train new releases with multiple replays of PROD logs

Fast initial orders, remove JIT compilation jitter
Java Challenges - Garbage Collection

“Stop The World” GC - All Application Threads Stalled
Java 8 - Concurrent Mark Sweep

Azul Zulu Prime JVM - Pauseless Garbage Collector
Azul C4 Garbage Collector
Network Optimizations

Multicast
  Consensus
  Output to order and market data gateways

Aeron - High Performance Messaging
  Reliable Transport over UDP
  Per-channel settings
    Congestion & Flow Control
    Socket Buffers - # data in flight ideally equal to Bandwidth Delay Product
    MTU - Jumbo Frames (9k) for batching
Network Optimizations

Kernel Bypass
Read from network card directly from user space
Decreases median, drastically reduces outliers
OpenOnload in data center w/ SolarFlare NICs
DPDK in the cloud - Aeron Support (premium)

Aeron point-to-point
Sending as fast as possible on AWS

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Max</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-DPDK</td>
<td>38μs</td>
<td>1897μs</td>
<td>80MB/s</td>
</tr>
<tr>
<td>DPDK</td>
<td>28us</td>
<td>515μs</td>
<td>500MB/s</td>
</tr>
</tbody>
</table>
Medians Good, Outliers Spiky

Weeks Before Launch
OS Scheduling Delay / Context Switches

How are CPU cycles not running your hot threads?

/proc/sched_debug - task running time per CPU

<table>
<thead>
<tr>
<th>runnable tasks</th>
<th>task</th>
<th>PID</th>
<th>tree-key</th>
<th>switches</th>
<th>prio</th>
<th>wait-time</th>
<th>sum-exec</th>
<th>sum-sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>cpu0/0</td>
<td>68</td>
<td>12.05</td>
<td>2351</td>
<td>23</td>
<td>0.000000</td>
<td>0.0388</td>
<td>0.000000</td>
</tr>
<tr>
<td>5</td>
<td>migrate/0</td>
<td>20</td>
<td>483.07</td>
<td>29277</td>
<td>20</td>
<td>0.000000</td>
<td>2.0850</td>
<td>0.000000</td>
</tr>
<tr>
<td>5</td>
<td>ksoftirqd/0</td>
<td>72</td>
<td>5764620364.7583</td>
<td>256</td>
<td>256</td>
<td>0.000000</td>
<td>0.0560</td>
<td>0.000000</td>
</tr>
<tr>
<td>1</td>
<td>kworker/0/0</td>
<td>72</td>
<td>5850162841.3460</td>
<td>5</td>
<td>5</td>
<td>0.000000</td>
<td>6.9517</td>
<td>0.000000</td>
</tr>
<tr>
<td>1</td>
<td>kworker/0/0</td>
<td>72</td>
<td>4.046563</td>
<td>33</td>
<td>33</td>
<td>0.000000</td>
<td>6.2756</td>
<td>0.000000</td>
</tr>
<tr>
<td>1</td>
<td>kworker/0/0</td>
<td>72</td>
<td>6.046563</td>
<td>33</td>
<td>33</td>
<td>0.000000</td>
<td>6.2756</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

/proc/interrupts - per CPU hardware interrupt #

<table>
<thead>
<tr>
<th>interrupt</th>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>CPU 3</th>
<th>CPU 4</th>
<th>CPU 5</th>
<th>CPU 6</th>
<th>CPU 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
</tr>
<tr>
<td>2</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
</tr>
<tr>
<td>3</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

/proc/schedstat

<table>
<thead>
<tr>
<th>time on cpu</th>
<th>time on runqueue</th>
<th># time slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>4200925624037</td>
<td>12872240906155</td>
<td>780539850</td>
</tr>
<tr>
<td>420096662712</td>
<td>12872240906155</td>
<td>780547937</td>
</tr>
<tr>
<td>4201007606214</td>
<td>12872323980891</td>
<td>780556132</td>
</tr>
<tr>
<td>4201046361274</td>
<td>12872441023508</td>
<td>780564249</td>
</tr>
</tbody>
</table>

perf - get thread runtime or counts individually on a given CPU

# perf record -e "sched.sched_stat_runtime" -C [core id]

# perf script | awk '{print $1}' | sort | uniq -c

15 kworker/3:1-H-kb
1 kworker/3:2-cgr
3 perf
1 rcu_sched
12356 sender

/proc/softirqs - per CPU hardware interrupt #

<table>
<thead>
<tr>
<th>softirq</th>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>CPU 3</th>
<th>CPU 4</th>
<th>CPU 5</th>
<th>CPU 6</th>
<th>CPU 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
</tr>
<tr>
<td>2</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
</tr>
<tr>
<td>3</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
<td>413.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Recommendation: Netdata

a nice visual holistic view of the system

per-cpu interrupts/softirqs/utilization

network, memory, disk, filesystem
OS Scheduling

Pin hot threads to hardcoded CPUs (taskset, sched_setaffinity)
   Prevents context switching & cache misses

Isolate hot CPUs or prioritize threads (ISOLCPUS, taskset, cpusets, nice, chrt)
   Prevent other user threads from taking CPU time
   Busy-spin hot threads to monopolize CPU (and for polling)

Set affinities to hardware interrupts, kernel workqueues, etc.
   Hardware interrupts - use tuna, or set /proc/irq/<irq#>/smp_affinity
   Softirq kernel params - rcu_nocbs, nohz_full
Other Tuning

NUMA locality
- If you have multiple CPU sockets, one is closer to NIC and memory
- Layout matters - lock hot threads to that CPU / Memory NUMA node

Hyperthreading
- Turn it off (or isolate corresponding logical CPU)
- More available L1/L2 cache without it
Exhibit D: Apply the learnings to improve The Legacy System

Where the real fun begins...
Fun with MicroServices

Solution: Another dashboard???
Life of an request

Tracing an single order placement request from start to finish
Graph
everything!!!

Beware:
- Client side view vs Server Side view
- E2E view vs per-unit view
- Tracing sampling
Happy Path: min/p50
~1200us: Elevated but not that outrageous

Infra Inefficiencies - 1000us -> 600us vs 50us
- Compute/Storage
- Network latency
  - Cross AZ traffic
  - Load balancer
- fsync()s

Per operation cost - 30us vs 1us
- Full native, no warmup issue
- Allocations, Pointers
- Metrics recording / Logging

Do you know how often your datadog metrics call is sending a UDP packet out?
Is it just misplaced fsync()s?

- Batched fsync on Optane or no fsync() here
- Non batched fsync() here

fsync() cost ~500us to 1ms on AWS hardware
Pointer & Memory Allocations In Golang

Heap escape analysis (-gcflags “-m”)
- Sending pointers or values containing pointers to channels.
- Storing pointers or values containing pointers in a slice. like \[*string\].
- Backing arrays of slices that get reallocated because an append would exceed their capacity.
- Calling methods on an interface type

Pass a small struct by value could be 8x faster vs passing by pointer, thus moving it to the heap. (x86_64 has cache line size 64 bytes)

Unhappy Path: p99/max

P99 ~4ms, Max 362ms
WTF is going on...

- GC pause?
- Scheduling delays?
- Non-FIFO behaviors?
Is Golang GC really a issue?

<table>
<thead>
<tr>
<th>SLOs then and now</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2014</strong></td>
</tr>
<tr>
<td>25% of the total CPU</td>
</tr>
<tr>
<td>Heap 2X live heap</td>
</tr>
<tr>
<td>10 ms STW pause every 50 ms</td>
</tr>
<tr>
<td>Goroutines allocation $\propto$ GC assists</td>
</tr>
<tr>
<td><strong>2018</strong></td>
</tr>
<tr>
<td>25% of the CPU during GC cycle</td>
</tr>
<tr>
<td>Heap 2X live heap or max heap</td>
</tr>
<tr>
<td>Two $&lt;500$ $\mu$s STW pauses per GC</td>
</tr>
<tr>
<td>Goroutines allocation $\propto$ GC assists</td>
</tr>
<tr>
<td>Minimal GC assists in steady state</td>
</tr>
</tbody>
</table>

https://go.dev/blog/ismmkeynote

https://tip.golang.org/doc/gc-guide

https://malloc.se/blog/zgc-jdk16

Hint: Goroutine explosion by GRPC

Golang grpc unary requests default to create new goroutine for every request, this cause starvation of any background goroutines, leads to tail latencies

Goroutines:
- runtime.geBgMarkWorker N=95
- google.golang.org/grpc.(*Server).serveStreams.func1_2 N=34041
- github.com/hashicorp/raft.(*raftState).goFunc.func1 N=14
- google.golang.org/grpc/internal/transport.NewServerTransport.func2 N=17
- google.golang.org/grpc.(*Server).handleRawConn.func1 N=17
- github.cbhq.net/engineering/csf/go/csf.(*DefaultSystemManager).AddService.func1 N=6
- github.com/hashicorp/raft.newNetPipeline.dwrap-40 N=4
- github.cbhq.net/mono/repo/pro/trading-engine/engine/internal/replicator.(*Replicator).Run.func1 N=1
- runtime.hexsween N=1
**Hint: Goroutine scheduler delay**

<table>
<thead>
<tr>
<th>Goroutine Name</th>
<th>github.cbhq.net/mono/repo/pro/trading-engine/engine/internal/replicator.(*Replicator).Run.func1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Goroutines</td>
<td>1</td>
</tr>
<tr>
<td>Execution Time</td>
<td>1.52% of total program execution time</td>
</tr>
<tr>
<td>Network Wait Time</td>
<td><a href="download">graph</a></td>
</tr>
<tr>
<td>Sync Block Time</td>
<td><a href="download">graph</a></td>
</tr>
<tr>
<td>Blocking Syscall Time</td>
<td><a href="download">graph</a></td>
</tr>
<tr>
<td>Scheduler Wait Time</td>
<td><a href="download">graph</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goroutine Total</th>
<th>Execution</th>
<th>Network Wait</th>
<th>Sync block</th>
<th>Blocking syscall</th>
<th>Scheduler wait</th>
<th>GC sweeping</th>
<th>GC pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>181</td>
<td>956ms</td>
<td>0ns</td>
<td>8670ms</td>
<td>0ns</td>
<td>373ms</td>
<td>3416μs (0.0%)</td>
<td>230ms (2.3%)</td>
</tr>
</tbody>
</table>
Goroutine is not your good old thread

- Go scheduler
- GOMAXPROCS = num CPUs
- Remember: Only GOMAXPROCS will run at same time
Visualizing how API-FIX works

REMEmBER: Only GOMAXPROCS amount of goroutines will run at any given time
Visualizing how OEGW works

1 RPC = 1 goroutine

Random: Not FIFO
https://github.com/golang/go/issues/31708

TCP Connection

Contended Resource
Visualizing how Trading Engine works

TCP Connection

inputCh

raftChan

ApplyCh

OutCh

Network Poller

TCP Connection

REMEMBER: Only GOMAXPROCS amount of goroutines will run at any given time
Mitigations: spinning important goroutine

```go
select {
    case item <- ch:
        // process item
    }
```

Note: Golang scheduler will force preempt long running go-routines every 10ms

```go
select {
    case item <- ch:
        // process item
    default:
        // busy spinning
        continue
}
```

Challenges:
Can’t spin too much, as you will run out of CPU and cause starvation.

```go
runtime.LockOSThread()
```
Mitigations: Always batch when using channels

```go
select {
    case item <- bufCh:
        items := make([]int, 20)
        items = append(items, item)
    Remaining:
        for i := 0; i < 19; i++ {
            select {
                case item <- bufCh:
                    items = append(items, item)
                default:
                    break Remaining
            }
        }
    // processing items
    default: continue
}
```

First Read

Grab outstanding messages while you are there

Why does this work?
- Avoid scheduler delays
- Better cache locality

Don’t forget spinning!
Realization: Golang is optimized for throughput

Most facilities in Golang Linux introduce an randomness element to optimize for throughput, not latency

- Go encourage you/libraries to spawn adhoc goroutines everywhere
- No goroutine priorities, and scheduler is randomized and job stealing

Writing low latency code in Golang is not easy, but again it’s not easy anywhere else either.

Recommendation: use GRPC in streaming mode, not unary mode!
Is it just misplaced fsync()s?

**FAST**

1. FIX Receive → RAFT
2. RAFT: Balance Check → Order Process → FIX Send
3. Batched fsync on Optane, or no fsync() here

**SLOW**

1. FIX Receive → Balance Check → RAFT: Order Process → FIX Send
2. DB
3. Non batched fsync() here

- fsync() cost ~500us to 1ms on AWS hardware
“let’s add this part or the process step in case we need it”... the most common error of a smart engineer, is to optimize the thing that should not exist....

Elon Musk on Engineering, interviewed by Tim Dodd

Latency Cost Rankings
<1us Kernel syscall overhead
~ 1us optimized application logic cost
~ 5us kernel context switching cost
~ 5us per network hop on LT hardware
~ 25us per network hop on AWS hardware
~ 30us per message unoptimized application logic cost
~ 50us - 100us RT Kernel scheduler delay [0]
~ <100us fsync on Optane
~ 250us golang GC pauses
~ 1ms fsync on AWS Instance Storage
~ N ms non-RT Kernel scheduler delay [0]
~ N to NNms golang scheduler delays

[0] https://bristot.me/files/research/papers/ecrts2020/slides.pdf