SRECON AMERICAS

coinbase

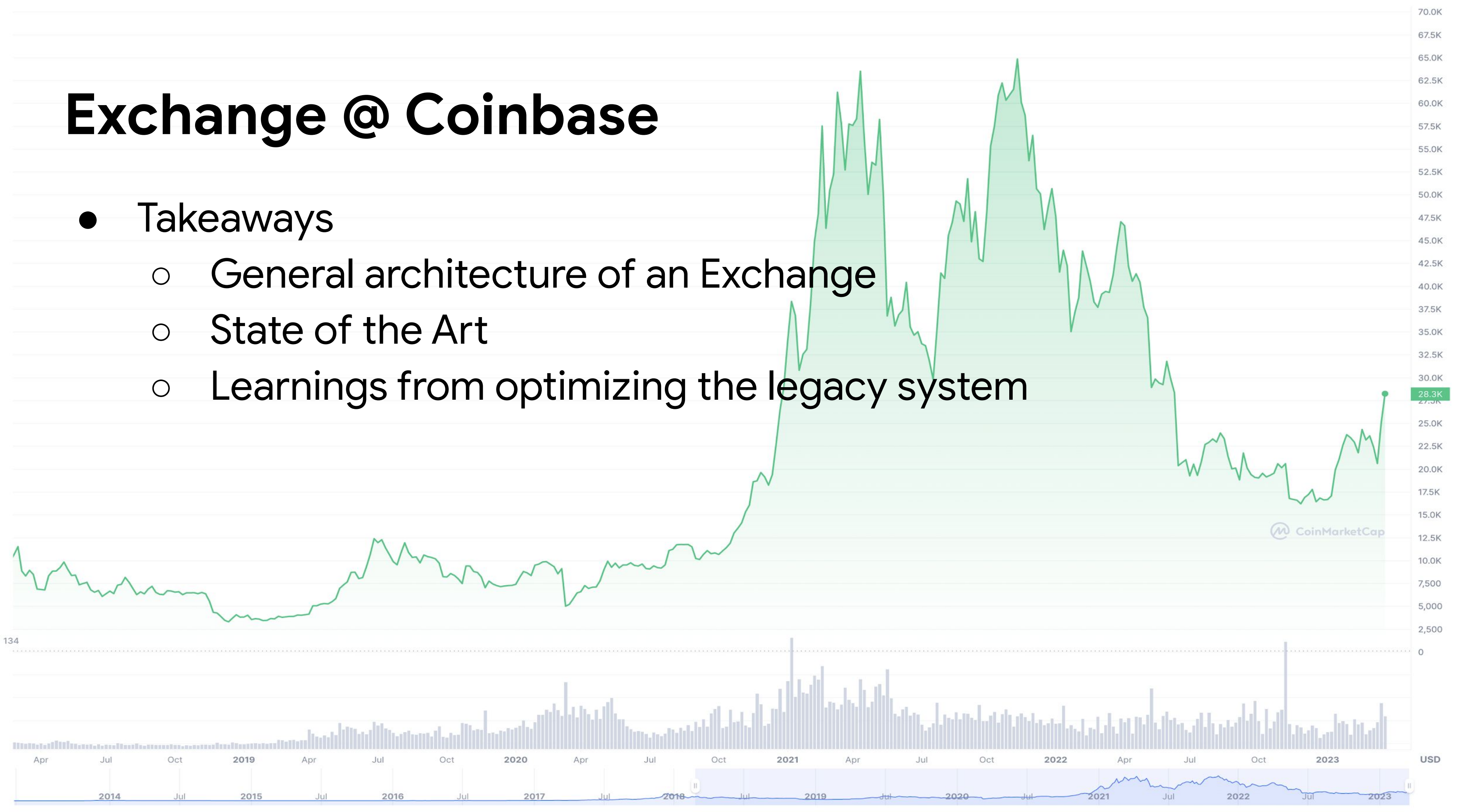
The making of an
Ultra Low Latency Trading System
With Java and Golang

Yucong Sun
Staff Software Engineer

Jonathan Ting
Senior Software Engineer

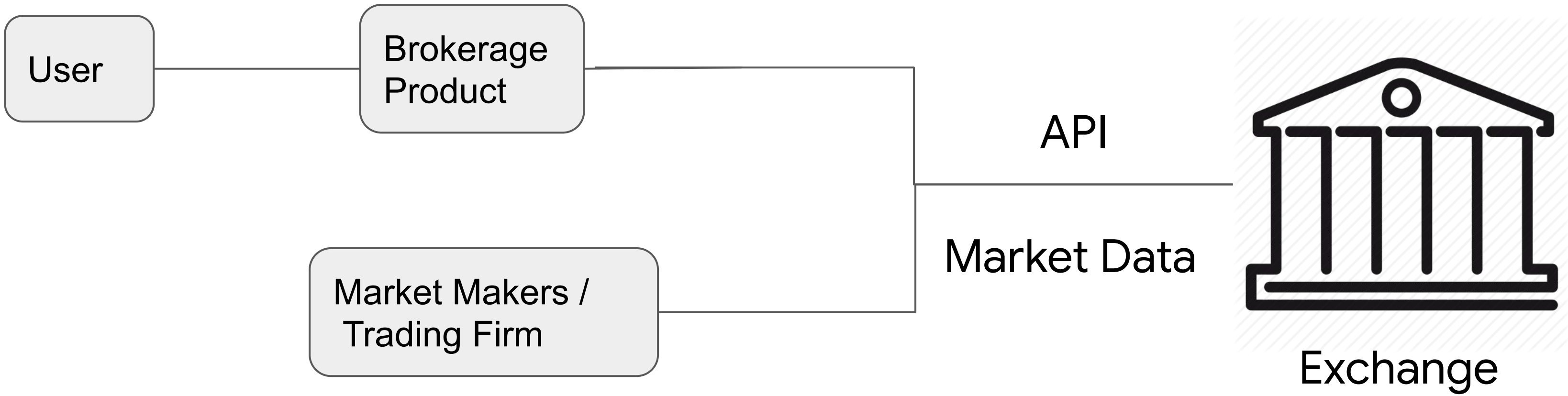
Exchange @ Coinbase

- 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



Orbital view of an Exchange

Order Management System:

Balance, Risk, Margin/Liquidations

Matching Engine: Order book

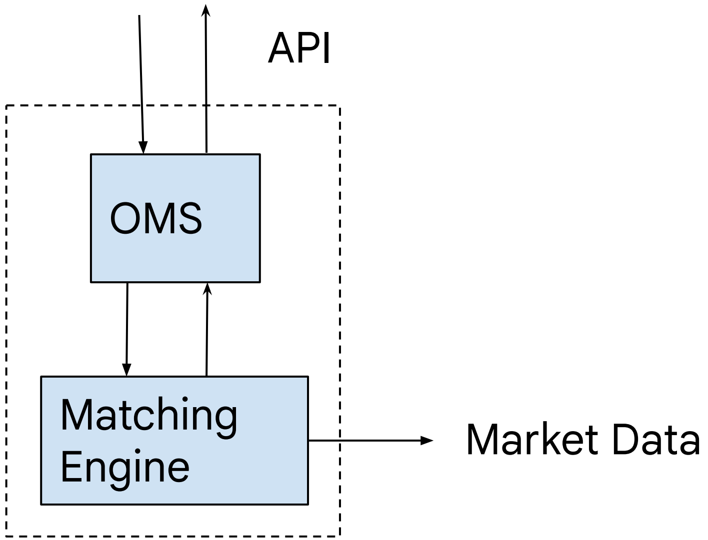
API: FIX, HTTP

MarketData: FIX, Websocket

Hot path: Balance check, Order Matching

Warm path: Settlement

Auxiliary: Market Data Feed



Assembly Lines of a Exchange

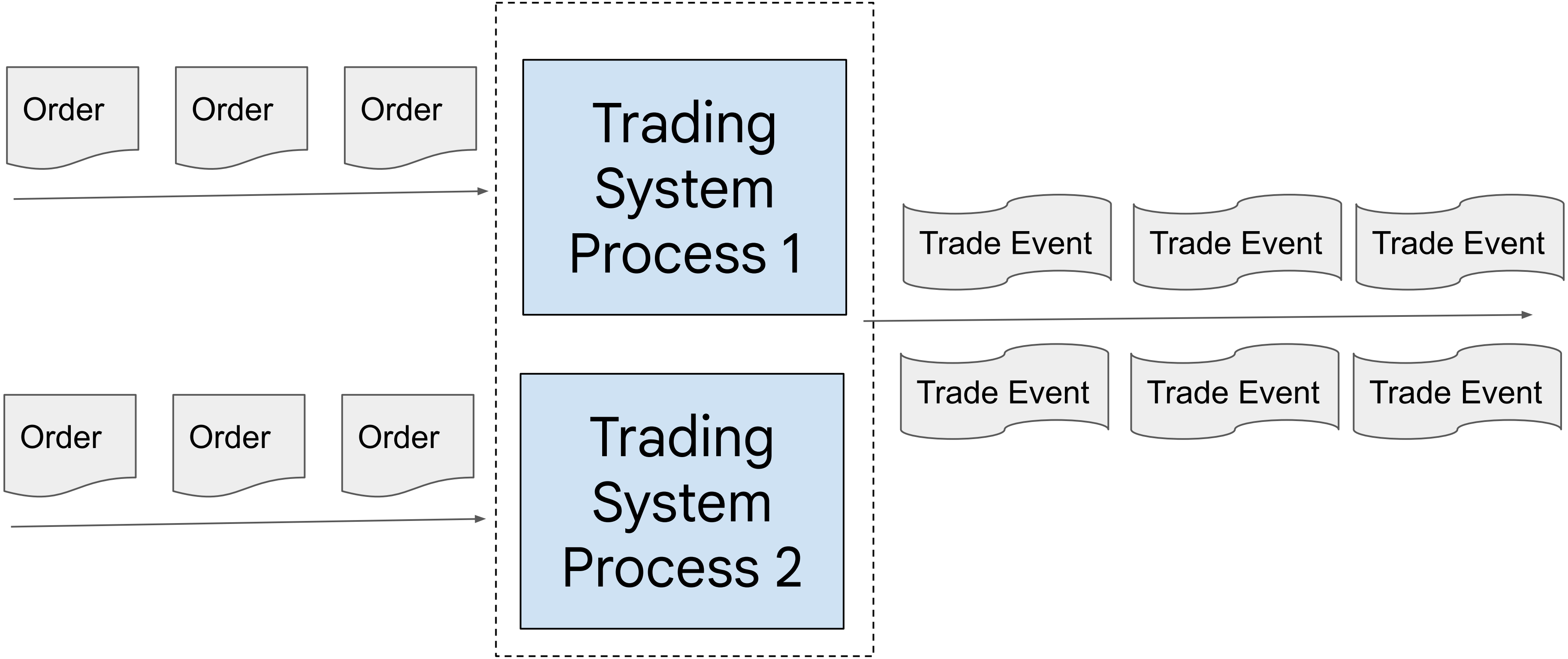


Exhibit A:
Coinbase Derivatives Exchange

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

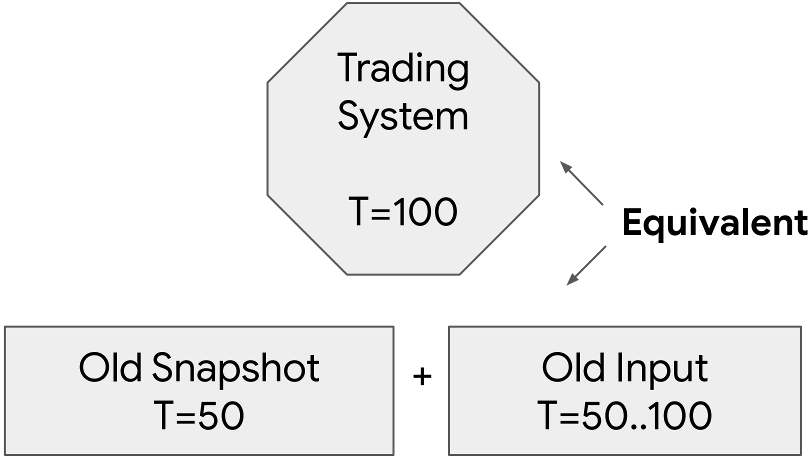
Trading System as **Deterministic State Machine**

$State_0 + Input_0 \Rightarrow State_1 + Output_1$ **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



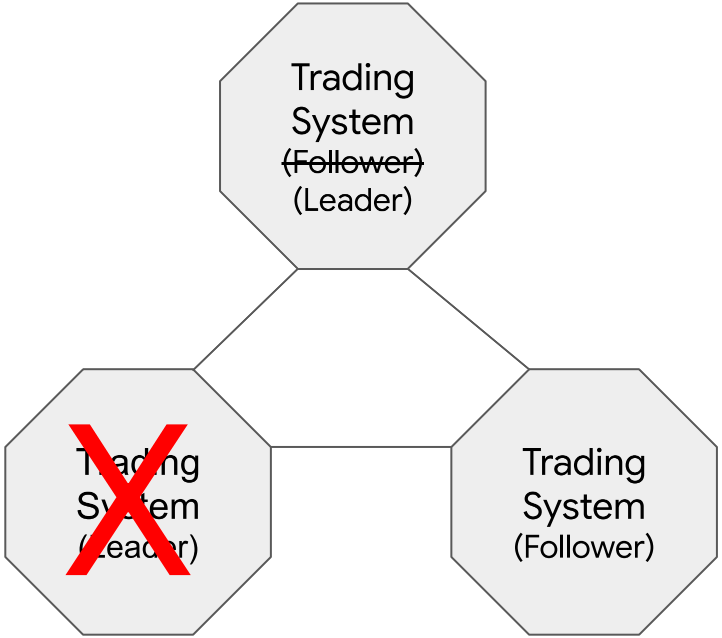
Fault Tolerance with RAFT

Aeron Cluster
High performance RAFT implementation

App has to be deterministic & single threaded

Consensus batched & pipelined with application

System throughput = 1 / App processing time

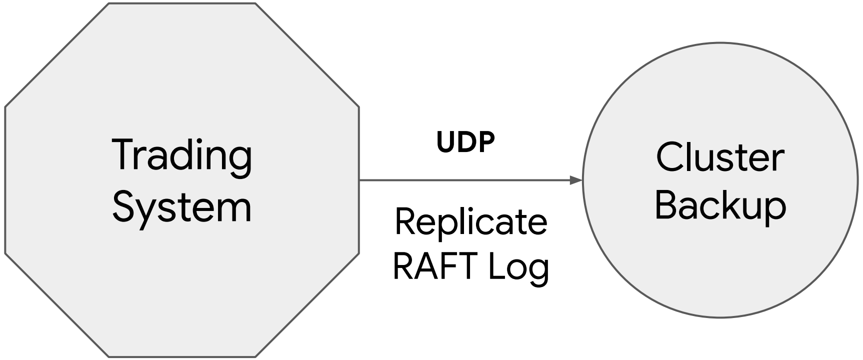


What is RAFT? Visualize it here

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

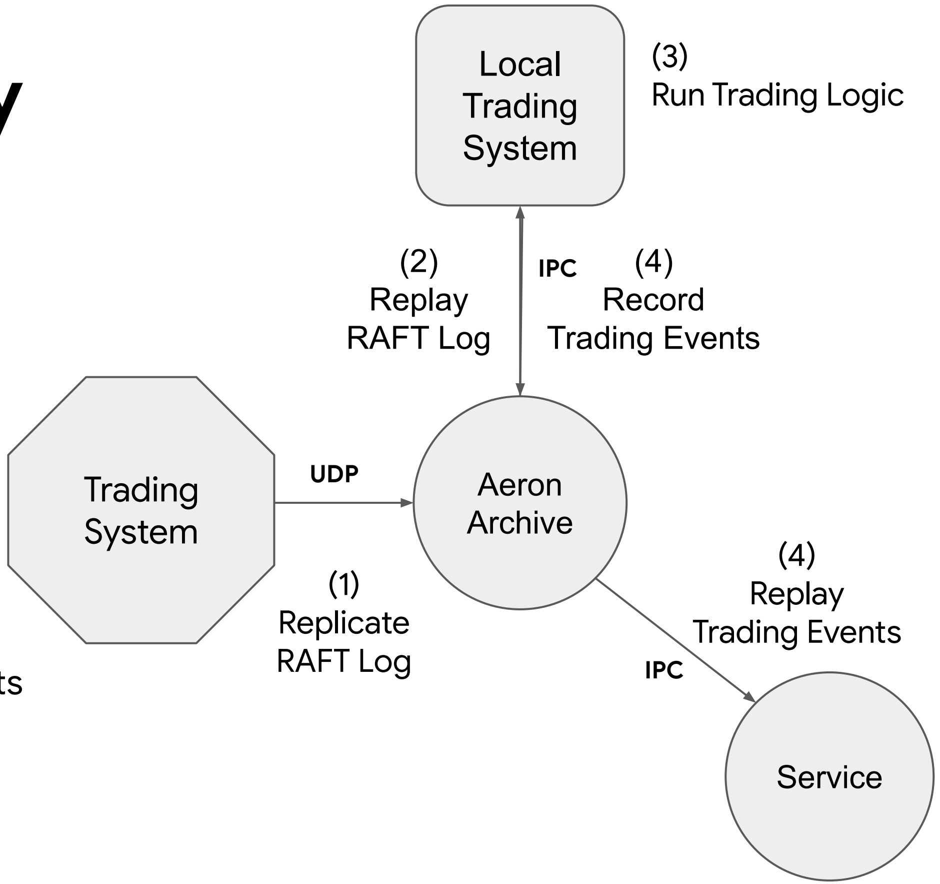
Replicating For Replay

Replicate *Input*, not *Output*

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

Output larger & unbounded

- 1 order => potentially cascading set of events

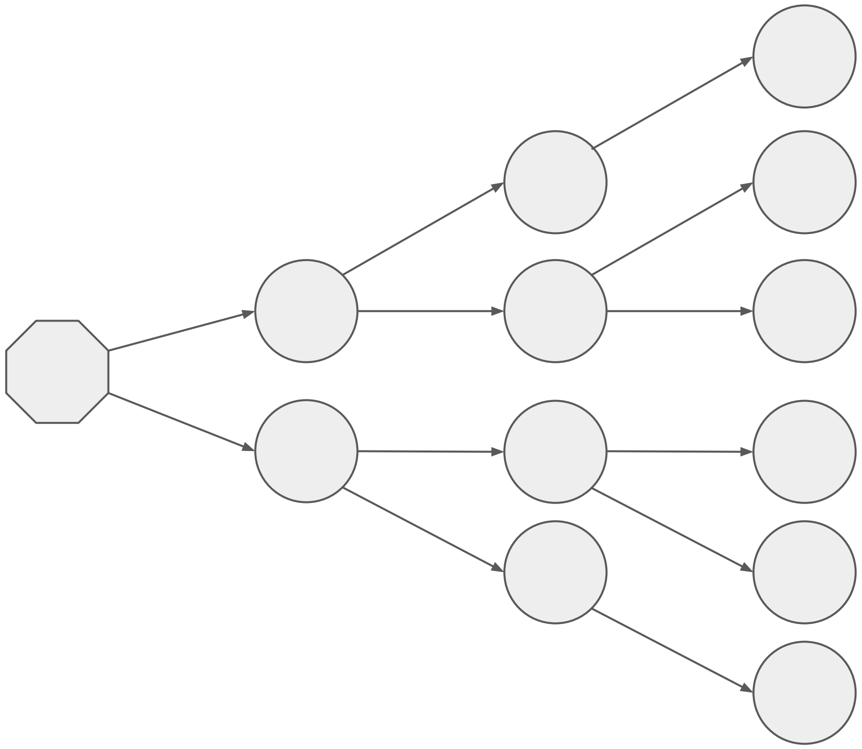


Replicating For Scalability

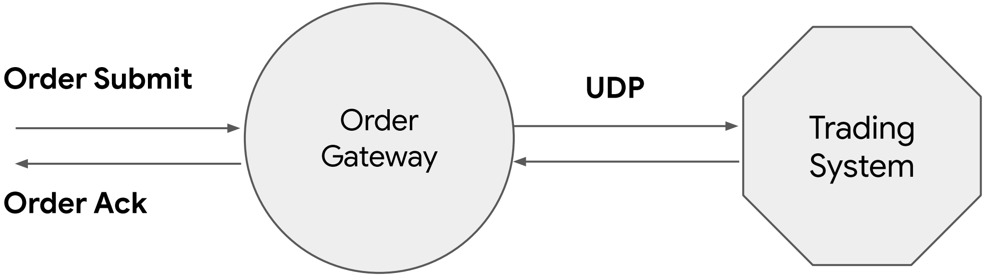
Binary tree replication

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

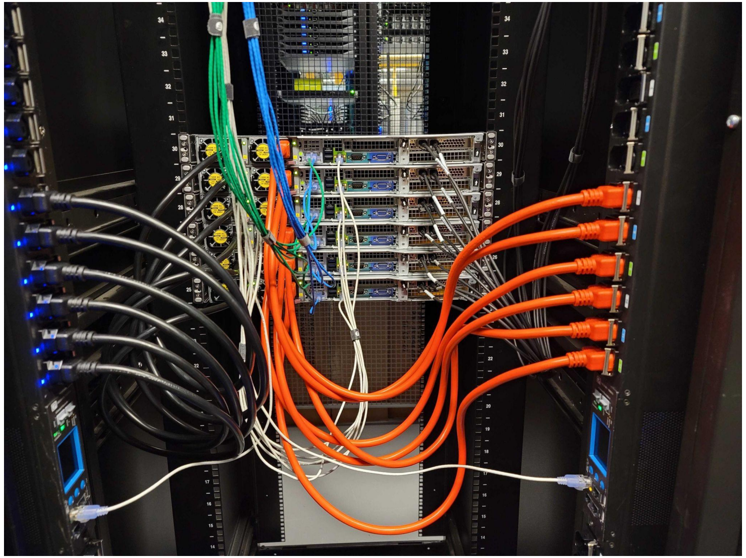


Exhibit B:

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?

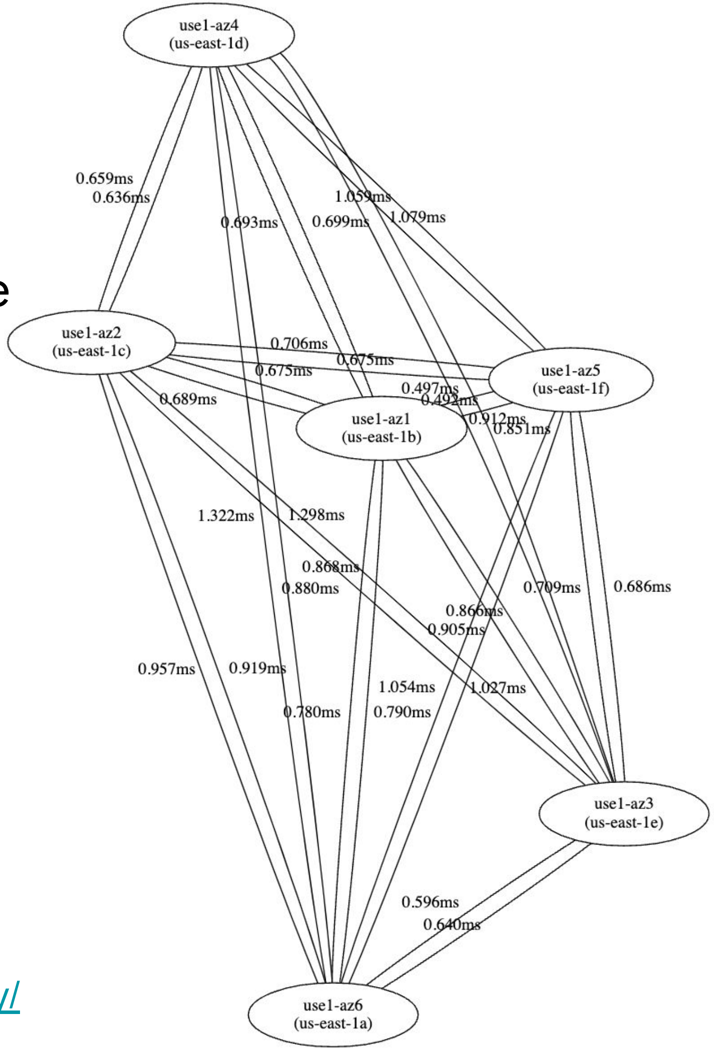
- Cut-through: <0.5us
- Store & forward: 5us - 50us

Secrets with AWS Networks

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

Availability Zone	Availability Zone ID
us-east-1a	use1-az4

https://www.xkyle.com/Measuring-AWS-Region-and-AZ-Latency/



Numbers On AWS

RTT outliers < 1 ms

RTT medians < 300 µs

Trading System Processing ~ 1 µs

10 x Network Hops (~250µs)

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

```
<types>
  <enum name="Side" encodingType="uint8">
    <validValue name="BUY">0</validValue>
    <validValue name="SELL">1</validValue>
  </enum>

  <type name="ClientOrderId" primitiveType="char" length="32">
</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

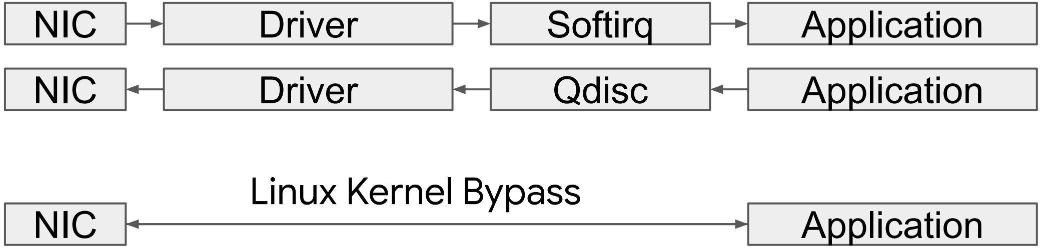
Azul C4 Garbage Collector

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



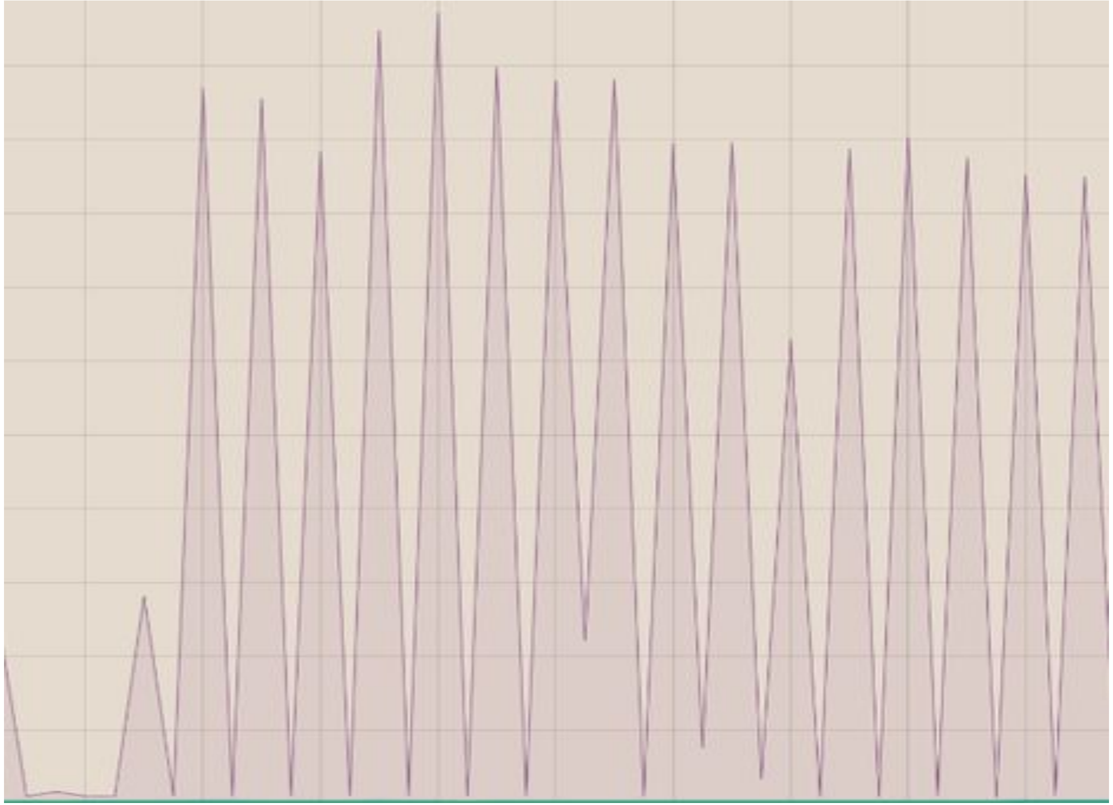
Kernel Bypass

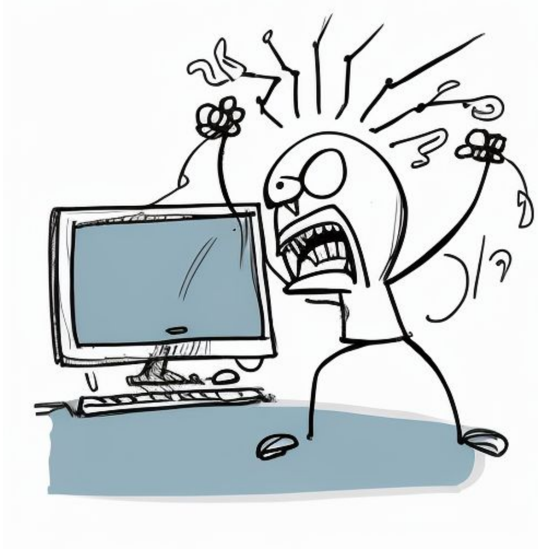
- Read from network card directly from user space
- Decreases median, drastically reduces outliers
- OpenOnload in data center w/ SolarFlare NICs
- DPDK in the cloud - Aeron Support (premium)

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

	Mean	Max	Throughput
non-DPDK	38µs	**1897µs**	80MB/s
DPDK	28us	**515µs**	500MB/s

Medians Good, Outliers Spiky



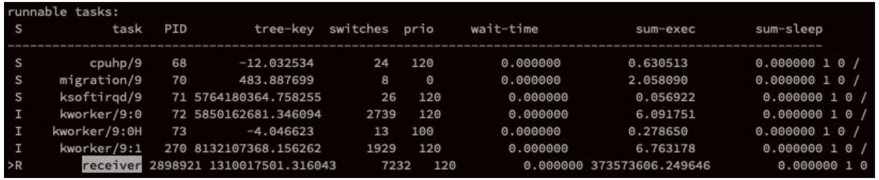


Weeks Before Launch

OS Scheduling Delay / Context Switches

How are CPU cycles are not running your hot threads?

/proc/sched_debug - task running time per CPU



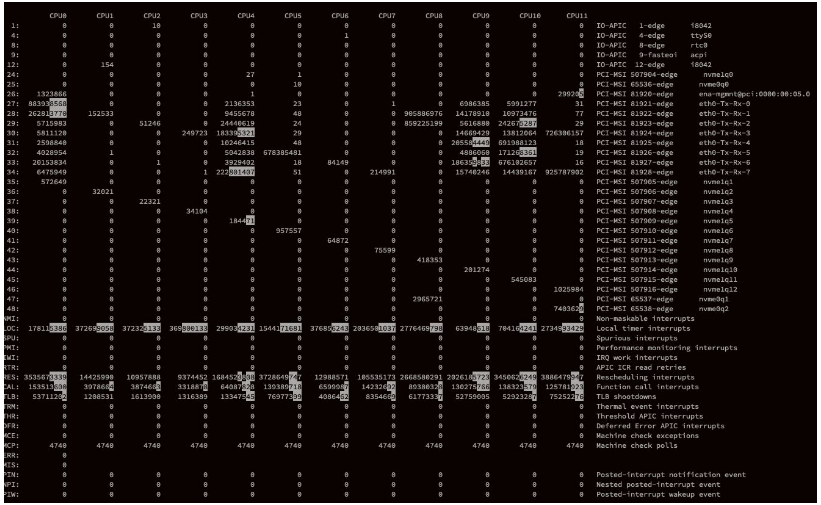
```
runnable tasks:
 S           task   PID         tree-key  switches  prio     wait-time             sum-exec        sum-sleep
-------------------------------------------------------------------------------------------------------------
 S        cpuhp/9    68       -12.032534        24   120      0.000000             0.630513       0.000000 1 0 /
 S    migration/9    70       483.887699         8     0      0.000000             2.058090       0.000000 1 0 /
 S    ksoftirqd/9    71 5764180364.758255       26   120      0.000000             0.056922       0.000000 1 0 /
 I    kworker/9:0    72 5850162681.346094     2739   120      0.000000             6.091751       0.000000 1 0 /
 I   kworker/9:0H    73        -4.046623        13   100      0.000000             0.278650       0.000000 1 0 /
 I    kworker/9:1   270 8132107368.156262     1929   120      0.000000             6.763178       0.000000 1 0 /
>R       receiver 2898921 1310017501.316043     7232   120      0.000000 373573606.249646       0.000000 1 0
```

/proc/<tid>/schedstat

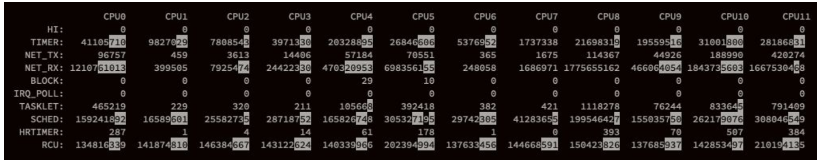
time on cpu	time on runqueue	# time slices
4200925624037	12872240906155	780539850
4200966662712	12872278642290	780547937
4201007606214	12872323980891	780556132
4201046361274	12872441023508	780564249

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

/proc/interrupts - per CPU hardware interrupt



/proc/softirqs - per CPU hardware interrupt



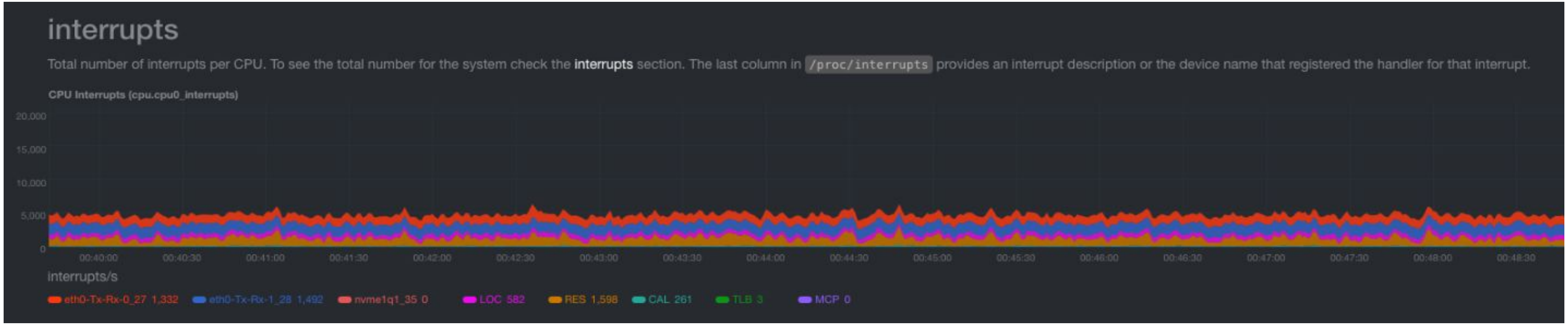
	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7	CPU8	CPU9	CPU10	CPU11
HI:	0	0	0	0	0	0	0	0	0	0	0	0
TIMER:	41105710	9827029	7808543	39971330	20328895	26846606	5376952	1737338	21698319	19559516	31001800	28186831
NET_TX:	96757	459	3613	14406	57184	70551	365	1675	114367	44926	188990	420274
NET_RX:	1210761013	399505	7925474	24422330	470320953	698356155	248058	1686971	1775655162	466064054	1843735603	1667530468
BLOCK:	0	0	0	0	29	10	0	0	0	0	0	0
IRQ_POLL:	0	0	0	0	0	0	0	0	0	0	0	0
TASKLET:	465219	229	320	211	105668	392418	382	421	1118278	76244	833645	791409
SCHED:	159241892	16589601	25582735	28718752	165826748	305327195	29742305	41283655	199546427	155035750	262179076	308046549
HRTIMER:	287	1	4	14	61	178	1	0	393	70	507	384
RCU:	134816339	141874810	146384667	143122624	140339966	202394994	137633456	144668591	150423826	137685937	142853497	210194135

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

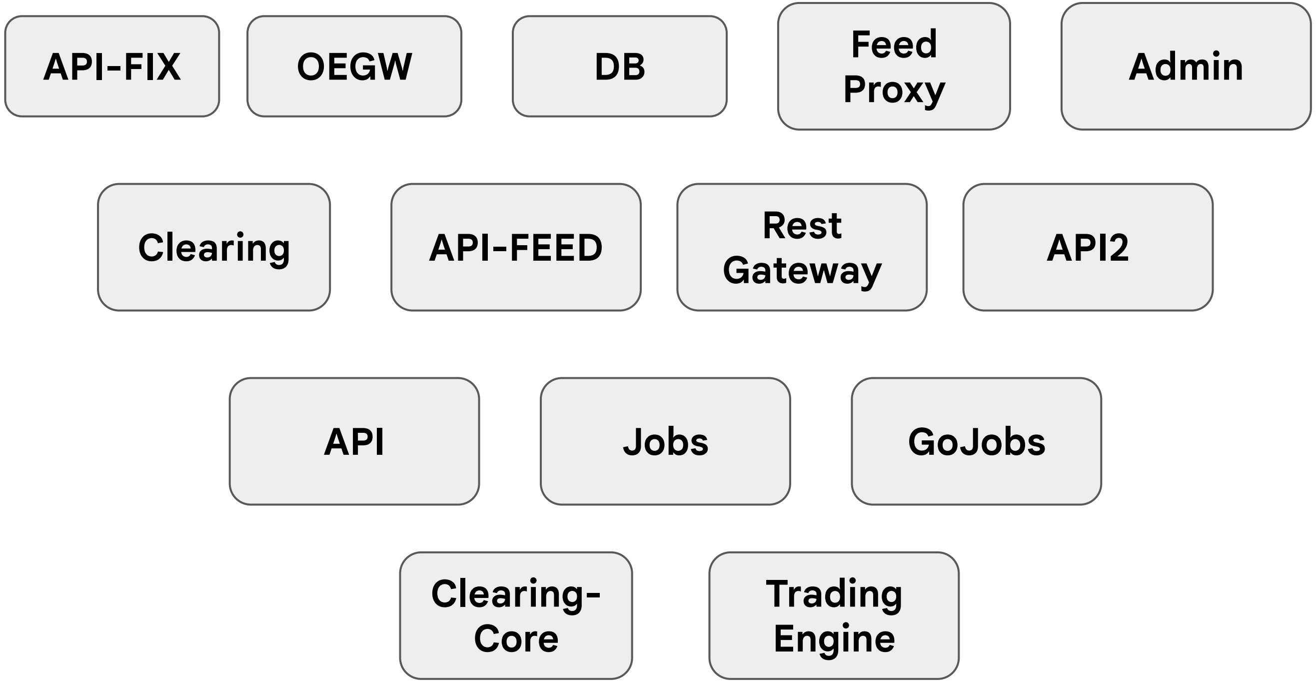
Hyperthreading

- 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...

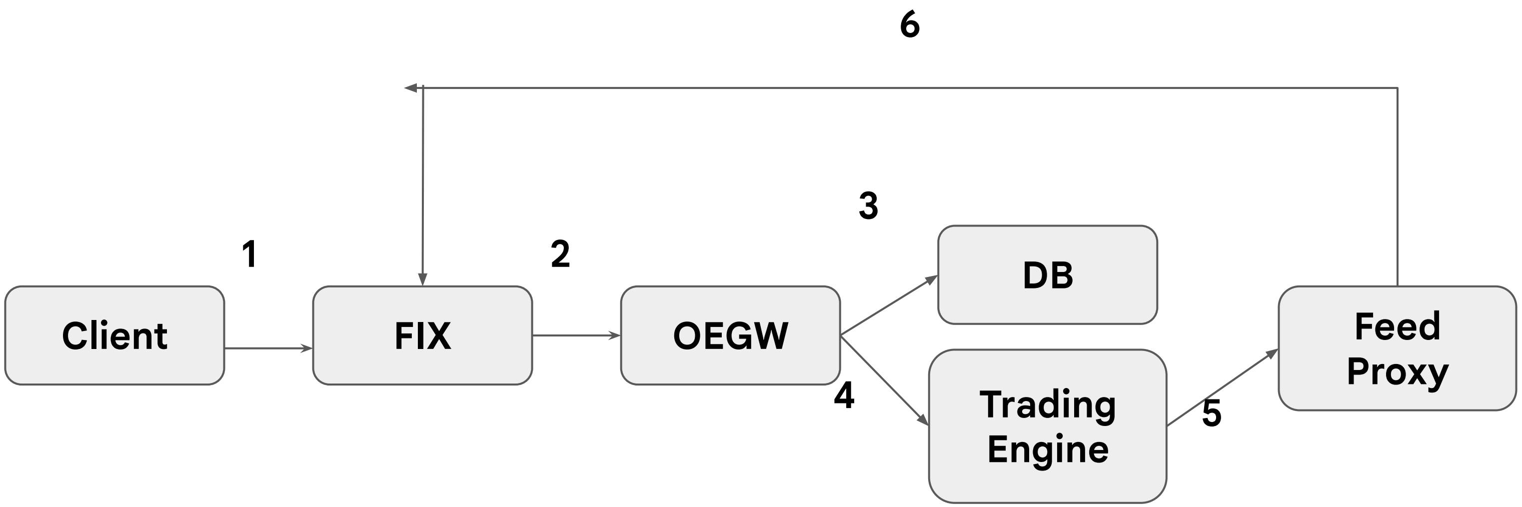
Fun with MicroServices

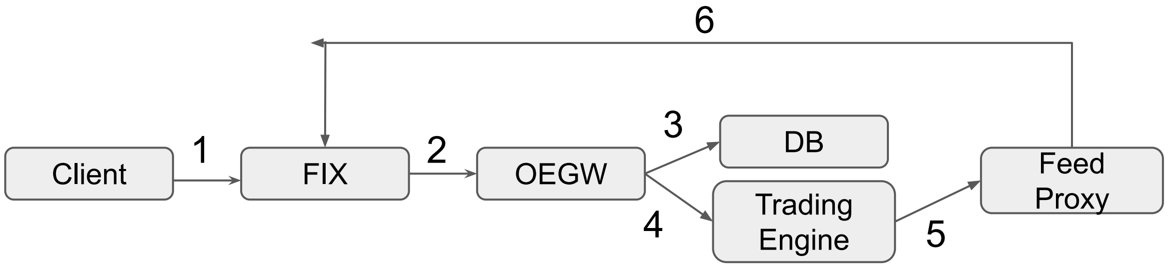


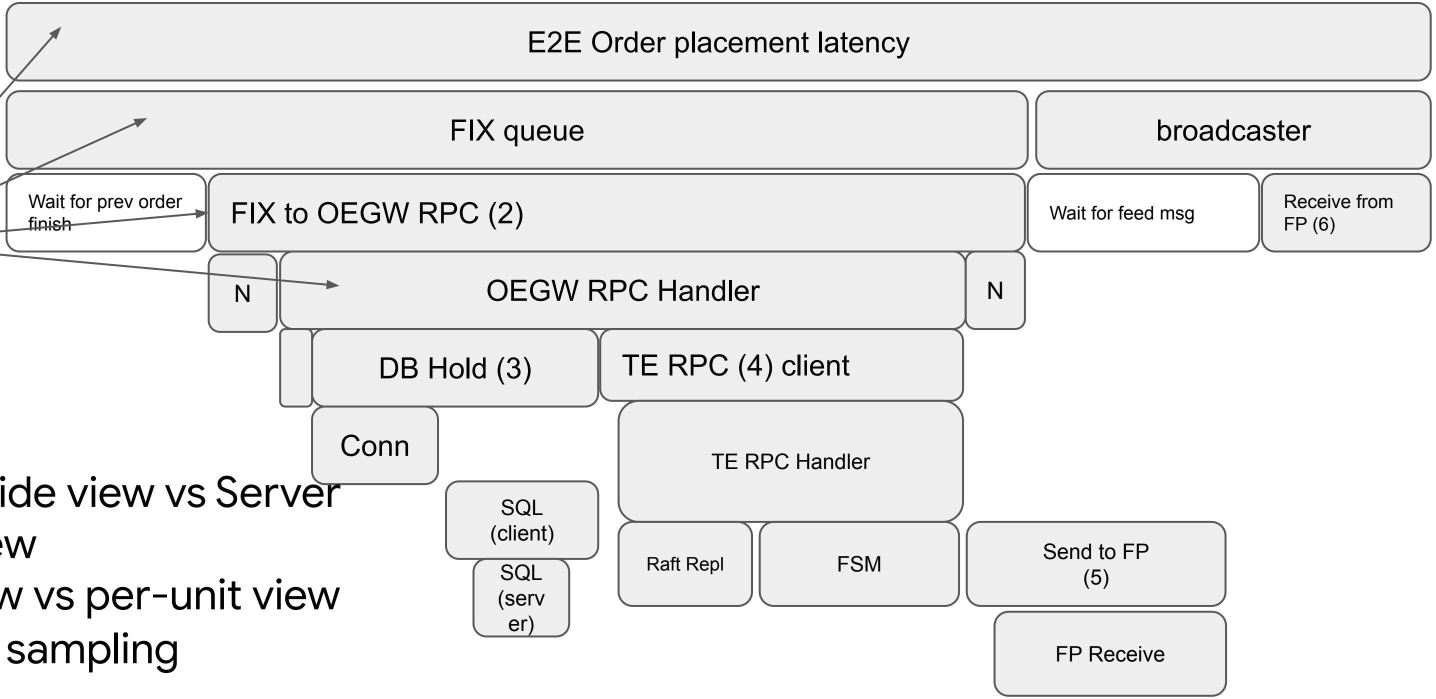
Solution: Another dashboard???

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

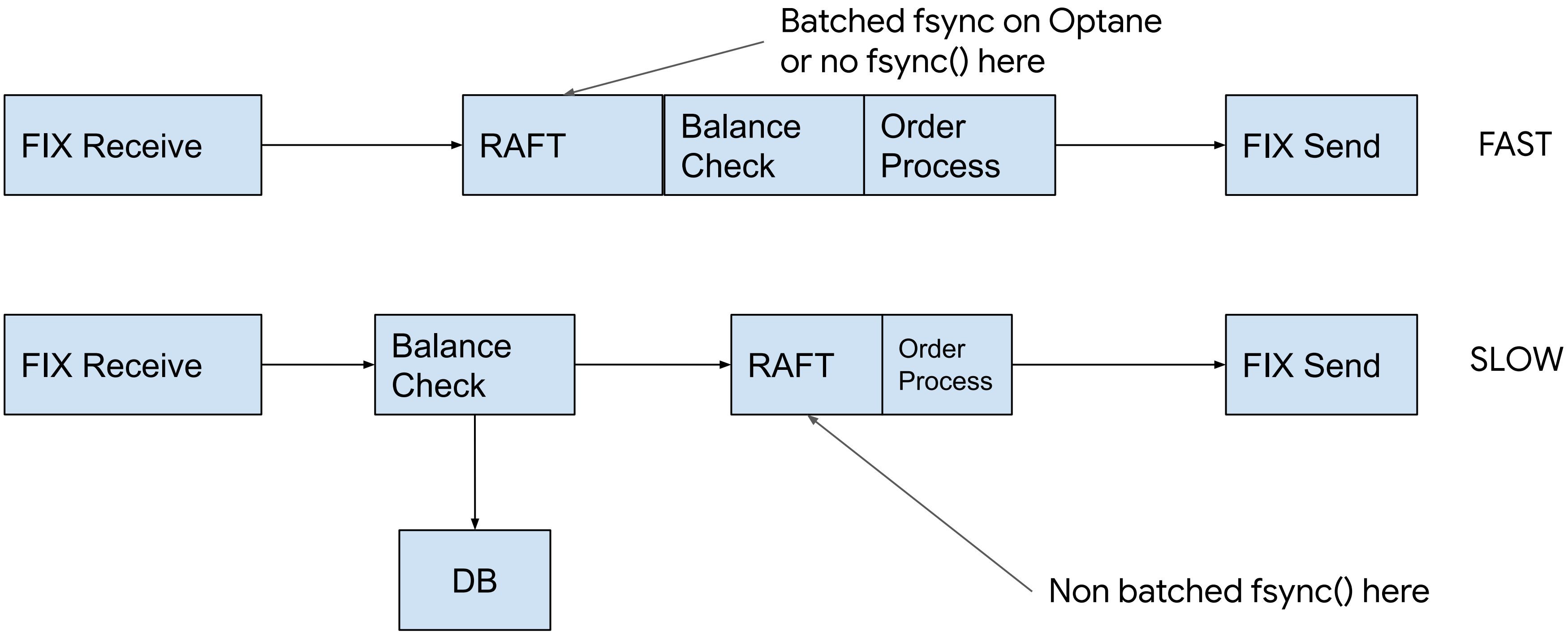
- Compute/Storage
- Network latency
  - Cross AZ traffic
  - Load balancer
- fsync()s

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?



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)

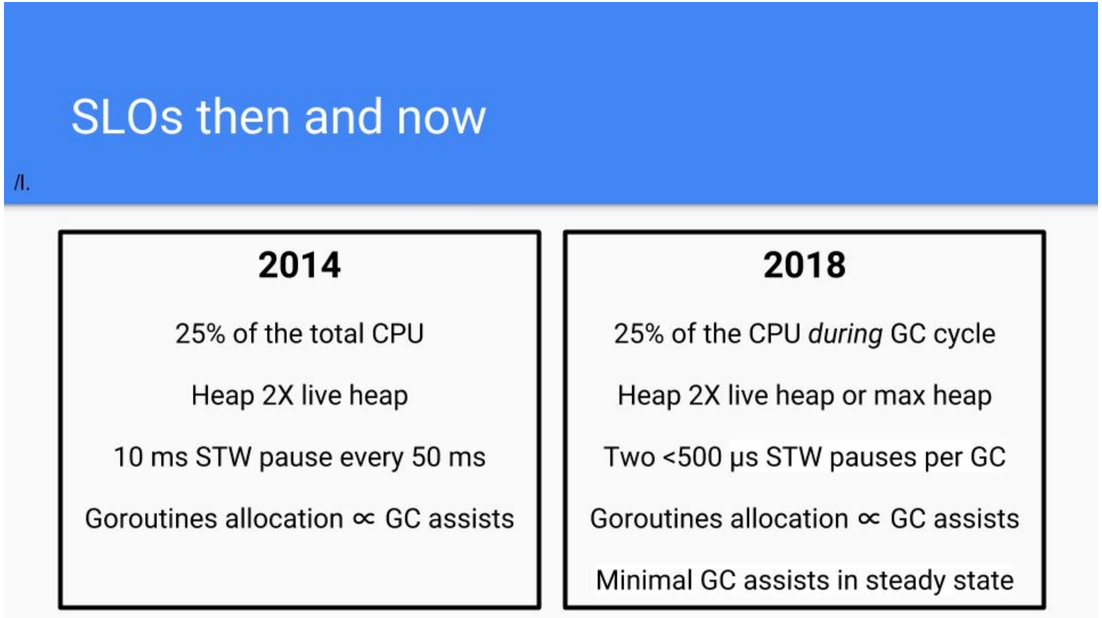
https://segment.com/blog/allocation-efficiency-in-high-performance-go-services/

Unhappy Path: p99/max

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

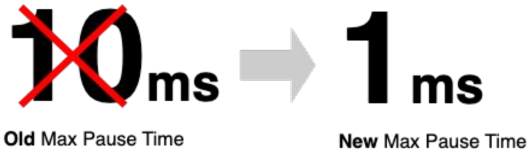
- GC pause?
- Scheduling delays?
- Non-FIFO behaviors?

Is Golang GC really a issue?



https://go.dev/blog/ismmkeynote

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



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

https://www.azul.com/sites/default/files/images/c4_paper_acm.pdf

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.gcBgMarkWorker 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.bgsweep N=1

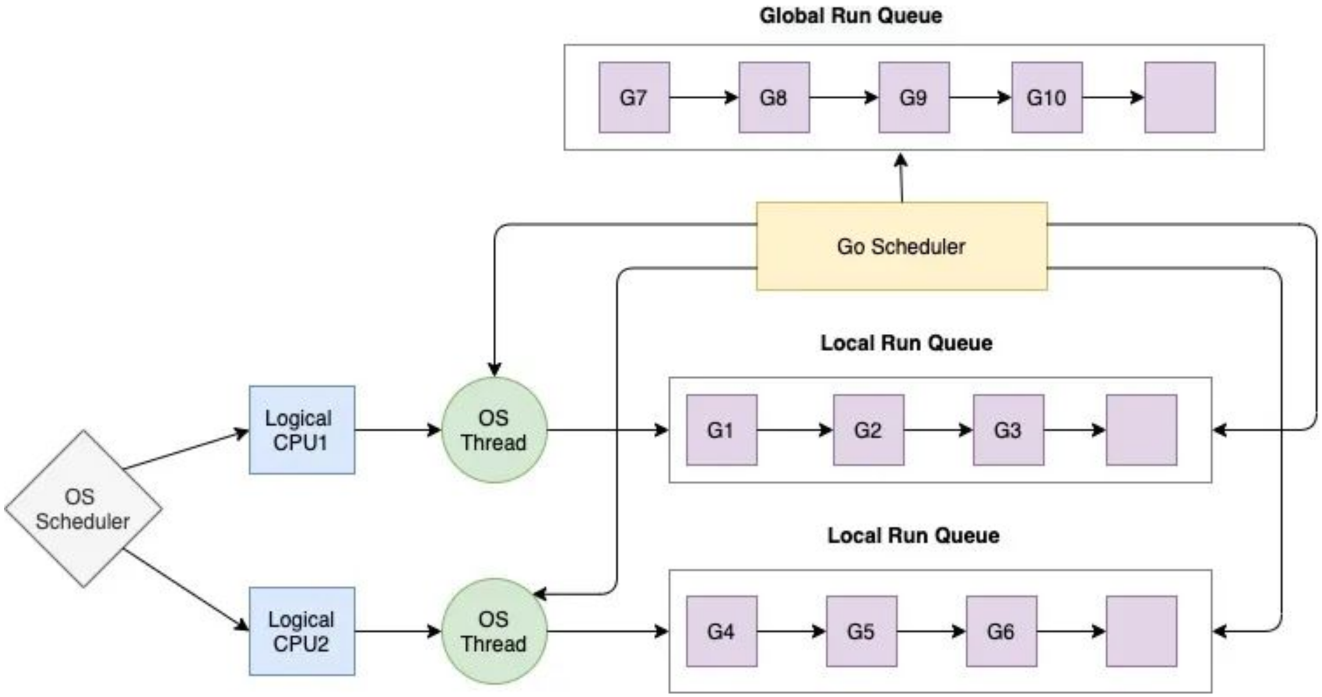
34041
goroutines???

Hint: Goroutine scheduler delay

Goroutine Name:	github.cbhq.net/mono/repo/pro/trading-engine/engine/internal/replicator.(*Replicator).Run.func1
Number of Goroutines:	1
Execution Time:	1.52% of total program execution time
Network Wait Time:	graph(download)
Sync Block Time:	graph(download)
Blocking Syscall Time:	graph(download)
Scheduler Wait Time:	graph(download)

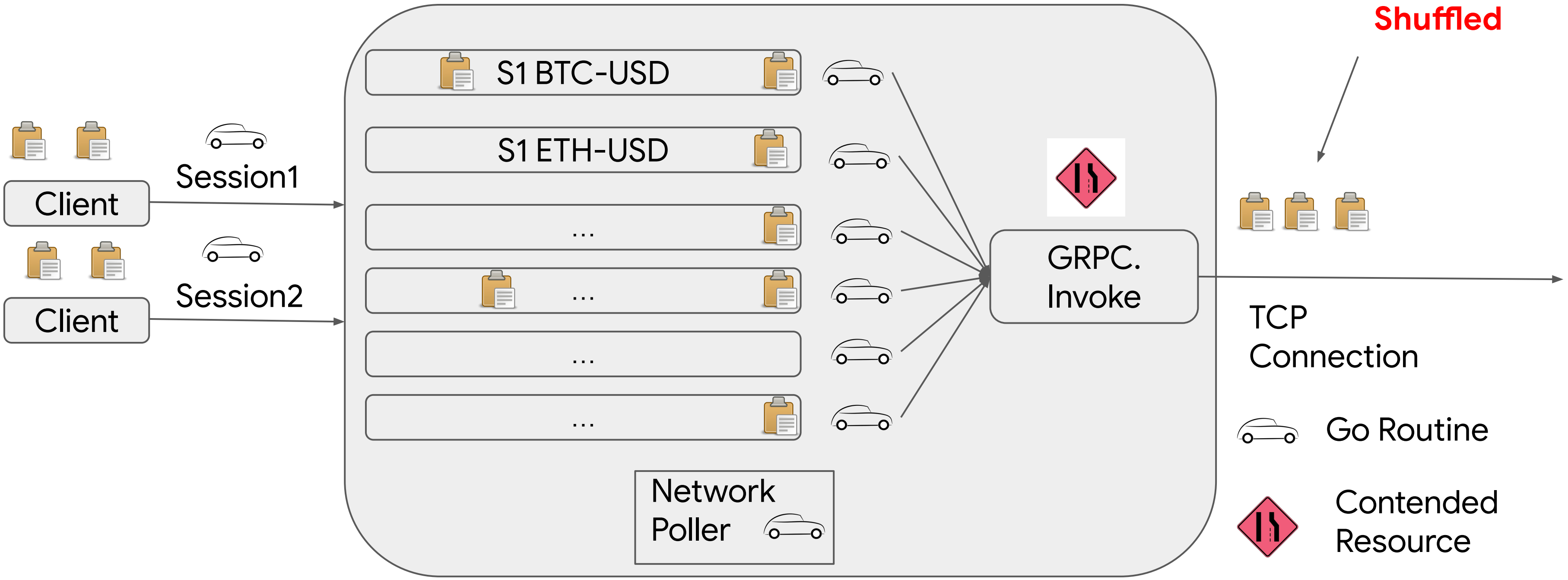
Goroutine	Total		Execution	Network wait	Sync block	Blocking syscall	Scheduler wait	GC sweeping	GC pause
181	10s		956ms	0ns	8670ms	0ns	373ms	3416μs (0.0%)	230ms (2.3%)

Goroutine is not your good old thread



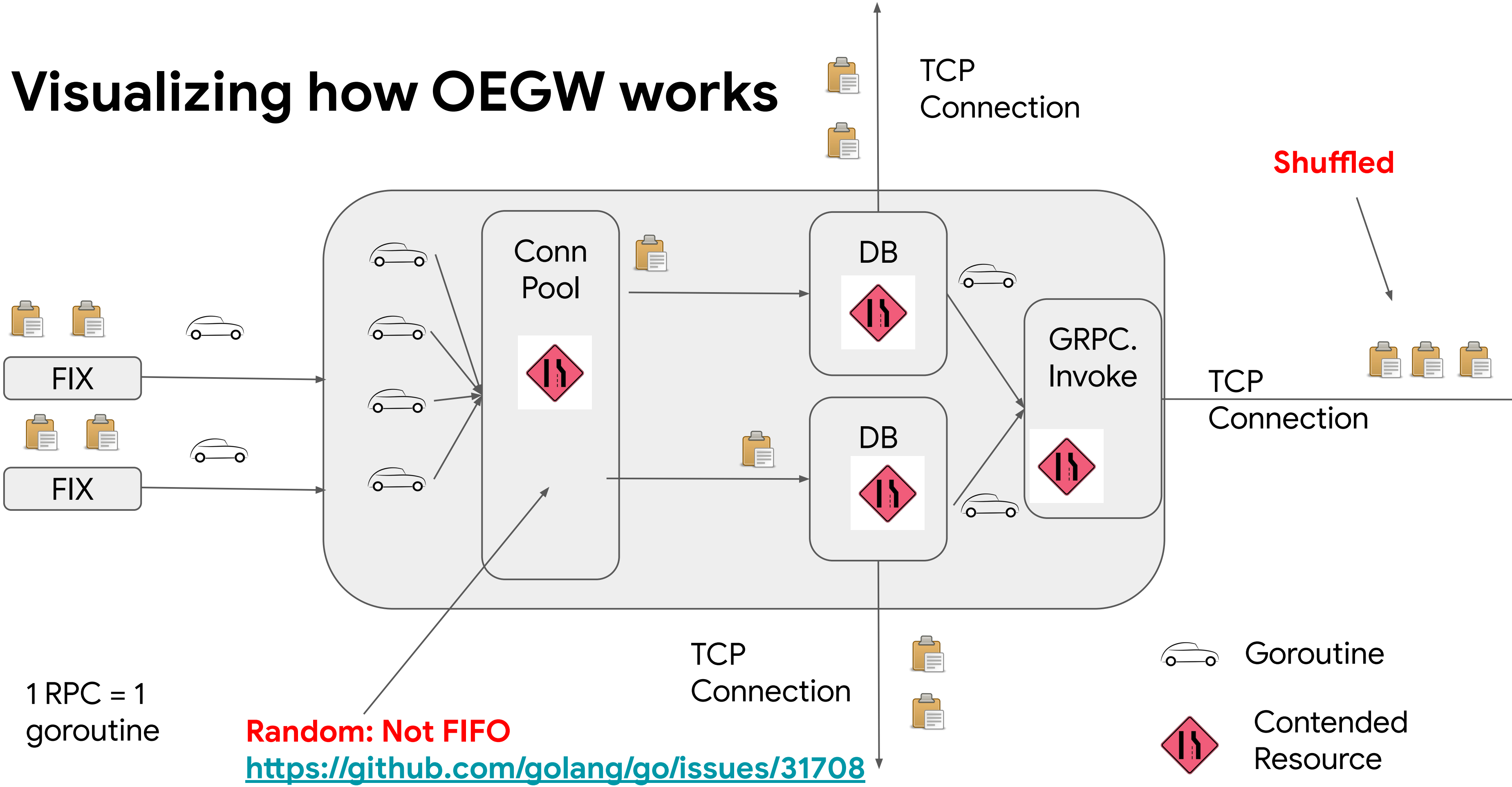
- [Go scheduler]
- GOMAXPROCS = num CPUs
- Remember: Only GOMAXPROCS will run at same time

Visualizing how API-FIX works

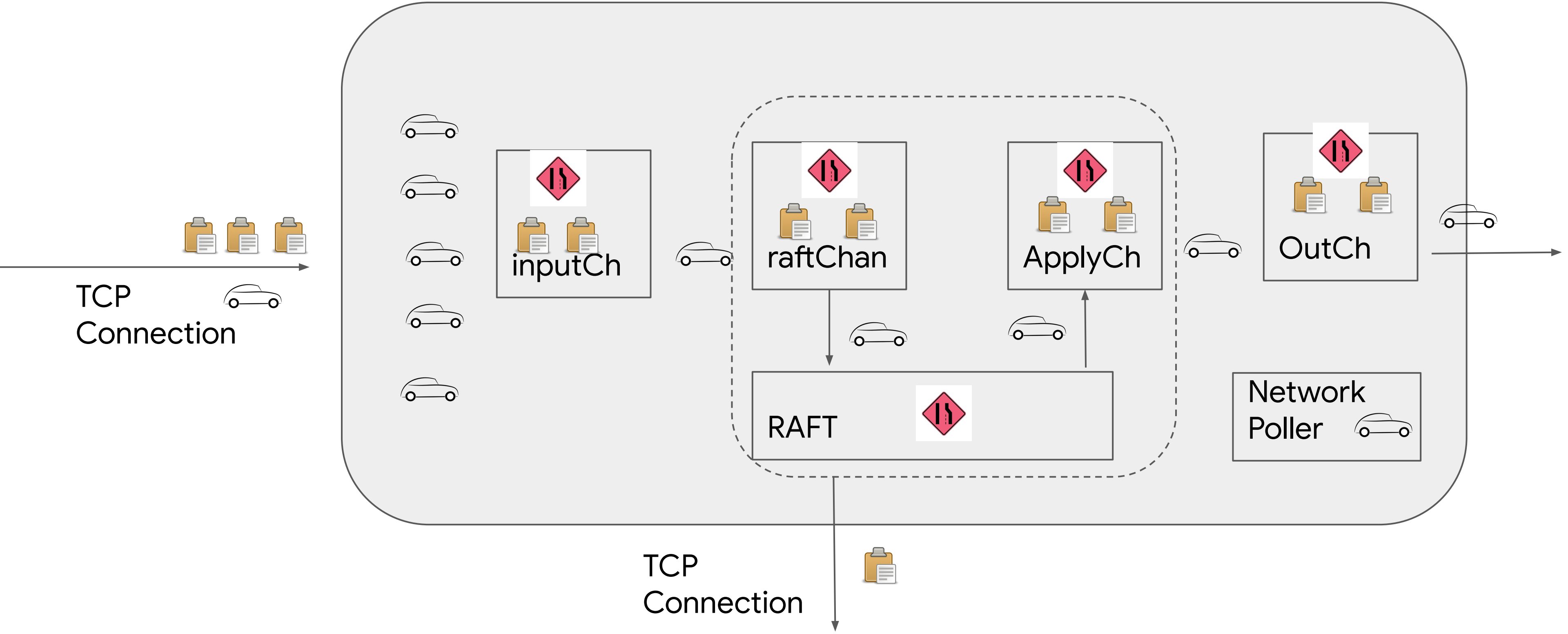


REMEMBER: Only GOMAXPROCS amount of goroutines will run at any given time

Visualizing how OEGW works



Visualizing how Trading Engine works



REMEMBER: Only GOMAXPROCS amount of goroutines will run at any given time

Mitigations: spinning important goroutine

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

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

```
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.

runtime.LockOSThread()

Mitigations: Always batch when using channels

```
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

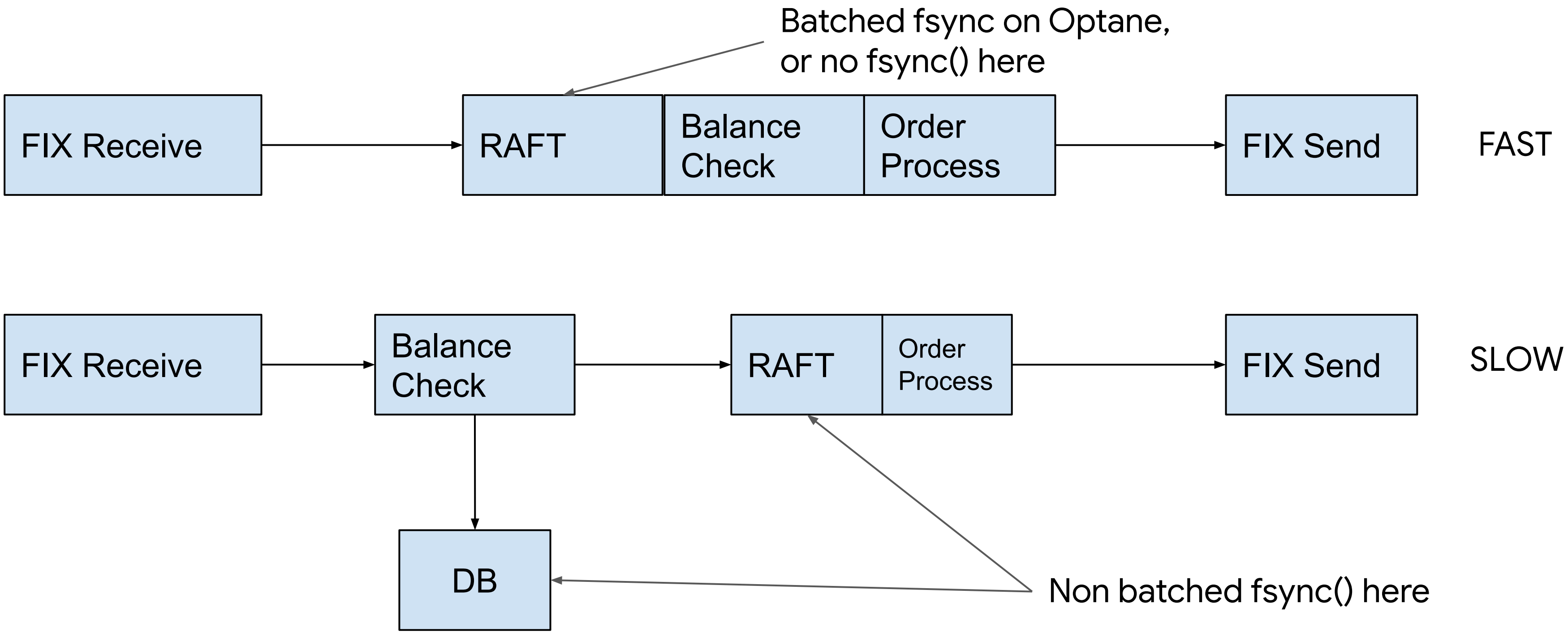
Most facilities in ~~Golang~~ Linux introduce an randomness element to optimize for throughput, not latency

- 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?



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