A Practical Guide to Market Making
Modern Methods in Low Latency Financial Services

Jacob Loveless
Chief Executive Officer

LUCERA™
FINANCIAL INFRASTRUCTURES
What is Low Latency

A systems development methodology whereby the optimization goal of speed is directly correlated to the overall value of the system.
An Example Model

To aide our conversation: let’s look at a sample market making model. For details see: “Barbarians at the Gateway”, Loveless, ACM Queue
An Example Model

Figure 6: The Order Book

- **Bid**: $8, $9
- **Ask**: $10, $11
- **Bid/Ask Spread**: $1
- **Imbalance**:
  - Bid: \( \frac{bs_0}{bs_0 + as_0} \)
  - Ask: \( \frac{as_0}{bs_0 + as_0} \)

- **UP**: 12, 10
- **DOWN**: 10, 12
- **Size Available on Each Queue**: 12, 10
- **Front of the Queue**: Before the queues
An Example Model

FIGURE 7

The Queue Position
An Example Model

**FIGURE 8**

The Trading Rate

percent of executions against the bid

probability of execution = 0.47pUP^{−1.593}
An Example Model

**FIGURE 10**

Two Methods of Getting to the Front of the Queue

a. Getting to the front: promotion

- **bid** time 0 **ask**
  - 8 | 9 | 10 | 11
  - 21 | 5 | 15 | 20
  
  we're at the second level that has been holding there for a while.

- **bid** time 1 **ask**
  - 8 | 10 | 11 | 9
  - 21 | 20 | 20 | 20

  a trade occurs and wipes out the best bid queue.

- **bid** time 2 **ask**
  - 8 | 10 | 11 | 9
  - 21 | 3 | 20 | 20

  we're now at the best bid, with good queue position.

b. Getting to the front: joining

- **bid** time 0 **ask**
  - 8 | 9 | 10 | 11
  - 21 | 25 | 15 | 20

  there is a very large bid imbalance (pUP is high).

- **bid** time 1 **ask**
  - 8 | 9 | 10 | 11
  - 21 | 25 | 15 | 20

  we join the best bid, because we're reserving a spot for later.

- **bid** time 2 **ask**
  - 8 | 9 | 11
  - 21 | 25 | 20

  Some people join behind us, but the ask queue gets eliminated, we're now set up.

c. Getting to the front: joining to help make the spread

- **bid** time 0 **ask**
  - 8 | 9 | 10 | 11
  - 21 | 25 | 15 | 20

  there is an imbalance (pUP is high enough to stay). Say pUP = 0.4, so pDN = 0.6.

- **bid** time 1 **ask**
  - 8 | 9 | 10 | 11
  - 21 | 25 | 15 | 20

  If our submit threshold is 0.45, then we can join this ask (pDN > 0.45).

- **bid** time 2 **ask**
  - 9 | 10 | 11
  - 25 | 15 | 20

  Some people join behind our ask, and we're front of the bid queue.
  So, in theory, if we get executed on the bid, we don't need the extra ask queue to get executed, only the amount in front of our ask.
Step 2: Implementation

Now Let’s Talk about the reality - this is a typical low latency design ...
Common Architecture Components

- **Edge Device:** NAT/PAT/Direct Pass/UDP Proxy
- **Network:** Edge Device: NAT/PAT/Direct Pass/UDP Proxy
- **Core Switch:** Cut through Non-Blocking
- **Microwave to Ethernet Translator:** Cut through Non-Blocking
- **TAP:** Cut through Non-Blocking
- **OS FEED HANDLER:** Tickerplant
- **Tickerplant:** FEED HANDLER
- **Data Cache:** EMS
- **EMS:** OMS
- **OMS:** Strategy Engine
- **Strategy Engine:** C&C
- **C&C:** Monitoring
- **Monitoring:** Replay Engine
- **Storage:** Market Data Store Partitions
- **PCAP File Stores:** Trade Store
- **Trade Store:** PCAP File Stores
- **Monitoring Output:** Trade Store
- **Partitions:** Market Data Store Partitions
This is Complicated

“Inside every large program, there is a small program trying to get out”

C.A.R. Hoare
Low Latency Goals: Operating System

Strong scheduling subsystem, with ability to burst
Easy to instrument in realtime
Fast IP Stack Layer
Low Latency Goals: Network

Efficient Translation from external network to internal
Dense and inexpensive external connectivity
Easy monitoring of latency
Ability to scale with bursts
100% In Order Packet Delivery internally
Low Latency Goals: Storage

High Density of capacity
Easy to backup and replicate
High performance on read and write
Strong consistency on write
High Availability on reads
Systems Engineering

Performance

REALITY

Reliability

Security
Low Latency Architecture: Operating System

“UNIX is a simple operating system, but you have to be a genius to understand the simplicity”.

-Dennis Ritchie
For this talk we will focus on Illumos (SmartOS), which is the base OS for all of Lucera operations.

http://illumOS.org

http://smartOS.org
Back to the Future: Illumos

Solaris 10 was a massively innovative operating system (ZFS, Dtrace, Crossbow, Firehouse, SMF, Zones). Circa 2001 Sun Microsystems had become a mecca for OS development, and this concentration bred innovation.

2005: Solaris is open sourced as ... OpenSolaris. January 2005 with Dtrace, June 2005 with the rest of the OS.


Few remaining bits of Solaris were still not open (tail, od, internationalization support for libC)

Sun bought by Oracle in 2009

Starting in the summer of 2010, Garrett D'Amore at Nexenta — with help from Rich Lowe, Jason King and others — began the process of either writing the closed bits from scratch or porting them from BSD

By early August, an entirely open system was booting

Dubbed “illumos” (from illuminare, Latin for illuminate and a pun on Sun) and was made available on August 3, 2010

On Friday, August 13th, 2010, an internal memo was circulated by the putative Solaris leadership:

We will distribute updates to approved CDDL or other open source-licensed code following full releases of our enterprise Solaris operating system. In this manner, new technology innovations will show up in our releases before anywhere else. We will no longer distribute source code for the entirety of the Solaris operating system in real-time while it is developed, on a nightly basis.

Solaris 11 was released on November 9, 2011 — and there was no source release

The Solaris diaspora (which was already underway) was greatly accelerated

Within 90 days, the entire DTrace team had left Oracle, all primary inventors of ZFS had left Oracle and primary engineers for both zones and networking had left Oracle

Fortunately, Oracle’s loss was illumos’s gain: nearly all of these engineers went to companies betting on illumos

Youtube “Fork Yeah: the rise of Illumos” for the single greatest talk on OS history ever, by Bryan Cantrill, father of Dtrace
A zone is similar to a virtual machine, but is distinct in that it shares the base system kernel, whereas each virtual machine runs its own OS kernel. Zones are an inherent part of the operating system and impose no additional overhead. Each process that runs includes the zone ID as an attribute. Thus, zones scale and perform better than virtual machines since there is no additional kernel or layering involved.
Zones: OS Level Virtualization

Fine grained resource control (using FSS or alternative scheduler)

Secure, contained process. Each process is its own OS from a user stack.

No Performance Impact (Bare Metal Speed)

Zones are allocated resources (preferably on a NUMA topology).

Access to resources is based on the scheduler. As a result, we can have CPU bursting (with integrated cross system knowledge).
DTrace

“In God we trust; all others must bring data.”
W. Edward Deming

Legacy Model

if (tracing_enabled)
    printf(“we got here!\n”);

Performance problems are increasingly likely to be seen in production, but they can be understood only in development. To address a performance problem seen in production, the problem must therefore be reproduced in either a development or test environment. In a software system, as in any sufficiently complicated system, disjointed pathologies can manifest the same symptoms: reproducing symptoms (e.g., high CPU load, heavy I/O traffic, long latency transactions, etc.) does not guarantee reproducing the same underlying problem.
DTrace

System performance problems are typically introduced at the highest layers of abstraction, but they are often first encountered and attributed at the lowest layers of abstraction.

It is because of this paradox that we have adopted the myth that the path to performance lies nearly exclusively with faster hardware: faster CPUs, more networking bandwidth, etc. When this fails, we have taught ourselves to move to the next layer of the stack: to demand faster operating systems, faster databases, and better compilers.

Improving these components undoubtedly improves performance, but it amounts to hunting vermin: depending on the relatively small iterative improvements at the lowest layers of the stack amounts to trying to feed a family on the likes of squirrel and skunk. If we can move up the stack—if we can find the underlying performance problems instead of merely addressing their symptoms—we can unlock much more substantial performance gains. This is bigger game to be sure; by focusing on performance problems higher in the stack, we can transition from hunting vermin to hunting cow—big, slow, stupid, tasty cow.

- Bryan Cantrill, Father of DTrace
DTrace

```
# dtrace -n syscall:::entry'{self->ts = vtimestamp}' -n syscall:::return'/self->ts/{@
quantize(vtimestamp - self->ts); self->ts = 0}' -x aggpack -x aggzoom
```

**Strong scheduling subsystem, with ability to burst**

**Easy to instrument in realtime**

**Fast IP Stack Layer**
DTrace

Strong scheduling subsystem, with ability to burst
Easy to instrument in realtime
Fast IP Stack Layer
Illumos Stack

“The key to performance is elegance, not battalions of special cases”.

--Jon Bentley and Doug McIlroy
Linux IP Stack, Simplified
Illumos Stack

Illumos IP Stack is 30% less system calls

Kernel Bypass is a clear sacrifice of reliability for performance (security as well)

Instead we use a vNIC concept, but same performance as bare metal

Add ability to failover (bonding), add security (anti-spoof) and of course DTrace!

Latency in single digit microseconds!

A Practical Guide to Designing Racetracks
Modern Architectures in Low Latency Financial Services

Strong scheduling subsystem, with ability to burst
Easy to instrument in realtime
Fast IP Stack Layer
Illumos Stack

Simpler management (can move a zone, retain network)

Dynamic VLANs (important for multicast isolation)

Less Syscalls = lower latency
Low Latency Architecture: Network

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature.”

- C.A.R Hoare
Low Latency Architecture: Network

Low Latency networks must be as simple as possible, but no simpler

External WAN links require redundancy, best handled at the application level

Internal redundancy best handled at the OS level (bonding, VLAG)

Flat Layer 2 is ideal

External connectivity can often be broken down to simple NAT/PAT or Passthrough

There is beauty in the flat design
SDN: A Reality

We operate our edge network using a custom Software Defined Network.

Our internal network uses different SDN technology but has the same effect.

- Efficient Translation from external network to internal
- Dense and inexpensive external connectivity
- Easy monitoring of latency
- Ability to scale with bursts
- 100% In Order Packet Delivery internally

Doing the network in software means we can take advantage of everything in the OS.

Dramatically reduces costs and puts networking equipment on the upgrade cycle of servers (thank you Moore’s Law).

Networking is an embarrassingly parallel problem: great fit for modern CPUs.
SDN: A Reality

Same burst capability in the OS

Financial networking is absurdly correlated at burst

Efficient Translation from external network to internal
Dense and inexpensive external connectivity
Easy monitoring of latency
Ability to scale with bursts
100% In Order Packet Delivery internally

Integrated monitoring means we can build dynamic and reactive networks!
Switches Matter

IB is wonderful. But difficult to scale (and code to).

Congestion is the side effect of Ethernet.

New higher end switches can do congestion free design (or protocol based QoS).

UDP based designs require this. Anti-congestion in the switch can get you IB style delivery with Ethernet.
Low Latency Architecture: Storage

“It’s impossible to comprehend the details of God’s plan for the Universe. But I can tell you it’s backed up on ZFS.”

- Jacob Loveless
Low Latency Architecture: ZFS

**L1ARC**
- uses “all available” memory for caching
- degrades gracefully multiple methods
- generally >80% hit rate

**L2ARC**
- SSD cache
- data moved from L1 to L2
- generally >95% hit rate

On slow disks 90% of reads are as fast as SSDs or even raw memory access

**ZFS Intent Log (ZIL)**
- Durable transaction log for ZFS operations
- Write calls can return before being committed to slow spinning disks

On slow disks writes are as fast as SSDs during normal load or short bursts
A Practical Guide to Designing Racetracks
Modern Architectures in Low Latency Financial Services

Classical Stack

Application
Data Processing
Data Integrity

Operating System
Caching

Files
Files

Volume Manager
Volumes

RAID
Blocks

Disk
Physical Location

ZFS-based Stack

Application
Data Processing

ZFS
Data Integrity
Caching
Files
Volumes
Blocks
Disks

Disk
Physical Location
Low Latency Architecture: ZFS

Filesystem Transactional Storage (analogous Software Transactional Memory)
  atomic operations
  copy on write

In reality this means
  No interruption of operations
  Instant snapshot creation/rollback, no overhead on time or storage
    size = deltas since snapshot creation
  Instant rollback from snapshots
Low Latency Architecture: Storage

There are essentially three protocols for scalable storage: block, file and object.

Block (i.e., a SAN) is far too low an abstraction — and notoriously expensive to scale.

File (i.e., NAS) is too permissive an abstraction — it implies a coherent store for arbitrary (partial) writes, trying (and failing) to be both C and A in CAP.

Object (e.g., S3) is similar “enough” to a file-based abstraction, but by not allowing partial writes, allows for proper CAP tradeoffs.
High-Availability Architecture: Object Store

Main abstraction: immutable objects; write once, read often

No strong relation between objects: objects are values accessible by a key

Replication and distribution increases performance and availability of the storage system beyond a single system: location awareness, rack-level replication, datacenter replication

Object sharding for concurrent, high-throughput reads and writes & virtually no size limit of objects

Runs on ZFS!

If you like then you better put a ring on it