The LMAX Architecture

LMAX is a new retail financial trading platform. As a result it has to process many trades with low latency. The system is built on the JVM platform and centers on a Business Logic Processor that can handle 6 million orders per second on a single thread. The Business Logic Processor runs entirely in-memory using event sourcing. The Business Logic Processor is surrounded by Disruptors - a concurrency component that implements a network of queues that operate without needing locks. During the design process the team concluded that recent directions in high-performance concurrency models using queues are fundamentally at odds with modern CPU design.

12 July 2011

Martin Fowler

APPLICATION ARCHITECTURE

EXPERIENCE REPORTS

EXPOSITIONAL ARCHITECTURES

EVENT ARCHITECTURES

CONTENTS

Table of Contents

Over the last few years we keep hearing that "the free lunch is over"[1] - we can't expect increases in individual CPU speed. So to write fast code we need to explicitly use multiple processors with concurrent software. This is not good news - writing concurrent code is very hard. Locks and semaphores are hard to reason about and hard to test - meaning we are spending more time worrying about satisfying the computer than we are solving the domain problem. Various concurrency models, such as Actors and Software Transactional Memory, aim to make this easier - but there is still a burden that introduces bugs and complexity.

So I was fascinated to hear about a talk at QCon London in March last year from LMAX. LMAX is a new retail financial trading platform. Its business innovation is that it is a *retail* platform - allowing anyone to trade in a range of financial derivative products[2]. A trading platform like this needs very low latency - trades have to be processed quickly because the market is moving rapidly. A retail platform adds complexity because it has to do this for lots of people. So the result is more users, with lots of trades, all of which need to be processed quickly.[3]

Given the shift to multi-core thinking, this kind of demanding performance would naturally suggest an explicitly concurrent programming model - and indeed this was their starting point. But the thing that got people's attention at QCon was that this wasn't where they ended up. In fact they ended up by doing all the business logic for their platform: all trades, from all customers, in all markets - on a single thread. A thread that will process 6 million orders per second using commodity hardware.[4]

Table of Contents

Processing lots of transactions with low-latency and none of the complexities of concurrent code - how can I resist digging into that? Fortunately another difference LMAX has to other financial companies is that they are quite happy to talk about their technological decisions. So now LMAX has been in production for a while it's time to explore their fascinating design.

Overall Structure

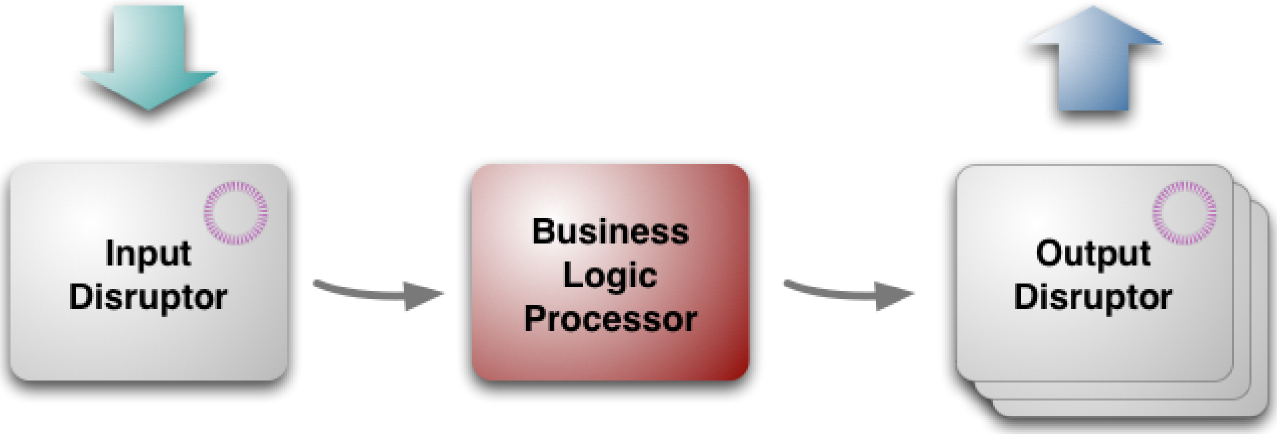


Figure 1: LMAX's architecture in three blobs

At a top level, the architecture has three parts

- business logic processor[5]
- input disruptor
- output disruptors

As its name implies, the business logic processor handles all the business logic in the application. As I indicated above, it does this as a single-events. Consequently it's a simple java program that doesn't require any platform frameworks to run other than the JVM itself, which allows it to be easily run in test environments.

Although the Business Logic Processor can run in a simple environment for testing, there is rather more involved choreography to get it to run in a production setting. Input messages need to be taken off a network gateway and unmarshaled, replicated and journaled. Output messages need to be marshaled for the network. These tasks are handled by the input and output disruptors. Unlike the Business Logic Processor, these are concurrent components, since they involve IO operations which are both slow and independent. They were designed and built especially for LMAX, but they (like the overall architecture) are applicable elsewhere.

Business Logic Processor

Keeping it all in memory

The Business Logic Processor takes input messages sequentially (in the form of a method invocation), runs business logic on it, and emits output events. It operates entirely in-memory, there is no database or other persistent store. Keeping all data in-memory has two important benefits. Firstly it's fast - there's no database to provide slow IO to access, nor is there any transactional behavior to execute since all the processing is done sequentially. The second advantage is that it simplifies programming - there's no object/relational mapping to do. All the code

Table of Contents

can be written using Java's object model without having to make any compromises for the mapping to a database.

Using an in-memory structure has an important consequence - what happens if everything crashes? Even the most resilient systems are vulnerable to someone pulling the power. The heart of dealing with this is Event Sourcing - which means that the current state of the Business Logic Processor is entirely derivable by processing the input events. As long as the input event stream is kept in a durable store (which is one of the jobs of the input disruptor) you can always recreate the current state of the business logic engine by replaying the events.

A good way to understand this is to think of a version control system. Version control systems are a sequence of commits, at any time you can build a working copy by applying those commits. VCSs are more complicated than the Business Logic Processor because they must support branching, while the Business Logic Processor is a simple sequence.

So, in theory, you can always rebuild the state of the Business Logic Processor by reprocessing all the events. In practice, however, that would take too long should you need to spin one up. So, just as with version control systems, LMAX can make snapshots of the Business Logic Processor state and restore from the snapshots. They take a snapshot every night during periods of low activity. Restarting the Business Logic Processor is fast, a full restart - including restarting the JVM, loading a recent snapshot, and replaying a days worth of journals - takes less than a minute.

Snapshots make starting up a new Business Logic Processor faster, but not quickly enough should a Business Logic Processor crash at 2pm. As a result LMAX keeps multiple Business Logic Processors running all the time[6]. Each input event is processed by multiple processors, but all but one processor has its output ignored. Should the live processor fail, the system switches to another one. This ability to handle fail-over is another benefit of using Event Sourcing.

By event sourcing into replicas they can switch between processors in a matter of micro-seconds. As well as taking snapshots every night, they also restart the Business Logic Processors every night. The replication allows them to do this with no downtime, so they continue to process trades 24/7.

> For more background on Event Sourcing, see the draft pattern on my site from a few years ago. The article is more focused on handling temporal relationships rather than the benefits that LMAX use, but it does explain the core idea.

Event Sourcing is valuable because it allows the processor to run entirely in-memory, but it has another considerable advantage for diagnostics. If some unexpected behavior occurs, the team copies the sequence of events to their development environment and replays them there. This allows them to examine what happened much more easily than is possible in most environments.

This diagnostic capability extends to business diagnostics. There are some business tasks, such as in risk management, that require significant computation that isn't needed for processing orders. An example is getting a list of the top 20 customers by risk profile based on their current trading positions. The team handles this by spinning up a replicate domain model and carrying out the computation there, where it won't interfere with the core order processing. These analysis domain models can have variant data models, keep different data sets in memory, and run on different machines.

Table of Contents

Tuning performance

So far I've explained that the key to the speed of the Business Logic Processor is doing everything sequentially, in-memory. Just doing this (and nothing really stupid) allows developers to write code that can process 10K TPS[7]. They then found that concentrating on the simple elements of good code could bring this up into the 100K TPS range. This just needs well-factored code and small methods - essentially this allows Hotspot to do a better job of optimizing and for CPUs to be more efficient in caching the code as it's running.

It took a bit more cleverness to go up another order of magnitude. There are several things that the LMAX team found helpful to get there. One was to write custom implementations of the java collections that were designed to be cache-friendly and careful with garbage[8]. An example of this is using primitive java longs as hashmap keys with a specially written array backed Map implementation (`LongToObjectHashMap`). In general they've found that choice of data structures often makes a big difference, Most programmers just grab whatever List they used last time rather than thinking which implementation is the right one for this context.[9]

Another technique to reach that top level of performance is putting attention into performance testing. I've long noticed that people talk a lot about techniques to improve performance, but the one thing that really makes a difference is to test it. Even good programmers are very good at constructing performance arguments that end up being wrong, so the best programmers prefer profilers and test cases to speculation.[10] The LMAX team has also found that writing tests first is a very effective discipline for performance tests.

Programming Model

Table of Contents

This style of processing does introduce some constraints into the way you write and organize the business logic. The first of these is that you have to tease out any interaction with external services. An external service call is going to be slow, and with a single thread will halt the entire order processing machine. As a result you can't make calls to external services within the business logic. Instead you need to finish that interaction with an output event, and wait for another input event to pick it back up again.

I'll use a simple non-LMAX example to illustrate. Imagine you are making an order for jelly beans by credit card. A simple retailing system would take your order information, use a credit card validation service to check your credit card number, and then confirm your order - all within a single operation. The thread processing your order would block while waiting for the credit card to be checked, but that block wouldn't be very long for the user, and the server can always run another thread on the processor while it's waiting.

In the LMAX architecture, you would split this operation into two. The first operation would capture the order information and finish by outputting an event (credit card validation requested) to the credit card company. The Business Logic Processor would then carry on processing events for other customers until it received a credit-card-validated event in its input event stream. On processing that event it would carry out the confirmation tasks for that order.

Working in this kind of event-driven, asynchronous style, is somewhat unusual - although using asynchrony to improve the responsiveness of an application is a familiar technique. It also helps the business process be more resilient, as you have to be more explicit in thinking about the different things that can happen with the remote application.

A second feature of the programming model lies in error handling. The traditional model of sessions and database transactions provides a helpful error handling capability. Should anything go wrong, it's easy to throw away everything that happened so far in the interaction. Session data is transient, and can be discarded, at the cost of some irritation to the user if in the middle of something complicated. If an error occurs on the database side you can rollback the transaction.

LMAX's in-memory structures are persistent across input events, so if there is an error it's important to not leave that memory in an inconsistent state. However there's no automated rollback facility. As a consequence the LMAX team puts a lot of attention into ensuring the input events are fully valid before doing any mutation of the in-memory persistent state. They have found that testing is a key tool in flushing out these kinds of problems before going into production.

Input and Output Disruptors

Although the business logic occurs in a single thread, there are a number tasks to be done before we can invoke a business object method. The original input for processing comes off the wire in the form of a message, this message needs to be unmarshaled into a form convenient for Business Logic Processor to use. Event Sourcing relies on keeping a durable journal of all the input events, so each input message needs to be journaled onto a durable store. Finally the architecture relies on a cluster of Business Logic Processors, so we have to replicate the input messages across this cluster. Similarly on the output side, the output events need to be marshaled for transmission over the network.

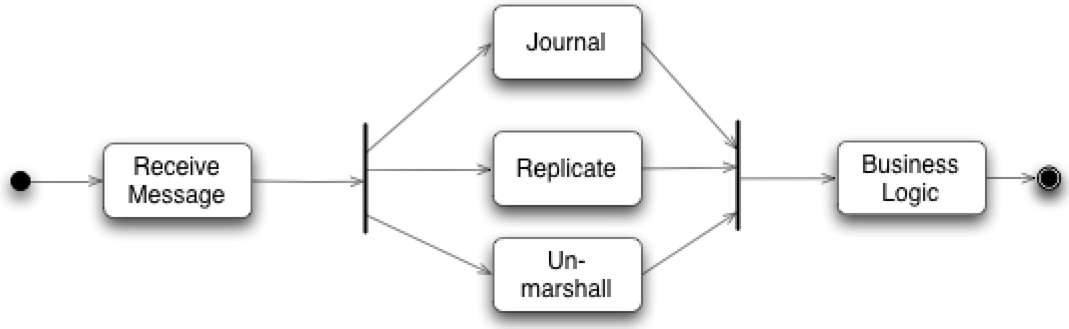


Figure 2: The activities done by the input disruptor (using UML activity diagram notation)

The replicator and journaler involve IO and therefore are relatively slow. After all the central idea of Business Logic Processor is that it avoids doing any IO. Also these three tasks are relatively independent, all of them need to be done before the Business Logic Processor works on a message, but they can done in any order. So unlike with the Business Logic Processor, where each trade changes the market for subsequent trades, there is a natural fit for concurrency.

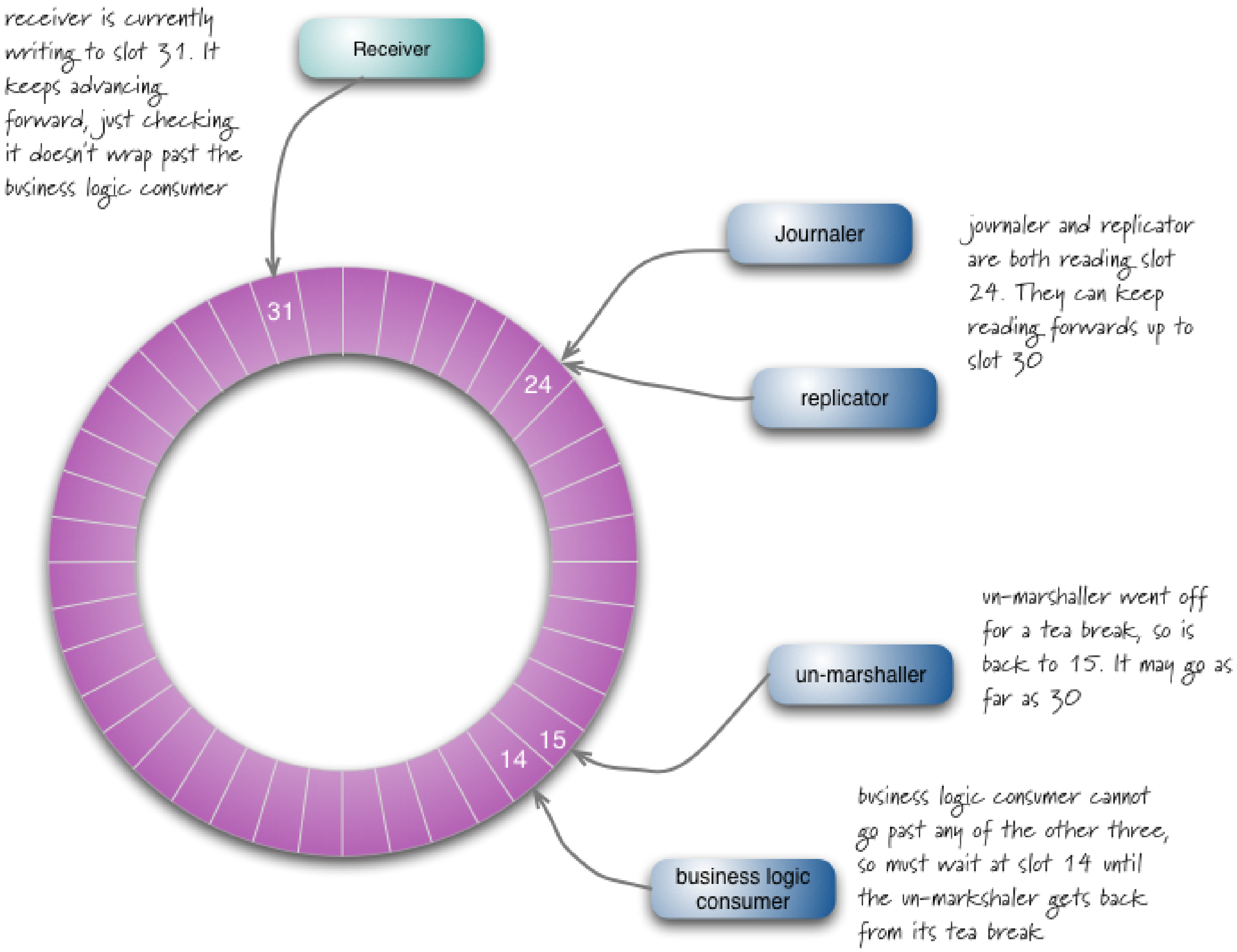
To handle this concurrency the LMAX team developed a special concurrency component, which they call a **Disruptor**[11].

The LMAX team have released the source code for the Disruptor with an open source licence.

At a crude level you can think of a Disruptor as a multicast graph of queues where producers put objects on it that are sent to all the consumers for parallel consumption through separate downstream queues. When you look inside you see that this network of queues is really a single data structure - a ring buffer. Each producer and consumer has a sequence counter to indicate which slot in the buffer it's currently working on. Each producer/consumer writes its own sequence counter

Table of Contents

but can read the others' sequence counters. This way the producer can read the consumers' counters to ensure the slot it wants to write in is available without any locks on the counters. Similarly a consumer can ensure it only processes messages once another consumer is done with it by watching the counters.



receiver is currently writing to slot 31. It keeps advancing forward, just checking it doesn't wrap past the business logic consumer

journaler and replicator are both reading slot 24. They can keep reading forwards up to slot 30

un-marshaller went off for a tea break, so is back to 15. It may go as far as 30

business logic consumer cannot go past any of the other three, so must wait at slot 14 until the un-markshaler gets back from its tea break

Figure 3: The input disruptor coordinates one producer and four consumers

Output disruptors are similar but they only have two sequential consumers for marshaling and output.[12] Output events are organized into several topics, so that messages can be sent to only the receivers who are interested in them. Each topic has its own disruptor.

The disruptors I've described are used in a style with one producer and multiple consumers, but this isn't a limitation of the design of the disruptor. The disruptor can work with multiple producers too, in this case it still doesn't need locks.[13]

A benefit of the disruptor design is that it makes it easier for consumers to catch up quickly if they run into a problem and fall behind. If the unmarshaler has a problem when processing on slot 15 and returns when the receiver is on slot 31, it can read data from slots 16-30 in one batch to catch up. This batch read of the data from the disruptor makes it easier for lagging consumers to catch up quickly, thus reducing overall latency.

I've described things here, with one each of the journaler, replicator, and unmarshaler - this indeed is what LMAX does. But the design would allow multiple of these components to run. If you ran two journalers then one would take the even slots and the other journaler would take the odd slots. This allows further concurrency of these IO operations should this become necessary.

The ring buffers are large: 20 million slots for input buffer and 4 million slots for each of the output buffers. The sequence counters are 64bit long integers that increase monotonically even as the ring slots wrap.[14] The buffer is set to a size that's a power of two so the compiler can do an efficient modulus operation to map from the sequence counter number to the slot number. Like the rest of the system, the disruptors are bounced overnight. This bounce is mainly done to wipe memory so that there is less chance of an expensive garbage collection event during trading. (I also think it's a good habit to regularly restart, so that you rehearse how to do it for emergencies.)

The journaler's job is to store all the events in a durable form, so that they can be replayed should anything go wrong. LMAX does not use a database for this, just the file system. They stream the events onto the disk. In modern terms, mechanical disks are horribly slow for random access, but very fast for streaming - hence the tag-line "disk is the new tape".[15]

Earlier on I mentioned that LMAX runs multiple copies of its system in a cluster to support rapid failover. The replicator keeps these nodes in sync. All communication in LMAX uses IP multicasting, so clients don't need to know which IP address is the leader node. Only the leader node listens directly to input events and runs a replicator. The replicator broadcasts the input events to the follower nodes. Should the leader node go down, it's lack of heartbeat will be noticed, another node becomes leader, starts processing input events, and starts its replicator. Each node has its own input disruptor and thus has its own journal and does its own unmarshaling.

Even with IP multicasting, replication is still needed because IP messages can arrive in a different order on different nodes. The leader node provides a deterministic sequence for the rest of the processing.

The unmarshaler turns the event data from the wire into a java object that can be used to invoke behavior on the Business Logic Processor. Therefore, unlike the other consumers, it needs to modify the data in the ring buffer so it can store this unmarshaled object. The rule here is that consumers are permitted to write to the ring buffer, but each writable field can only have one parallel consumer that's allowed to write to it. This preserves the principle of only having a single writer. [16]

Table of Contents

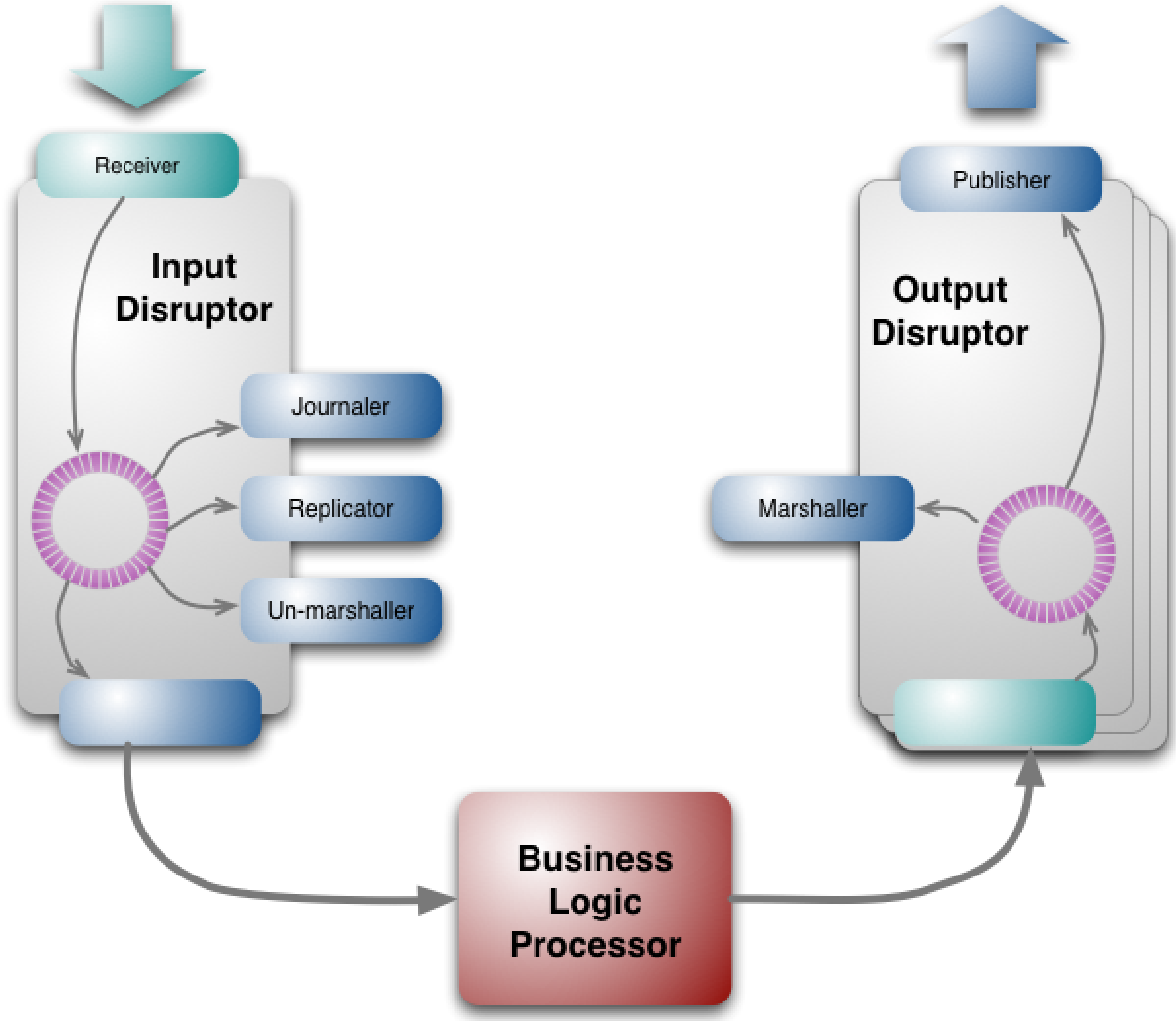


Figure 4: The LMAX architecture with the disruptors expanded

The disruptor is a general purpose component that can be used outside of the LMAX system. Usually financial companies are very secretive about their systems, keeping quiet even about items that aren't germane to their business. Not just has LMAX been open about its overall architecture, they have open-sourced the disruptor code - an act that makes me very happy. Not just will this allow other organizations to make use of the disruptor, it will also allow for more testing of its concurrency properties.

Table of Contents

Queues and their lack of mechanical sympathy

The LMAX architecture caught people's attention because it's a very different way of approaching a high performance system to what most people are thinking about. So far I've talked about how it works, but haven't delved too much into why it was developed this way. This tale is interesting in itself, because this architecture didn't just appear. It took a long time of trying more conventional alternatives, and realizing where they were flawed, before the team settled on this one.

Most business systems these days have a core architecture that relies on multiple active sessions coordinated through a transactional database. The LMAX team were familiar with this approach, and confident that it wouldn't work for LMAX. This assessment was founded in the experiences of Betfair - the parent company who set up LMAX. Betfair is a betting site that allows people to bet on sporting events. It handles very high volumes of traffic with a lot of contention - sports bets tend to burst around particular events. To make this work they have one of the hottest database installations around and have had to do many unnatural acts in order to make it work. Based on this experience they knew how difficult it was to maintain Betfair's performance and were sure that this kind of architecture would not work for the very low latency that a trading site would require. As a result they had to find a different approach.

Their initial approach was to follow what so many are saying these days - that to get high performance you need to use explicit concurrency. For this scenario, this means allowing orders to be processed by multiple threads in parallel. However, as is often the case with concurrency, the difficulty comes because these threads have to communicate with each other. Processing an order changes market conditions and these conditions need to be communicated.

The approach they explored early on was the Actor model and its cousin SEDA. The Actor model relies on independent, active objects with their own thread that communicate with each other via queues. Many people find this kind of concurrency model much easier to deal with than trying to do something based on locking primitives.

The team built a prototype exchange using the actor model and did performance tests on it. What they found was that the processors spent more time managing queues than doing the real logic of the application. Queue access was a bottleneck.

When pushing performance like this, it starts to become important to take account of the way modern hardware is constructed. The phrase Martin Thompson likes to use is "mechanical sympathy". The term comes from race car driving and it reflects the driver having an innate feel for the car, so they are able to feel how to get the best out of it. Many programmers, and I confess I fall into this camp, don't have much mechanical sympathy for how programming interacts with hardware. What's worse is that many programmers think they have mechanical sympathy, but it's built on notions of how hardware used to work that are now many years out of date.

One of the dominant factors with modern CPUs that affects latency, is how the CPU interacts with memory. These days going to main memory is a very slow operation in CPU-terms. CPUs have multiple levels of cache, each of which of is significantly faster. So to increase speed you want to get your code and data in those caches.

At one level, the actor model helps here. You can think of an actor as its own object that clusters code and data, which is a natural unit for

- and the LMAX team observed that it's the queues that interfere with caching.

The explanation runs like this: in order to put some data on a queue, you need to write to that queue. Similarly, to take data off the queue, you need to write to the queue to perform the removal. This is write contention - more than one client may need to write to the same data structure. To deal with the write contention a queue often uses locks. But if a lock is used, that can cause a context switch to the kernel. When this happens the processor involved is likely to lose the data in its caches.

The conclusion they came to was that to get the best caching behavior, you need a design that has only one core writing to any memory location[17]. Multiple readers are fine, processors often use special high-speed links between their caches. But queues fail the one-writer principle.

This analysis led the LMAX team to a couple of conclusions. Firstly it led to the design of the disruptor, which determinedly follows the single-writer constraint. Secondly it led to idea of exploring the single-threaded business logic approach, asking the question of how fast a single thread can go if it's freed of concurrency management.

The essence of working on a single thread, is to ensure that you have one thread running on one core, the caches warm up, and as much memory access as possible goes to the caches rather than to main memory. This means that both the code and the working set of data needs to be as consistently accessed as possible. Also keeping small objects with code and data together allows them to be swapped between the caches as a unit, simplifying the cache management and again improving performance.

An essential part of the path to the LMAX architecture was the use of

Table of Contents

based approach came from building and performance testing a prototype. Similarly much of the steps in improving the performance of the various components were enabled by performance tests. Mechanical sympathy is very valuable - it helps to form hypotheses about what improvements you can make, and guides you to forward steps rather than backward ones - but in the end it's the testing gives you the convincing evidence.

Performance testing in this style, however, is not a well-understood topic. Regularly the LMAX team stresses that coming up with meaningful performance tests is often harder than developing the production code. Again mechanical sympathy is important to developing the right tests. Testing a low level concurrency component is meaningless unless you take into account the caching behavior of the CPU.

One particular lesson is the importance of writing tests against null components to ensure the performance test is fast enough to really measure what real components are doing. Writing fast test code is no easier than writing fast production code and it's too easy to get false results because the test isn't as fast as the component it's trying to measure.

Should you use this architecture?

At first glance, this architecture appears to be for a very small niche. After all the driver that led to it was to be able to run lots of complex transactions with very low latency - most applications don't need to run at 6 million TPS.

But the thing that fascinates me about this application, is that they have ended up with a design which removes much of the programming complexity that plagues many software projects. The traditional model of concurrent sessions surrounding a transactional database isn't free of hassles. There's usually a non-trivial effort that goes into the relationship with the database. Object/relational mapping tools can help much of the pain of dealing with a database, but it doesn't deal with it all. Most performance tuning of enterprise applications involves futzing around with SQL.

These days, you can get more main memory into your servers than us old guys could get as disk space. More and more applications are quite capable of putting all their working set in main memory - thus eliminating a source of both complexity and sluggishness. Event Sourcing provides a way to solve the durability problem for an in-memory system, running everything in a single thread solves the concurrency issue. The LMAX experience suggests that as long as you need less than a few million TPS, you'll have enough performance headroom.

There is a considerable overlap here with the growing interest in CQRS. An event sourced, in-memory processor is a natural choice for the command-side of a CQRS system. (Although the LMAX team does not currently use CQRS.)

So what indicates you shouldn't go down this path? This is always a tricky questions for little-known techniques like this, since the profession needs more time to explore its boundaries. A starting point, however, is to think of the characteristics that encourage the architecture.

One characteristic is that this is a connected domain where processing are processed. With transactions that are more independent of each other, there's less need to coordinate, so using separate processors running in parallel becomes more attractive.

LMAX concentrates on figuring the consequences of how events change the world. Many sites are more about taking an existing store of information and rendering various combinations of that information to as many eyeballs as they can find - eg think of any media site. Here the architectural challenge often centers on getting your caches right.

Another characteristic of LMAX is that this is a backend system, so it's reasonable to consider how applicable it would be for something acting in an interactive mode. Increasingly web application are helping us get used to server systems that react to requests, an aspect that does fit in well with this architecture. Where this architecture goes further than most such systems is its absolute use of asynchronous communications, resulting in the changes to the programming model that I outlined earlier.

These changes will take some getting used to for most teams. Most people tend to think of programming in synchronous terms and are not used to dealing with asynchrony. Yet it's long been true that asynchronous communication is an essential tool for responsiveness. It will be interesting to see if the wider use of asynchronous communication in the javascript world, with AJAX and node.js, will encourage more people to investigate this style. The LMAX team found that while it took a bit of time to adjust to asynchronous style, it soon became natural and often easier. In particular error handling was much easier to deal with under this approach.

The LMAX team certainly feels that the days of the coordinating transactional database are numbered. The fact that you can write software more easily using this kind of architecture and that it runs more quickly removes much of the justification for the traditional central database.

For my part, I find this a very exciting story. Much of my goal is to concentrate on software that models complex domains. An architecture like this provides good separation of concerns, allowing people to focus on Domain-Driven Design and keeping much of the platform complexity well separated. The close coupling between domain objects and databases has always been an irritation - approaches like this suggest a way out.

Footnotes

1: The Free Lunch is Over

This is the title of a famous essay by Herb Sutter. He describes the "free lunch" as the ever increasing clock speed of processors that regularly gave us more CPU performance every year. His point was that such clock cycle increases were no longer going to happen, instead performance increases would come in terms of multiple cores. But to take advantage of multiple cores, you need software that is capable of working concurrently - so without a shift in programming style people would no longer get the performance lunch for free.

2: I shall remain silent on what I think about the value of this innovation

Table of Contents

All trading systems need low latency, since one trade can affect later trades and there's a lot of competition based on rapid reaction. Most trading platforms are for professionals - banks, brokers, etc - and typically have hundreds of users. A retail system has the potential for many more users, Betfair has millions of users and LMAX is designed for that scale. (The LMAX team isn't allowed to disclose its actual volumes.)

As it turns out, although a retail system has a lot of users, most of the activity in comes from market makers. During volatile periods an instrument can get hundreds of updates per second, with unusual micro-bursts of hundreds of transactions within a single microsecond.

4: Hardware

The 6 million TPS benchmark was measured on a 3Ghz dual-socket quad-core Nehalem based Dell server with 32GB RAM.

5: The team does not use the name Business Logic Processor, in fact they have no name for that component, just referring to it as the business logic or core services. I've given it a name to make it easier to talk about in this article.

6: Currently LMAX runs two Business Logic Processors in its main data center and a third at a disaster recovery site. All three process input events.

7: What's in a transaction

When people talk about transaction timing, one of the problems is what exactly is in a transaction. In some cases it's little more than inserting a new record in a database. LMAX's transactions are reasonably complex, more complex than a typical retail sale.

Table of Contents

- checking the target market is open to take orders
- checking the order is valid for that market
- choosing the right matching policy for the type of order
- sequencing the order so that each order is matched at the best possible price and matched with the right liquidity
- creating and publicizing the trades made as a consequence of the match
- updating prices based on the new trades

8: At this scale of latency, you have to be aware of the garbage collector. For almost all systems these days, a modern GC compaction isn't going to have any noticeable effect on performance. However when you are trying to process millions of transactions per second with minimum jitter, a GC pause becomes a problem. The thing to remember is that short lived objects are ok, as they get collected quickly. So are objects that are permanent, since they will live for ever. The problematic objects are those that will get promoted to an older generation, but will eventually die. As this fragments the older generation region, it will trigger the compaction.

9: I rarely think about which collection implementation to use. This is perfectly reasonable when you're not in performance critical code. Different contexts suggest different behavior.

10: An interesting side-note. While the LMAX team shares much of the current interest in functional programming, they believe that the OO approach provides a better approach for this kind of problem. They've noticed that as they work to write faster code, they move away from a functional style towards OO style. Partly this because of the copying of data that functional styles require to maintain immutability. But it's also because objects provide a better model of a complex domain with a richer choice of data structures.

Table of Contents

11: The name "disruptor" was inspired from a couple of sources. One is the the fact that the LMAX team sees this component as something that disrupts current thinking on concurrency. The other is a response to the fact that Java is introducing a phaser, so it's natural to include disruptors too.

12: It would be possible to journal the output events too. This would have the advantage of not needing to recalculate them should they need to be replayed for downstream services. In practice, however, this isn't worthwhile. The business logic is deterministic and very fast, so there's no gain from storing the results.

13: Although it does need to use CAS instructions in this case. See the disruptor technical paper for more information.

14: This does mean that if they process a billion transactions per second the counter will wrap in 292 years, causing some hell to break loose. They have decided that fixing this is not a high priority.

15: SSDs are better at random access, but a disk-like IO system slows them down.

16: Another complication when writing fields is you have to ensure that any fields being written to are separated into different cache lines.

17: Ensuring a single writer to a memory location

A complication in following the single-writer principle is that processors don't grab memory one location at a time. Rather they sweep up multiple contiguous locations, called a **cache line**, into cache in one go. Accessing memory in cache line chunks is obviously more efficient, but also means that you have to ensure you don't have locations within that cache line that are written by different cores. So, for example, the Disruptor's sequence counter are padded to ensure they appear in separate cache lines.

Acknowledgments

Financial institutions are usually secretive with their technical work, usually with little reason. This is a problem as it hampers the ability for the profession to learn from experience. So I'm especially thankful for LMAX's openness in discussing their experiences - both with this article and in their other material.

The main creators of the Disruptor are Martin Thompson, Mike Barker, and Dave Farley

Martin Thompson and Dave Farley gave me a detailed walk-through of the LMAX architecture that served as the basis for this article. They also responded swiftly to email questions to improve my early drafts.

Concurrent programming is a tricky field that requires lots of attention to be competent at - and I have not put that effort in. As a result I'm entirely dependent upon others for understanding on concurrency and am thankful for their patient advice.

Further Reading

If you'd prefer a video description of the LMAX architecture from LMAX team members, your best bet is the QCon presentation given in San Francisco 2010 by Martin Thompson and Michael Barker.

The source code for the Disruptor is available as open source. There is also a good technical paper (pdf) that goes into more depth as well as a collection of blogs and articles on it.

Various members of the LMAX team have their own blogs: Martin Thompson, Michael Barker, and Trisha Gee.

▶ Significant Revisions

Table of Contents