The LMAX Disruptor – Open-Sourced Mechanical Sympathy in Action

An Interview with Martin Thompson

By Mike O'Hara, 1st August 2011

In this interview for the High Frequency Trading Review, Mike O'Hara talks to **Martin Thompson**, Chief Technical Officer of LMAX (http://www.lmaxtrader.co.uk), about the design principles behind the Disruptor, the Concurrent Programming Framework for high-performance, low-latency transaction processing, which is used to run LMAX's own trading platform and was open-sourced earlier this year.

HFT Review: *Martin, before we look in detail at what the Disruptor actually does and how it does it, can you give us some background on LMAX and how the Disruptor came about?*

Martin Thompson: Yes, LMAX is an online exchange for retail investors who want to trade FX and CFDs (contracts for difference) on a range of different products.

In terms of background, our parent company Betfair followed the typical Internet trajectory of a successful company and become bound by technology for its own growth. So the question was, how could we come up with a new exchange that could basically put to bed the growth issues for a while and cope with any capacity issues that we might face over the next five years?

Our goal was to come up with an exchange that was one hundred times the throughput capacity of the current Betfair exchange, so we called that project 100X. It was a successful project; the final result was a prototype that was capable of 120X. That heavily influenced the design of the exchange here at LMAX when we started.

HFTR*: What kind of message volumes and throughput are we talking about here?*

MT: We internally set ourselves a goal of designing an exchange that could do over 100,000 orders per second, with less than one millisecond latency. And although we're not seeing those transaction volumes yet, we're building the systems capable of scaling to more than that.

***HFTR**: What were the main challenges you faced when building this infrastructure?*

MT: One of the things we learned very quickly as we started profiling this application was that we had significant latency. In the retail space, latency is very critical to people managing their risk when getting out of positions by getting a closing order onto an order book quickly, so we looked into what was the cause of that latency and we discovered that the queues in the different stages of processing an order, or managing someone's risk, were actually taking up more time than the business logic itself

We were using an architecture back then called SEDA (Staged Event Driven Architecture), where you break down the process into a number of steps, a CPU thread is allocated to each step, it does part of the work, puts the results of that work onto the queue for the next step to pick it up and then moves on to the next step in the queue, basically working like a pipeline through the process. But we discovered that putting things onto the queue and taking things off the queue was taking the most time, introducing a lot of latency into the process.

We also realised that lots of things couldn't run in parallel. For example, inside our core exchange, whenever we get a message off the network, it's just a stream of bytes, which we might want to do a number of things with. We might want to write that message to disk, so we can recover it at some point in the future. We might want to send it to another server so that we can have hot failover. We might also have to turn that stream of bytes into an object that exists in the programming language we're dealing with, so it actually processes the message at that point in time. Now all three of those things are completely independent and can happen at the same time, but we found that with the previous model, we were doing one, then the next, followed by the next, all of which was adding latency.

So we asked ourselves how we could make all of this happen at the same time without the cost of having queues between the different stages. Or how do we just make queues much faster? We were kicking around these ideas for a while and we did manage to create new types of queues, new implementations that were much faster than the standard available ones, but they still had issues and we couldn't run all these steps in parallel.

Then eventually we came up with the design that ultimately became the Disruptor.

***HFTR**: I understand that one of the key problems you faced was how to deal with contention, is that correct?*

MT: Yes, as with anything in life where two or more things are contending to use any given resource, you have to work out how to arbitrate the contention. With a queue, you've fundamentally got two "writers", because you've got something putting things on to a queue (the producer) and something taking things off the queue (the consumer). In fact you can have multiple producers and multiple consumers. They all modify the queue structure by either adding things into it or removing things out of it, and this gives you a contention problem. You need to manage all of the interaction that's going on and you can do that in one of two ways.

One way to manage that interaction is to use a lock around the whole structure. What basically happens here is you need to get arbitration involved, i.e. you go down to the operating system kernel, which says, "You've got the lock first so you can continue to run", then to the other thread, "but you're second so I'm going to put your process to sleep until the first guy has finished and released the lock". That requires a lot of overhead because of things like ring transitions in the processor for letting the kernel run in privileged mode to do the stop & start on the threads and so on. Now the two threads will have pulled data into their caches (and caches are usually accessed more than one or two orders of magnitude faster than main memory is accessed); when the kernel gets involved, it has to pull its own data and ends up polluting the cache, losing the data that's been "warmed up" by the previous two threads. So when those previous two threads go to run again, they end up having to pull their data through from main memory once again. That adds a lot of overhead, and with that overhead can come significant latency.

***HFTR**: I believe you actually have some figures in your white paper (http://disruptor.googlecode.com/files/Disruptor-1.0.pdf) on how much latency is actually introduced by these processes?*

MT: Yes, there's one section where I run some comparisons: updating a variable; updating a variable with a lock; two variables contending on a lock, etc. For example, if I want to increment a single counter, I can do it 500,000,000 times in 300 milliseconds when there's no locking and

nothing else involved. If I end up with two threads contending to update that counter and using locks, it's 224,000 milliseconds, which is nearly four minutes! At that high level of contention, the pain and expense is obvious.

The second way to manage contention in a queue is to use CAS operations (Compare and Swap) where you say, for a single CPU instruction, "I want to set a variable to this value if its current value is what I expect it to be". When you do that you don't get the kernel involved for arbitration, you're just using the CAS operations. But it's important that these two things happen together because somebody else could have snuck in and done a piece of work underneath you. The problem is, with something queue-based, it's going to take a minimum of two CAS operations to do anything. And CAS operations are fairly expensive themselves, usually an order of magnitude more expensive than just doing a simple operation. On top of that, you're still fundamentally trying to touch the same structures; imagine two processes going for something, one gets it with the CAS and the other one misses it so has to keep spinning and waiting until the other one concludes. OK, the kernel hasn't got involved but the CPU and underlying infrastructure still have to fight between these two, trying to co-ordinate them for that piece of memory, which is bouncing backwards and forwards between the two of them, fighting for ownership all of the time in the CPU cache.

The point is, when you've got contention, you've got a problem. With CAS, you might not have the hugely expensive arbitrator in place (i.e. the kernel), but you've still got the problem where two things are fighting over the same thing. A better solution would be to look at how to remove that contention problem altogether. And that's what we've done with the Disruptor.

***HFTR**: So what are the key, fundamental elements of the Disruptor's design. How is it different?*

MT: Well first of all, for any piece of data in our system only one thread ever owns it for write - it's fine if everything else reads it, you can have a lot of readers - but as long as any given thing can only be modified by one entity, you don't have a contention problem.

What's great about this approach is that it gives you very predictable latency. Because you're never having that clash, you always know exactly how long it's going to take. It's also very simple. You just apply one basic rule, which is that any piece of data can only be updated by one

thread. All threads can read it, but only one can update it. So you have to look at how you can split your problem to follow that rule.

In nature this happens all the time with complex adaptive systems, which is where a very complex entity can exist based upon some very simple rules. Ant colonies are a really good example of this. There isn't a great pile of legal proceedings and instructions for all the ants to behave in a certain way, they all just follow a basic set of rules and a very complex system evolves, and it works. That can be applied in system design too. Humans always seem to want to make things more complex, they're almost afraid to just take a step back and say, don't do things in a complicated way, have the confidence of your convictions and just do things in a simple way and see what emerges.

In fact, there's a technique called emergent design, which is where you start following simple rules and wait to see what designs emerge. Sometimes really beautiful and elegant designs emerge that solve the problem by letting nature take its course. A lot of the design of the Disruptor has evolved that way. We decided we were going to follow some really simple rules, i.e. we were going to try to drive everything to run in parallel, and have it so that only one thread could ever modify any single piece of data.

Eventually, in the end, those rules evolved into the Disruptor.

***HFTR**: One of the key design elements of the Disruptor is the ring buffer. Can you tell us more about how you use that?*

MT: Yes, the ring buffer is at the centre of it. The way we get around the problem of the "queue reads" that modify is to never remove elements from the ring buffer; you just claim the next slot then you copy in your data. We pre-allocate a nice big ring and just go around re-using it over and over again. Because we don't remove anything from it, only the producers can actually write to it and the consumers never modify anything. As a result, you don't have that write contention problem we've been talking about.

***HFTR**: So if you're over-writing data as you go around the ring buffer, what happens if you need to read something from earlier that's now been over-written?*

MT: It's not an issue because we allocate very large ring buffers in the first place. The way the producers and the consumers work is as follows. The consumers want to consume from the ring buffer as soon as possible, so our consumers are ready to go all of the time. Whenever a new item is produced into the system, for example an order that comes in from the network, the producer puts it into the ring buffer so that consumers are ready to consume it and take it straight away to process it. If the producers were forced to go fast all of a sudden because they had just received a huge burst of orders, and the consumers were just working through as fast as possible, they'd just keep adding those to the ring. The problem you're asking about is what would happen if the ring would "wrap"? Well, the ring will eventually wrap but as long as the consumers are within the range of the size of the ring, it's fine. Both producers and consumers work on a constantly increasing sequence so the producers start at 1, 2, 3, and just keep counting up forever. Consumers do the same thing. As long as the consumers are no further behind the producers than the size of the ring, it doesn't matter if it wraps.

The way the Disruptor is designed is that the ring buffer is not accessed directly; it's always accessed through barriers. There's a producer barrier and a consumer barrier and what those barriers do is make sure, first of all, that the item you are about to consume is valid. For example, producers will not be allowed to overtake the consumers by greater size than the ring buffer; consumers can't consume something until you've actually produced yet. That way, the Disruptor co-ordinates the exchange of items between producers and consumers.

***HFTR**: If we look at the actual application of this technology, I understand you're using it for the matching engine at LMAX, but you're also using the Disruptor for doing real-time risk calculations too, is that correct?*

MT: Yes, it's a pattern we use in a number of places within our system, so on the matching engine side of things, as an order comes in, it gets matched against the book and an execution report comes out. It's a very similar pattern in our risk management system. As an order flows through it, we'll apply a delta for that order to the portfolio risk position for a given user and from there we know what the risk value is. In fact, the orders come in and flow first through our risk management engine, then through the order matching engine. They just happen in a pipeline and because they're very fast and hand off to each other, it adds minimal latency, so if the risk management engine decides that actually this user doesn't have sufficient margin to place a particular order, the order won't go through to the matching engine, it will send a rejection back with the reason explained.

***HFTR**: Are there other areas of functionality that you're starting to look at using this technology outside of the matching and the risk engine?*

MT: Yes, we're finding lots of uses for it. For example, monitoring traffic in and out of our exchange. We're looking at all orders coming in and execution reports going out. We need to measure the latency for the total time through the exchange, but we don't want to put the measurement in the exchange as that impact latency, so we measure from the front of our network, correlating packets coming in with packets going out. We used the Disruptor to write a network probe that takes all the Ethernet packets off our network, assembles TCP streams, assembles FIX sessions on top of that and then looks at the FIX messages for correlating orders to executions, to determine latency. We can also do monitoring for what sessions are live, what message rates we're seeing, and so on. We built all of this from scratch using the Disruptor. We wrote the probe in Java and when we set out to do that, I had a lot of people say, "You'll never write a network probe in Java, that's completely the wrong tool; it will never be able to keep up". It keeps up absolutely fine.

***HFTR**: What was your thinking behind open sourcing the core code for the Disruptor? How would you like to see it evolve?*

MT: As we started talking to customers and potential customers, we realised many people are building trading systems themselves that need to interact with an exchange. A lot of these people have some great trading ideas but they're struggling with what technology they should be using, what technology gives them the appropriate low latency. They read a lot about how important low latency is, but sometimes people are not quite sure where to start. So what we want to do, especially for our API-based traders, is offer out some of these pieces of technology, to give them a head start, so they can start thinking about their algorithms and not worry so much about the technology itself, they can focus on their trading strategies.

Generally most of the providers in this space have an interest in keeping their technology secret. Because we're open and we're enabling retail exchange trading in a way that does not take a position against our customers, we're in a unique position of where, if we stimulate the market it is to our advantage.

***HFTR**: Finally, do you anticipate that the design principles of the Disruptor will be picked up by other exchanges or MTFs? Or maybe even other vertical markets? If so, where would you initially see that happening?*

MT: It is a possibility. Whenever anything is open-sourced it could be applied to a range of end uses. I've had people contacting us and asking questions from quite diverse backgrounds, from people writing trading systems, to people writing games technology, to people interested in using this within telcos. So it's gaining interest from many different fields, but most of the people who have come back are people trading from home, running their own trading strategies. And it's been nice feedback for us to see that's what's been happening.

Basically, it provides a step forward for any system where you've got a performance and low latency need. Gaming is obviously interesting because latency is key when you've got a given frame rate targets in any given game. A nice side benefit to a lot of this is you can get significantly greater throughput than many of the traditional approaches, which means you need significantly less hardware to write a solution based on it, which also means it's got a nice "green" energy effect. As people start building systems using these types of techniques, they'll be spending less on hardware.

The key thing I've learned from all of this is to focus on the simplicity and not on complexity. By focusing on simplicity you can get a much more elegant solution that actually performs much better from a throughput and a latency perspective. We, as humans, seem to be lured to complexity like a moth to a flame, but by doing that, we always end up with very complex solutions that don't give us the latency and throughput that we could otherwise actually achieve. I like to strip things back. We use the term "mechanical sympathy" a lot at LMAX, where if you understand what the underlying hardware is doing, and you get it to do just the minimum it needs to do to carry out the task, you get great performance, and you can do it with a lot less hardware.

***HFTR**: Thank you Martin.*

- o -

Biography

Martin has had a passion for pushing software and electronics to the limit since childhood. He was the kid who literally took the video recorder apart and then fixed it. Since then he has always been attracted to business problems where high-performance computing can open new, previously impossible, opportunities. Ranging from PCs based stock market data feeds in the early 90s, the first generation of Internet banks, implementing the largest content management systems, the world's largest betting exchange at Betfair, and now founding LMAX. Martin brings his mechanical sympathy for hardware, to the software that he creates, which has taken him deep into the subjects of concurrency and parallel computing. Martin is the co-founder and CTO of LMAX where he leads building world's first retail focused financial exchange using a radical new architecture that takes it all back to basics.

