

Introduction

Michael Barker edited this page on Mar 2 2015 · 8 revisions


The best way to understand what the Disruptor is, is to compare it to something well understood and quite similar in purpose. In the case of the Disruptor this would be Java's [BlockingQueue](#). Like a queue the purpose of the Disruptor is to move data (e.g. messages or events) between threads within the same process. However there are some key features that the Disruptor provides that distinguish it from a queue. They are:

- Multicast events to consumers, with consumer dependency graph.
- Pre-allocate memory for events.
- Optionally lock-free.

Pages 13

Clone this wiki locally

<https://github.com/LMAX-Exc> 

 Clone in Desktop

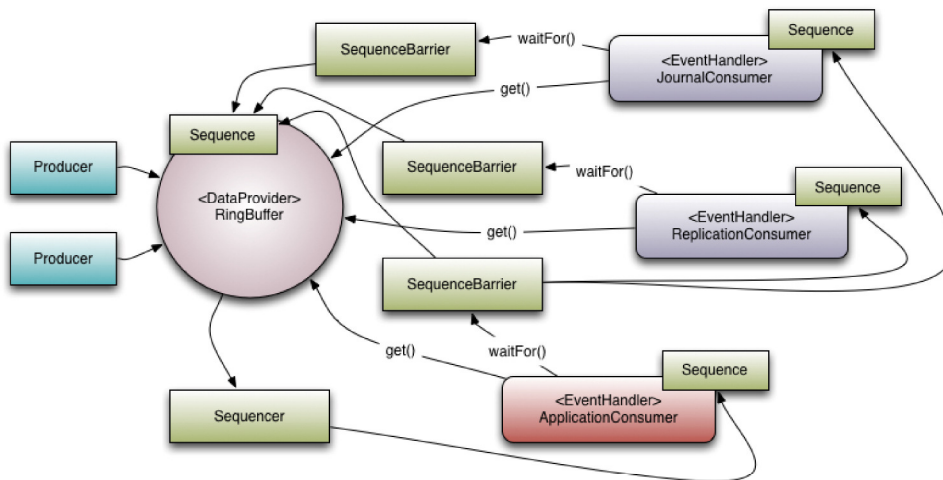
Core Concepts

Before we can understand how the Disruptor works, it is worthwhile defining a number of terms that will be used throughout the documentation and the code. For those with a DDD bent, think of this as the ubiquitous language of the Disruptor domain.

- **Ring Buffer:** The Ring Buffer is often considered the main aspect of the Disruptor, however from 3.0 onwards the Ring Buffer is only responsible for the storing and updating of the data (Events) that move through the Disruptor. And for some advanced use cases can be completely replaced by the user.
- **Sequence:** The Disruptor uses Sequences as a means to identify where a particular component is up to. Each consumer (EventProcessor) maintains a Sequence as does the Disruptor itself. The majority of the concurrent code relies on the the movement of these Sequence values, hence the Sequence supports many of the current features of an AtomicLong. In fact the only real difference between the 2 is that the Sequence contains additional functionality to prevent false sharing between Sequences and other values.
- **Sequencer:** The Sequencer is the real core of the Disruptor. The 2 implementations (single producer, multi producer) of this interface implement all of the concurrent algorithms use for fast, correct passing of data between producers and consumers.
- **Sequence Barrier:** The Sequence Barrier is produced by the Sequencer and contains references to the main published Sequence from the Sequencer and the Sequences of any dependent consumer. It contains the logic to determine if there are any events available for the consumer to process.
- **Wait Strategy:** The Wait Strategy determines how a consumer will wait for events to be placed into the Disruptor by a producer. More details are available in the section about being optionally lock-free.
- **Event:** The unit of data passed from producer to consumer. There is no specific code representation of the Event as it defined entirely by the user.
- **EventProcessor:** The main event loop for handling events from the Disruptor and has ownership of consumer's Sequence. There is a single representation called [BatchEventProcessor](#) that contains an efficient implementation of the event loop and will call back onto a used supplied implementation of the EventHandler interface.
- **EventHandler:** An interface that is implemented by the user and represents a consumer for the Disruptor.
- **Producer:** This is the user code that calls the Disruptor to enqueue Events. This concept also has no representation in the code.

To put these elements into context, below is an example of how LMAX uses the Disruptor within its high performance core services, e.g. the exchange.

Figure 1. Disruptor with a set of dependent consumers.



Multicast Events

This is the biggest behavioural difference between queues and the Disruptor. When you have multiple consumers listening on the same Disruptor all events are published to all consumers in contrast to a queue where a single event will only be sent to a single consumer. The behaviour of the Disruptor is intended to be used in cases where you need to independent multiple parallel operations on the same data. The canonical example from LMAX is where we have three operations, journalling (writing the input data to a persistent journal file), replication (sending the input data to another machine to ensure that there is a remote copy of the data), and business logic (the real processing work). The Executor-style event processing, where scale is found by processing different events in parallel at the same is also possible using the [WorkerPool](#). Note that is bolted on top of the existing Disruptor classes and is not treated with the same first class support, hence it may not be the most efficient way to achieve that particular goal.

Looking at Figure 1. is possible to see that there are 3 Event Handlers listening (JournalConsumer, ReplicationConsumer and ApplicationConsumer) to the Disruptor, each of these Event Handlers will receive all of the messages available in the Disruptor (in the same order). This allow for work for each of these consumers to operate in parallel.

Consumer Dependency Graph

To support real world applications of the parallel processing behaviour it was necessary to support co-ordination between the consumers. Referring back to the example described above, it necessary to prevent the business logic consumer from making progress until the journalling and replication consumers have completed their tasks. We call this concept gating, or more correctly the feature that is a super-set of this behaviour is called gating. Gating happens in two places. Firstly we need to ensure that the producers do not overrun consumers. This is handled by adding the relevant consumers to the Disruptor by calling `RingBuffer.addGatingConsumers()`. Secondly, the case referred to previously is implemented by constructing a `SequenceBarrier` containing `Sequences` from the components that must complete their processing first.

Referring to Figure 1, there are 3 consumers listening for Events from the Ring Buffer. There is a dependency graph in this example. The `ApplicationConsumer` depends on the `JournalConsumer` and `ReplicationConsumer`. This means that the `JournalConsumer` and `ReplicationConsumer` can run freely in parallel with each other. The dependency relationship can be seen by the connection from the `ApplicationConsumer`'s `SequenceBarrier` to the Sequences of the `JournalConsumer` and

ReplicationConsumer. It is also worth noting the relationship that the Sequencer has with the downstream consumers. One of its roles is to ensure that publication does not wrap the Ring Buffer. To do this none of the downstream consumer may have a Sequence that is lower than the Ring Buffer's Sequence less the size of the Ring Buffer. However using the graph of dependencies an interesting optimisation can be made. Because the ApplicationConsumers Sequence is guaranteed to be less than or equal to JournalConsumer and ReplicationConsumer (that is what that dependency relationship ensures) the Sequencer need only look at the Sequence of the ApplicationConsumer. In a more general sense the Sequencer only needs to be aware of the Sequences of the consumers that are the leaf nodes in the dependency tree.

Event Preallocation

One of the goals of the Disruptor was to enable use within a low latency environment. Within low-latency systems it is necessary to reduce or remove memory allocations. In Java-based system the purpose is to reduce the number stalls due to garbage collection (in low-latency C/C++ systems, heavy memory allocation is also problematic due to the contention that be placed on the memory allocator).

To support this the user is able to preallocate the storage required for the events within the Disruptor. During construction and EventFactory is supplied by the user and will be called for each entry in the Disruptor's Ring Buffer. When publishing new data to the Disruptor the API will allow the user to get hold of the constructed object so that they can call methods or update fields on that store object. The Disruptor provides guarantees that these operations will be concurrency-safe as long as they are implemented correctly.

Optionally Lock-free

Another key implementation detail pushed by the desire for low-latency is the extensive use of lock-free algorithms to implement the Disruptor. All of the memory visibility and correctness guarantees are implemented using memory barriers and/or compare-and-swap operations. There is only one use-case where a actual lock is required and that is within the BlockingWaitStrategy. This is done solely for the purpose of using a Condition so that a consuming thread can be parked while waiting for new events to arrive. Many low-latency systems will use a busy-wait to avoid the jitter that can be incurred by using a Condition, however in number of system busy-wait operations can lead to significant degradation in performance, especially where the CPU resources are heavily constrained. E.g. web servers in virtualised-environments.

