**Disruptor:**

High performance alternative to bounded queues for exchanging data between concurrent threads

Martin Thompson (CTO)

Dave Farley (Head of Software Development)

Michael Barker (Lead Developer)

May-2011

<http://code.google.com/p/disruptor/>

# Abstract

The Disruptor is a new concurrent programming framework for exchanging data between concurrent execution contexts in a shared memory system. The most common mechanism used to achieve such exchange is through the use of bounded queues. However at the most fundamental level, all implementations of bounded queues present problems of contention on modern memory systems and so place limits on performance.

Queues are commonly used in real world systems to build complex topologies of dependencies between processing nodes running on separate threads.

The Disruptor can handle a variety of cases. At its simplest it can act as a more cache-friendly replacement for a simple queue when used with a single producer and single consumer. However its primary value is when used to replace whole graphs of dependencies more conventionally separated by queues.

The Disruptor is designed with a strong focus on separating the concerns of data storage, producer synchronisation, and consumer synchronisation. These concerns are normally conflated with queue-based implementations. By avoiding this conflation, the concerns can be managed independently resulting in greater flexibility in both design and usage, and in the establishment of a new standard in performance. The resulting implementation manifests this performance as greater throughput with lower and more predictable latency.

# Overview

The Disruptor is the result of our efforts to build the World’s highest performance financial exchange at LMAX. Early designs focused on architectures derived from SEDA[[1]](#endnote-1) and Actors[[2]](#endnote-2) using pipelines for throughput. After profiling various implementations it became evident that the queuing of events between stages in the pipeline was dominating the costs. We found that queues also introduced latency and high levels of jitter. We expended significant effort on developing new queue implementations with better performance. However it became evident that queues as a fundamental data structure are limited due to the conflation of design concerns for the producers, consumers, and their data storage. The Disruptor is the result of our work to build a concurrent structure that cleanly separates these concerns.

# The Complexities of Concurrency

Concurrent execution of code is about two things, isolation of change and visibility of change. Isolation is simple if you keep separate copies of data and work on them independently as opposed to contending on the same data for mutation which happens in a queue implementation. Visibility of change is the source of complexity.

The most costly operation in any concurrent environment is a contended write access to any resource. The resource may be a database, file, socket or even a location in memory. To have multiple threads write to the same resource requires complex and expensive coordination. Typically this is achieved by employing a locking strategy.

## The Cost of Locks

Locks provide mutual exclusion and ensure that the visibility of changes occur in an ordered manner. Locks are incredibly expensive because they require arbitration when contended. This arbitration is achieved by a context switch to the operating system kernel which will suspend threads waiting on a lock until it is released. Such context switches are enormously expensive. As well as releasing control to the operating system, which may decide to do other house-keeping tasks while it has control, such a context switch can loose previously cached data and instructions. This can have a serious performance impact on modern processors. Fast user mode locks can be employed but these are only of benefit when not contended. In a queue-based high throughput system the most common en-queuing/de-queuing case will involve contended locks.

We will illustrate the cost of locks with a simple demonstration. The experiment is to call a function which increments a 64-bit counter in a loop 500 million times. This can be executed by a single thread on a 2.4Ghz Intel Westmere EP in just 300ms if written in Java. The language is unimportant to this experiment and will be similar across all languages with the same basic primitives.

Once a lock is introduced to provide mutual exclusion the cost goes up significantly and even more so when two or more threads contend as the table below shows.

|  |  |
| --- | --- |
| **Method** | **Time (ms)** |
| Single thread | 300 |
| Single thread with lock | 10,000 |
| Two threads with lock | 224,000 |
| Single thread with CAS | 5,700 |
| Two threads with CAS | 30,000 |
| Single thread with volatile write | 4,700 |

## The Costs of “CAS”

A more efficient alternative to locks can be employed for updating single memory words. These alternatives are based upon atomic or interlocked instructions implemented in modern processors. These are commonly known as CAS (Compare And Swap) operations, e.g. “*lock cmpxchg*” on x86. A CAS operation is a special machine code instruction that allows a word in memory to be conditionally set as an atomic operation. For the “increment a counter experiment” each thread can spin in a loop reading the counter then try to atomically set it to its new incremented value. The old and new values are provided as parameters to this instruction. If when the operation is executed the value of the counter matches the supplied expected value the counter is updated with the new value. If, on the other hand, the value is not as expected, the CAS operation will fail. It is then up to the thread attempting to perform the change to retry, re-reading the counter incrementing from that value and so on until the change succeeds. This CAS approach is significantly more efficient than locks because no context switch to the kernel is required for arbitration. However CAS operations are not free of cost. The processor must lock its instruction pipeline to ensure atomicity and employ a memory barrier to make the changes visible to other threads. CAS operations are available in Java by using the *java.util.concurrent.Atomic\** classes.

If the critical section of the program is more complex than a simple increment of a counter it may take a complex state machine using multiple CAS operations to orchestrate the contention. Developing concurrent programs using locks is difficult; developing lock-free algorithms using CAS operations and memory barriers is many times more complex and very difficult to prove that they are correct.

The ideal algorithm would be one with only a single thread owning all writes to a single resource with other threads reading the results. To read the results in a multi-processor environment requires memory barriers to make the changes visible to threads running on other processors.

## Memory Barriers

Modern processors perform out-of-order execution of instructions and out-of-order loads and stores of data between memory and execution units for performance reasons. The processors need only guarantee that program logic produces the same results regardless of execution order. While this is not an issue for single threaded programs it is important that, when threads share state, all memory changes appear in order at the point where needed for the data exchange to be successful. Memory barriers are used by processors to indicate sections of code where the ordering of memory updates is important. They are the means by which hardware ordering and visibility of change is achieved between threads. These hardware barriers are in addition to similar software barriers that compilers put in place to ensure ordering of compiled code.

Modern CPUs are now much faster than the current generation of memory systems. To bridge this divide CPUs use complex cache systems which are effectively fast hardware hash tables without chaining. These caches are kept coherent with other processor cache systems via message passing protocols. In addition processors have “store buffers” to offload writes to these caches and “invalidate queues” so that the cache coherency protocols can work efficiently when a write is about to happen.

What this means for data is that the latest version of any value could, at any stage after being written, be in a register, a store buffer, one of many layers of cache, or in main memory. If threads are to share this value, it needs to be made visible in an ordered fashion and this is achieved through the coordinated exchange of cache coherency messages. The timely generation of these messages can be controlled by memory barriers.

A read memory barrier orders load instructions on the CPU that executes it by marking a point in the invalidate queue for changes coming into its cache. This gives it a consistent view of the world for write operations ordered before the read barrier.

A write barrier orders store instructions on the CPU that executes it by marking a point in the store buffer, thus flushing writes out via its cache. This barrier gives an ordered view to the world of what store operations happen before the write barrier.

A full memory barrier orders both loads and stores but only on the CPU that executes it.

Some CPUs have more variants in addition to these three primitives but they are sufficient to understand the complexities of what is involved. In the Java memory model the read and write of a *volatile* field implements the read and write barriers respectively. This was made explicit in the Java Memory Model[[3]](#endnote-3) as defined with the release of Java 5.

## Cache Lines

The way in which caching is used in modern processors is of immense importance to successful high performance operation. Such processors are enormously efficient at churning through data and instructions held in cache and yet, comparatively, are massively inefficient when cache miss occurs.

Our hardware does not move memory around in bytes or words. For efficiency caches are organised into cache lines that are typically 32-256 bytes in size with the most common being 64 bytes. This is the level of granularity at which cache coherency protocols operate. What this means for data contention is that if two variables are in the same cache line and they are written to by different threads then they present the same problems of write contention as if they were a single variable. This is a concept know as “false sharing”. Therefore it is important to ensure that independent, but concurrently written, variables do not sure the same cache line if contention is to be minimised.

When accessing memory in a predictable manner CPUs are able to hide the latency cost of accessing main memory by predicting which memory is likely to be accessed next and pre-fetching it into the cache in the background. This only works if the processors can detect a pattern of access such as walking memory with a predictable “stride”. When iterating over the contents of an array the stride is predictable and so memory will be pre-fetched in cache lines, maximizing the efficiency of the access. However data structures like linked lists and trees tend to have nodes that are randomly allocated in memory with no predictable stride of access and so at the level of processor memory access can be more than two orders of magnitude less efficient.

## The Problems of Queues

Queues are typically implemented as either linked-lists or backed by arrays. If an in-memory queue is allowed to be unbounded then for many classes of problem it can grow unchecked until it reaches the point of catastrophic failure by exhausting memory. This happens when producers outpace the consumers. Unbounded queues can be useful in systems where the consumers are guaranteed to outpace the producers and memory is a precious resource, but there is always a risk if this assumption doesn’t hold. To avoid this catastrophic outcome, queues are commonly constrained in size (bounded). Keeping a queue bounded requires that it is either array-backed or that the size is tracked.

Queue implementations tend to have write contention on the head, tail, and size variables. In use queues are typically always close to full or empty due to the differences in pace between consumers and producers, they very rarely operate in a balanced middle ground where the rate of production and consumption is evenly matched. This propensity to be always full or always empty results in high levels of contention and/or expensive cache coherence. The problem is that even when the head and tail mechanisms are separated using different concurrent objects such as locks or CAS variables, they generally occupy the same cache-line.

The concerns of managing producers claiming the head of a queue, consumers claiming the tail, and the storage of nodes in between make the designs of concurrent implementations very complex to manage beyond using a single large-grain lock on the queue. A large grain lock on the whole queue for *puts* and *takes* is simple to implement but represents a significant bottleneck to throughput. If the concurrent concerns are teased apart within the semantics of a queue then the implementations become very complex for anything other than a single producer – single consumer implementation.

In Java there is a further problem with the use of queues in that they are significant sources of garbage, objects have to be allocated and placed in the queue as well as, if linked-list backed, then additional allocation of objects representing the nodes themselves, all of which need to be reclaimed.

## Pipelines and Graphs

For many classes of problem it makes sense to be wire together several processing stages into pipelines. Such pipelines often have parallel paths, being organised into graph-like topologies. The links between each stage are often implemented by queues with each stage having its own thread.

This approach comes at a cost, for each stage we have to incur the cost of en-queuing and de-queuing units of work. The number of targets multiplies this cost when the path must fork, and incurs an inevitable cost of contention when it must re-join after such a fork.

It would be ideal if the graph of dependencies could be expressed without incurring the cost of putting the queues between stages.

# The Design of the LMAX disruptor

The LMAX disruptor is designed to address these all of the issues outlined above and to attempt to maximize the efficiency of memory allocation, and operate in a cache-friendly manner so that it will perform optimally on modern hardware.

At the heart of the disruptor mechanism sits a pre-allocated bounded data structure in the form of a ring-buffer. Data is added to the ring buffer through one or more producers and processed by one or more consumers.

## Memory Allocation

All memory for the ring buffer is pre-allocated on start up. A ring-buffer can store either an array of pointers to entries or as an array of structures representing the entries, though in the Java implementation the limitations of the language mean that entries are associated with the ring-buffer as pointers to objects. Each of these entries is typically not the data being passed itself, but a container for it. This allows for the entries themselves to be pre-allocated eliminating any issues in languages that support garbage collection, since the entries will be re-used and live for the duration of the use of the Disruptor. The memory for these entries is allocated at the same time and it is highly likely that it will be laid out contiguously in main memory and so support cache striding. There is a proposal by John Rose to introduce “value types”[[4]](#endnote-4) to the Java language which would allow arrays of tuples, like other languages such as C, and so ensure that memory would be allocated contiguously and avoid the pointer indirection.

Garbage collection can be problematic when developing low-latency systems in a managed runtime environment like Java. The more memory that is allocated the greater the burden this puts on the garbage collector. Garbage collectors work at their best, when the lives of objects are either very short-lived or effectively immortal. The pre-allocation of memory in the ring buffer, and for its entries, means that it is, as far as garbage collection is concerned, immortal and so represents little burden.

Under heavy load queue-based systems can back up, this can lead to a reduction in the rate of processing and result in the allocated objects surviving longer than they should and being promoted beyond the young generation with generational garbage collectors. This has two implications. First, the objects have to be copied between generations which cause latency jitter. Second, these objects have to be collected from the old generation which is typically a much more expensive operation and increases the likelihood of “stop the world” pauses that result when the fragmented memory space requires compaction. In large memory heaps this can cause pauses of seconds per GB in duration.

## Birth of the Disruptor

While trying to address the problems described above a design emerged from a concentration on the rigorous separation of the concerns that we saw as conflated in queues. This approach was combined with a focus of ensuring that any data should be owned by only one thread for write access, therefore eliminating write contention. That design became known as the “Disruptor”. It was so named because it had elements of similarity for dealing with graphs of dependencies to the concept of “Phasers”[[5]](#endnote-5) in Java 7, introduced to support Fork-Join. The temptation to make the obvious Star Trek pun was too great, compounded with the fact that we wanted to disrupt current thinking about concurrent programming.

## Teasing Apart the Concerns

We saw the following concerns as being conflated in all queue implementations, to the extent that this collection of behaviours, that we saw as distinct, tend to define the interfaces that queues implement:

1. Storage of items being exchanged
2. Coordination of producers claiming the next sequence for exchange
3. Coordination of consumers being notified that a new item is available

When designing a financial exchange in a language that uses garbage collection, too much memory allocation can be problematic. So, as we have described linked-list backed queues are a not a good approach. Garbage collection is minimized if the entire storage for the exchange of data between processing stages can be pre-allocated. Further if this allocation can be performed in a uniform chunk then traversal of that data will be done in a manner that is very friendly to the caching strategies employed by modern processors. A data-structure that meets this requirement is an array with all the slots pre-filled. On creation of the ring buffer the Disruptor utilises the abstract factory pattern to pre-allocate the entries. When an entry is claimed, a producer can copy its data into the pre-allocated structure. The ring buffer should be sized to be a power of 2 as this allows a bit mask be used for the mod operation rather than a divide operation on access.

As we described earlier bounded queues suffer from contention at the head and tail of the queue. The ring buffer data structure is free from this contention and concurrency primitives because these concerns have been teased out into producer and consumer barriers through which the ring buffer must be accessed. The logic for these barriers is described below.

In most common usages of the Disruptor there is commonly only one producer. Typical producers are file readers or network listeners. In cases where there is a single producer there is no contention on sequence/ entry allocation and so allocation is un-contended.

In more unusual usages where there are multiple producers, producers will race one another to claim the next entry in the ring-buffer. Contention on claiming the next available entry can be managed with a simple CAS operation on the sequence number for that slot.

Once a producer has copied the relevant data to the claimed entry it can make it public to consumers by committing the sequence. This can be done without CAS by a simple busy spin until the other producers have reached this sequence in their own commit so this producer can then advance the cursor signifying the next available entry in the ring buffer for consumption. Producers can avoid wrapping the ring by tracking the sequence of consumers as a simple read operation before they write to the ring buffer.

Consumers wait for a sequence to become available in the ring buffer before they read the entry. Various strategies can be employed while waiting. If CPU resource is precious they can wait on a condition variable within a lock that gets signalled by the producers. This obviously is a point of contention and only to be used when CPU resource is more important than latency or throughput. The consumers can also loop checking the cursor which represents the currently available sequence in the ring buffer. This could be done with or without a thread yield by trading CPU resource against latency. This scales very well as we have broken the contended dependency between the producers and consumers if we do not use a lock and condition variable. Lock free multi-producer – multi-consumer queues do exist but they require multiple CAS operations on the head, tail, size counters. The Disruptor does not suffer this CAS contention.

## Sequencing

Sequencing is the core concept to how the concurrency is managed in the Disruptor. Each producer and consumer works off a strict sequencing concept for how it interacts with the ring buffer. Producers claim the next slot in sequence when claiming an entry in the ring. This sequence of the next available slot can be a simple counter in the case of only one producer or an atomic counter updated using CAS operations in the case of multiple producers. Once a sequence value is claimed, this entry in the ring buffer is now available to be written to by the claiming producer. When the producer has finished updating the entry it can commit the changes by updating a counter which represents the cursor on the ring buffer for what is the latest available entry to be consumed. This is a different counter for the next claim sequence used by the producers. The ring buffer cursor can be read and written in a bust spin by the producers using memory barrier without requiring a CAS operation as below.

long expectedSequence = claimedSequence – 1;

while (cursor != expectedSequence)

{

// busy spin

}

cursor = claimedSequence;

Consumers wait for a given sequence to become available by using a memory barrier to read the cursor. Once the cursor has been updated the memory barriers ensure the changes to the entries in the ring buffer are visible to the consumers who have waited on the cursor advancing.

Consumers each contain their own sequence which they update as they process entries from the ring buffer. These consumer sequences allow the producers to track consumers to prevent the ring from wrapping. Consumer sequences also allow consumers to coordinate work on the same entry in an ordered manner

In the case of having only one producer, and regardless of the complexity of the consumer graph, no locks or CAS operations are required. The whole concurrency coordination can be achieved with just memory barriers on the discussed sequences.

## Batching Effect

When consumers are waiting on an advancing cursor sequence in the ring buffer an interesting opportunity arises that is not possible with queues. If the consumer finds the ring buffer cursor has advanced a number of steps since it last checked it can process up to that sequence without getting involved in the concurrency mechanisms. This results in the lagging consumer quickly regaining pace with the producers when the producers burst ahead thus balancing the system. This type of batching increases throughput while reducing and smoothing latency at the same time. Based on our observations this effect results in a close to constant time for latency regardless of load up unto memory performance is saturated. This is very different to the “J” curve effect on latency we have observed with queues as load increases.

## Dependency Graphs

A queue represents the simple one step pipeline dependency between producers and consumers. If the consumers form a chain or graph like structure of dependencies then queues are required between the stages of the graph. This incurs the fixed costs of queues been applied many times within the graph of dependent stages. When designing the LMAX financial exchange our profiling showed how when taking a queue based approach resulted in queuing costs dominating the execution costs for processing an order representing a financial transaction.

Because the producer and consumer concerns are separated with the Disruptor pattern it is possible to have a complex graph of dependencies between consumers represented yet only using a single ring buffer at the core. This results in greatly reduced fixed costs of execution thus increasing throughput while reducing latency.

If a single ring buffer is used to sort entries in a complicated manner the entry can have a complex structure representing the whole workflow in a single cohesive place. Care must be taken in the design of such a structure so that the same state is not worked on by multiple consumers or false sharing of cache lines will occur. This is not as costly as concurrent write contention but still has an impact on throughput.

## Code Example

The code below is an example of a single producer and single consumer using the convenience interface *BatchHandler* for implementing a consumer. The consumer runs on a separate thread receiving entries as they become available.

// Callback handler which can be implemented by consumers

final BatchHandler<ValueEntry> batchHandler = new BatchHandler<ValueEntry>()

{

public void onAvailable(final ValueEntry entry) throws Exception

{

// process a new entry as it becomes available.

}

public void onEndOfBatch() throws Exception

{

// useful for flushing results to an IO device if necessary.

}

public void onCompletion()

{

// do any necessary clean up before shutdown

}

};

RingBuffer<ValueEntry> ringBuffer =

new RingBuffer<ValueEntry>(ValueEntry.ENTRY\_FACTORY, SIZE,

ClaimStrategy.Option.SINGLE\_THREADED,

WaitStrategy.Option.YIELDING);

ConsumerBarrier<ValueEntry> consumerBarrier = ringBuffer.createConsumerBarrier();

BatchConsumer<ValueEntry> batchConsumer =

new BatchConsumer<ValueEntry>(consumerBarrier, batchHandler);

ProducerBarrier<ValueEntry> producerBarrier = ringBuffer.createProducerBarrier(batchConsumer);

// Each consumer can run on a separate thread

EXECUTOR.submit(batchConsumer);

// Producers claim entries in sequence

ValueEntry entry = producerBarrier.nextEntry();

// copy data into the entry container

// make the entry available to consumers

producerBarrier.commit(entry);

## Disruptor Class Diagram

The core relationships in the Disruptor framework are depicted in the class diagram below. This diagram leaves out the convenience classes which can be used to simplify the programming model. After the dependency graph is constructed the programming model is simple. Producers claim entries in sequence via a *ProducerBarrier*, write their changes into the claimed entry, then commit that entry back via the *ProducerBarrier* making them available for consumption. As a consumer all one needs do is provide a *BatchHandler* implementation that receives call backs when a new entry is available. This resulting programming model is event based having a lot of similarities to the Actor Model.



# Throughput Performance Testing

As a reference we choose Doug Lea’s excellent *java.util.concurrent.ArrayBlockingQueue*[[6]](#endnote-6) which has the highest performance of any bounded queue based on our testing. The tests are conducted in a blocking programming style to match that of the Disruptor. The tests cases detailed below are available in the Disruptor open source project. **Note**: running the tests requires a system capable of executing at least 4 threads in parallel.

**Unicast: 1P – 1C**

**Three Step Pipeline: 1P – 3C**

**Sequencer: 3P – 1C**

**Multicast: 1P – 3C**

**Diamond: 1P – 3C**

For the above configurations an *ArrayBlockingQueue* was applied for each arc of data flow compared to barrier configuration with the Disruptor. The following table shows the performance results in operations per second using a Java 1.6.0\_25 64-bit Sun JVM, Windows 7, Intel Core i7 860 @ 2.8 GHz without HT and Intel Core i7-2720QM, Ubuntu 11.04, and taking the best of 3 runs when processing 500 million messages. Results can vary substantially across different JVM executions and the figures below are not the highest we have observed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Nehalem 2.8Ghz – Windows 7 SP1 64-bit** | | **Sandy Bridge 2.2Ghz – Linux 2.6.38 64-bit** | |
|  | **ABQ** | **Disruptor** | **ABQ** | **Disruptor** |
| Unicast: 1P – 1C | 5,339,256 | 25,998,336 | 4,057,453 | 22,381,378 |
| Pipeline: 1P – 3C | 2,128,918 | 16,806,157 | 2,006,903 | 15,857,913 |
| Sequencer: 3P – 1C | 5,539,531 | 13,403,268 | 2,056,118 | 14,540,519 |
| Multicast: 1P – 3C | 1,077,384 | 9,377,871 | 260,733 | 10,860,121 |
| Diamond: 1P – 3C | 2,113,941 | 16,143,613 | 2,082,725 | 15,295,197 |

# Latency Performance Testing

To measure latency we take the 3 step pipeline and generate events at less than saturation. This is achieved by waiting 1 microsecond after injecting an event before injecting the next and repeating 50 million times. To time at this level of precision it is necessary to use time stamp counters from the CPU. We choose CPUs with an invariant TSC because older processors suffered from changing frequency due to power saving and sleep states. Intel Nehalem and later processors use an invariant TSC which can be accessed by the latest Oracle JVMs running on Ubuntu 11.04. No CPU binding has been employed for this test.

For comparison we use the *ArrayBlockingQueue* once again. We could have used *ConcurrentLinkedQueue*[[7]](#endnote-7) which is likely to give better results but we want to use a bounded queue implementation to ensure producers do not outpace consumers by creating back pressure. The results below are for 2.2Ghz Core i7-2720QM running Java 1.6.0\_25 64-bit on Ubuntu 11.04.

Average latency per hop for the Disruptor comes out at 51 nanoseconds compared to 32,757 nanoseconds for *ArrayBlockingQueue*. Profiling shows the use of locks and signalling via a condition variable are the main cause of latency for the *ArrayBlockingQueue*.

|  |  |  |
| --- | --- | --- |
|  | **Array Blocking Queue (ns)** | **Disruptor (ns)** |
| Average Latency | 32,757 | 51 |
| 99% Latency | 314,510 | 128 |
| 99.99% Latency | 2,097,152 | 2,228 |

# Conclusion

The Disruptor is a major step forward for both increasing throughput and reducing latency in a concurrent environment. By separating the concerns normally conflated into queue implementations it allows for more flexible configurations when a graph of dependencies exist for processing a source of data via multiple execution stages.

Our testing shows that the batching effect described can introduce a new characteristic in high performance systems which usually suffer greater contention and increased latency under burst conditions is inverted with latency staying low until saturation occurs rather than the usual “J” curve experienced with queue based approaches.

1. Staged Event Driven Architecture – http://www.eecs.harvard.edu/~mdw/proj/seda/ [↑](#endnote-ref-1)
2. Actor model – http://dspace.mit.edu/handle/1721.1/6952 [↑](#endnote-ref-2)
3. Java Memory Model - http://www.ibm.com/developerworks/library/j-jtp02244/index.html [↑](#endnote-ref-3)
4. Value Types - http://blogs.oracle.com/jrose/entry/tuples\_in\_the\_vm [↑](#endnote-ref-4)
5. Phasers - http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/Phaser.html [↑](#endnote-ref-5)
6. ArrayBlockingQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ArrayBlockingQueue.html [↑](#endnote-ref-6)
7. ConcurrentLinkedQueue - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html [↑](#endnote-ref-7)