

Event Sourcing - Disruptor Pattern

Event Sourcing
Async Event Processing
Producer/Consumer – Disruptor Pattern

Abstract

Layered Architecture, Microkernel Architecture, Microservices Architecture and Event-Driven (Event-Sourcing) Architecture each solve specific challenges.

This document outlines a demo application that examines Event-Driven Architecture.

The central force in an Event-Driven Architecture is Event-Sourcing. Event-Sourcing requires a temporal *Message Broker*.

The absence of blocking data structures is what allows an Event-Sourcing platform to handle a massive number of client requests w/ one thread.

A Message Broker can be implemented by adopting ideas found in the *Disruptor Design Pattern*, which in turn leverages a data structure variously called a *Ring Buffer* or a *Circular Buffer*.

Mutable/immutable *Events* can be used in a *Producer/Consumer* fashion, but the important idea is that the core event-processing implementation employs a single thread (no locks, no synchronization).

The Disruptor concept has its roots in the LMAX trading Platform (UK) which was open-sourced by LMAX circa 2012 (it is a battle-tested idea).

Apache `log4j2` adopts the Disruptor pattern, via its *Disruptor.jar*, which enables `log4j2` to offer an asynchronous `logging` modality.

If combined with a *Correlation ID (CID)*, asynchronous `log4j2` can act as a *Replayable Journal* for an Event-Sourcing platform.

| Background

A Ring Buffer can be used to provide fast, [single-threaded Event Processing](#).

In order to prevent race conditions (thread safety) when using a single thread of execution, memory blocks must be aligned to assure that False Sharing does not occur in CPU Cache Lines (complicated by the fact that modern computers have multiple CPUs, each containing multi-cores).

At the lowest level, this means that memory must be “padded” to assure that word-tearing never happens.

The conceptualization was implemented by [LMAX*](#), a European commercial trading platform.

Open-sourced circa 2012, LMAX’s underlying *Disruptor* technology is used by [log4j2†](#) for its async logging feature/function via a binary distribution:

`Disruptor.jar`

Here are the Maven coordinates for log4j2 async enablement‡:

```
<dependency>
  <groupId>com.lmax</groupId>
  <artifactId>disruptor</artifactId>
  <version>3.2.1</version>
</dependency>
```

Similarly, here are the Maven coordinates which are needed by this ([Exchanger](#)) app’s *Disruptor/RingBuffer* async enablement§:

```
<dependencies>
  <dependency>
    <groupId>com.googlecode.disruptor</groupId>
    <artifactId>disruptor</artifactId>
    <version>2.10.4</version>
  </dependency>
```

Design, high-level

In each of the three interrelated apps, the `publish` (Bash script) enables the sharing of the functionality within the three interdependent modules. Sharing is accomplished via Maven `<dependency/>` elements.

The `Event` is presented to the `Exchanger` – by the action of data having been made accessible to an `EventProducer` callback method named `publisherCallback()`.

The `publisherCallback()` decides what kind/volume of data to insert introduce into the `Exchanger`'s Ring Buffer.

The `Exchanger` polls the `EventProducer.publisherCallback()` method, using a `while()` loop.

The demo application has two instances of `EventProducer`, and a single instance of `EventConsumer`. This proves the efficacy of the `Exchanger`'s Ring Buffer.

When `Exchanger` the detects that an `Event` has been added to its (internal) Ring Buffer, its sole (internal) `EventHandler` instance – of type `EventHandler<Event>` – responds by calling the sole method in `EventConsumer`:

```
take(Event)
```

This completes one event-sourcing cycle.

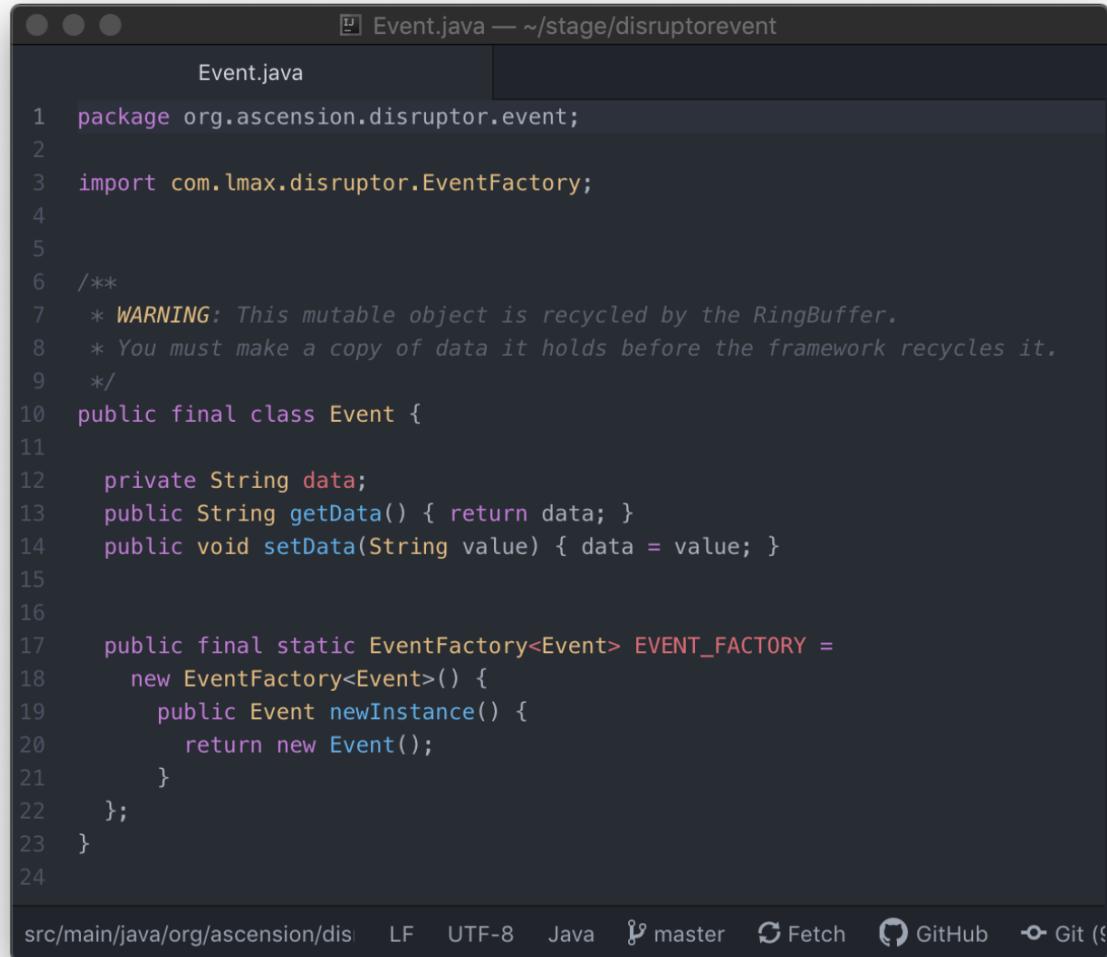
Event, detailed analysis

On lines 17-22, the `org.ascension.disruptor.event.Event.java` class wraps the canonical `EventFactory` class.

Within the `Event` class the inline `EventFactory` constructor is assigned to a `public static final` reference (of `EventFactory`).

This construct (routine) is called indirectly within the same class that constructs the Disruptor. In this application that class is:

`org.ascension.disruptor.event.engine.Exchanger.java`



The screenshot shows a terminal window with the title "Event.java — ~/stage/disruptorevent". The code displayed is as follows:

```
1 package org.ascension.disruptor.event;
2
3 import com.lmax.disruptor.EventFactory;
4
5
6 /**
7  * WARNING: This mutable object is recycled by the RingBuffer.
8  * You must make a copy of data it holds before the framework recycles it.
9  */
10 public final class Event {
11
12     private String data;
13     public String getData() { return data; }
14     public void setData(String value) { data = value; }
15
16
17     public final static EventFactory<Event> EVENT_FACTORY =
18         new EventFactory<Event>() {
19             public Event newInstance() {
20                 return new Event();
21             }
22         };
23 }
```

The terminal status bar at the bottom shows: "src/main/java/org/ascension/disruptor/event" and "LF UTF-8 Java ⚡ master ⚡ Fetch ⚡ GitHub ⚡ Git (9)".

Producer, detailed analysis

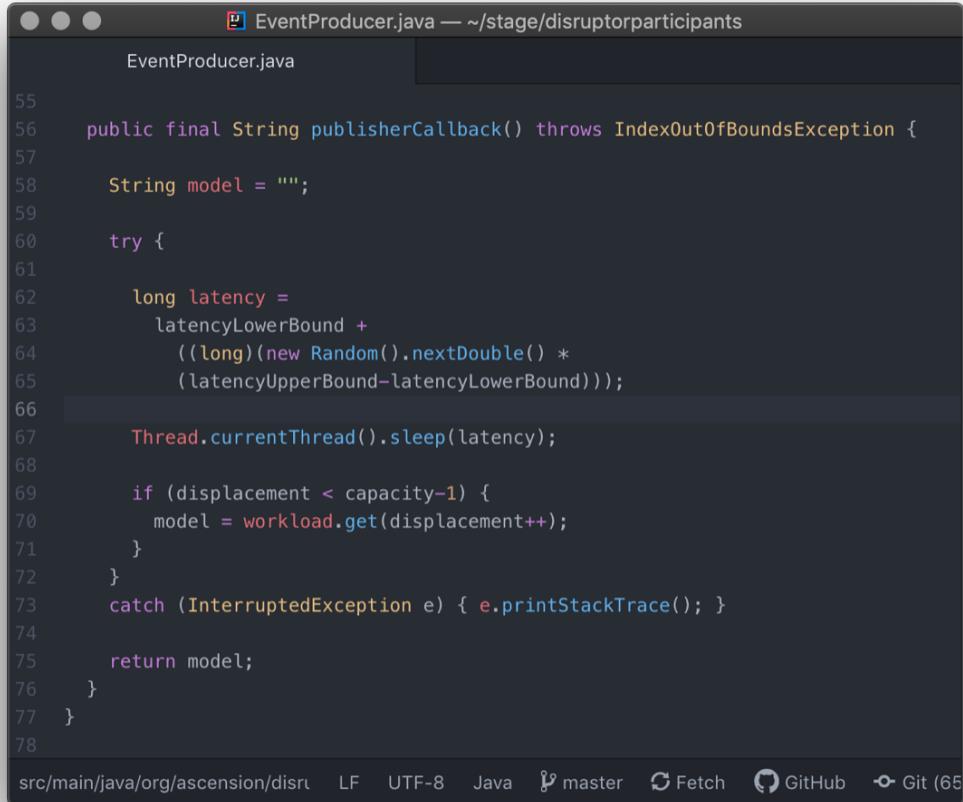
In the demo app a *Producer*, of type:

```
org.ascension.disruptor.event.producer.EventProducer.java
```

synthesizes a set of arbitrary `String` values into a model that is then published (in batch mode) as `Events` to the *RingBuffer* (the `Exchanger.java` class).

A callback method – `publisherCallback()` – introduces artificial latency into the publication. The latency simulates unpredictability (asynchronicity) into the response to the `Exchanger.java` instance's polling.

This arrangement proves the efficacy of the *RingBuffer* that is encapsulated within the `Exchanger` – there are two (2) *Producers* and `Events` from each interleave with one another||.



The screenshot shows a terminal window with the title "EventProducer.java — ~/stage/disruptorparticipants". The window contains the Java code for the `EventProducer` class. The code defines a public final string `model` and a publisher callback method. The callback uses a random sleep duration between `latencyLowerBound` and `latencyUpperBound`. It then checks if `displacement` is less than `capacity-1` and updates the `model` accordingly. A catch block handles `InterruptedException`. Finally, it returns the `model`. The code is annotated with line numbers from 55 to 78. At the bottom of the terminal window, there are status indicators: "src/main/java/org/ascension/disr", "LF", "UTF-8", "Java", "master", "Fetch", "GitHub", and "Git (65)".

```
55
56     public final String publisherCallback() throws IndexOutOfBoundsException {
57
58         String model = "";
59
60         try {
61
62             long latency =
63                 latencyLowerBound +
64                 ((long)(new Random().nextDouble() *
65                 (latencyUpperBound-latencyLowerBound)));
66
67             Thread.currentThread().sleep(latency);
68
69             if (displacement < capacity-1) {
70                 model = workload.get(displacement++);
71             }
72         } catch (InterruptedException e) { e.printStackTrace(); }
73
74         return model;
75     }
76 }
77 }
```

src/main/java/org/ascension/disr LF UTF-8 Java master Fetch GitHub Git (65)

Consumer, detailed analysis

The [EventConsumer](#) simply grabs [Event](#) data, via its single (*Interface*) method, with the signature:

```
take(Event)
```

It acquires [Event](#) data on demand.

The “demand” is a call which originates with the canonical `EventHandler.onEvent(...)` method being triggered, then calling the `public EventConsumer.take()` method (see Lines 24-6).

The canonical `EventHandler.onEvent(...)` method exists within the [Exchanger](#) instance and listens/responds to *RingBuffer* publication events.

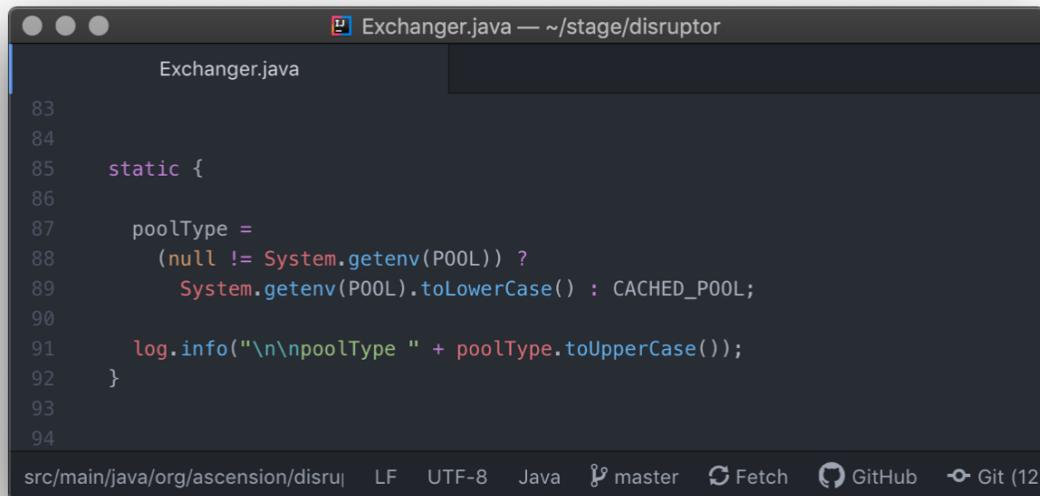
```
EventConsumer.java — ~/stage/disruptorparticipants
EventProducer.java | EventConsumer.java
7
8 /**
9  * Plays the role of a <i>Consumer</i> of an <i>Event</i>.
10 */
11 public final class EventConsumer implements Consumer {
12
13     private static final Logger log =
14         LogManager.getLogger(EventConsumer.class);
15
16     /*
17      * Lifecycle: This takes the place of a formal registration
18      * process with the EventProcessor (<code>Exchanger</code>)
19      */
20     public static final EventConsumer REF =
21         new EventConsumer();
22
23
24     public final void take(Event event) {
25         log.info("    Event.getData() " + event.getData());
26     }
27 }
28
```

src/main/java/org/ascension/disru LF UTF-8 Java ⚡ master Fetch GitHub Git (65)

Exchanger, detailed analysis

The [Exchanger](#) class encapsulates the implementation of the core *Event-Sourcing* elements.

The [Exchanger](#) class also observes the 12-Factor application credo when it allows the *Producer* thread pool model to be configured via the value of an environment variable:



```
Exchanger.java — ~/stage/disruptor
Exchanger.java
83
84
85     static {
86
87         poolType =
88             (null != System.getenv(POOL)) ?
89                 System.getenv(POOL).toLowerCase() : CACHED_POOL;
90
91         log.info("\n\npoolType " + poolType.toUpperCase());
92     }
93
94
```

src/main/java/org/ascension/disru| LF UTF-8 Java ⚡ master Fetch GitHub Git (12)

When run from within a shell, the [Exchanger](#) component's thread pool model can be specified – issue;

```
export POOL=CACHED_POOL
```

While the [Exchanger](#) component defaults to a thread pool value of `CACHED_POOL`, a CLI override to `SINGLE_POOL` can be done.

Semantics:

```
source ok C
source ok S
```

The arguments to the `ok` script are *case-insensitive*.

A value containing the letter `C` – `CACHED_POOL`

A value containing the letter `S` – `SINGLE_POOL`

The arguments to the `ok` script are *case-insensitive*.

The screenshot shows a terminal window with the following content:

```
Exchanger.java — ~/stage/disruptor
Exchanger.java

124
125     /*
126      * Preallocate the <i>RingBuffer</i>.|
127      */
128     disruptor =
129         new Disruptor<Event>(
130             Event.EVENT_FACTORY, SLOT_COUNT, execPool);
131
132
133     final EventHandler<Event> handler = new EventHandler<Event>() {
134
135         /*
136          * WARNING
137          * The <i>Event</i> is eventually recycled by the
138          * <i>Disruptor</i> when its <i>RingBuffer</i> wraps
139          */
140         public void onEvent(Event event, long seq, boolean endOfBatch) {
141
142             consumer.take(event);
143
144             log.info("\nSequence " + seq + " - Event: " + event.getData());
145         }
146     };
147
148
149     // Build dependency graph
150     disruptor.handleEventsWith(handler);
151
152     RingBuffer<Event> ringBuffer = disruptor.start();
153
```

At the bottom of the terminal window, there are several status indicators: src/main/java/org/ascension/disru, LF, UTF-8, Java, master, Fetch, GitHub, and Git (12).

The section of [Exchanger](#) code (above) registers a new instance of the (canonical) `EventHandler` (lines 133-46) with a new (canonical) `Disruptor` (lines 128-30) to create (a canonical) `RingBuffer` instance (line 152).

This operation also pre-allocates the memory for the slots in the `RingBuffer` (line 150).

The screenshot shows a terminal window with the title "Exchanger.java — ~/stage/disruptor". The code is a Java snippet for a "Two phase commit" process using a RingBuffer. It includes imports for java.util.concurrent.atomic.AtomicLong and org.apache.disruptor.Event, and uses annotations like @Stage and @Processor. The code handles a loop where it acquires a slot from the ring buffer, logs the slot ID, retrieves an event, and then publishes it to the ring buffer. Error handling is provided for IndexOutOfBoundsException.

```
156     int roundRobinUnweighted=0;
157
158     while (true) {
159
160         // Two phase commit, first acquire a RingBuffer slot
161         long slotID = ringBuffer.next();
162
163         log.info("[slot " + slotID + "]");
164
165         Event event = ringBuffer.get(slotID);
166
167         if (roundRobinUnweighted % 2 == 0) {
168             event.setData(producerOne.publisherCallback());
169         }
170         else {
171             event.setData(producerTwo.publisherCallback());
172         }
173
174         roundRobinUnweighted++;
175
176         ringBuffer.publish(slotID);
177     }
178 }
179
180 catch(IndexOutOfBoundsException e) {
```

src/main/java/org/ascension/disru LF UTF-8 Java ⚡ master Fetch GitHub Git (12)

In a polling loop, reference the next available *RingBuffer* slot (line 162), to access an [Event](#), line 166.

Based on a simplistic *Round-Robin* algorithm (line 168, 175), nominate one of two *Producers* from which to get its publishable data (lines 168-73).

Publish the (*Producer's*) payload to the *RingBuffer* slot (line 177).

Application(s), wiring together of

The three components in the demo app are independent, but interdependent.

The wiring of the three components – at a binary level – is accomplished using standard Maven dependencies.

[The mappings are shown immediately below.](#)

```
<groupId>org.ascension</groupId>
<artifactId>disruptorevent</artifactId>
<packaging>jar</packaging>
<version>1.0.0-SNAPSHOT</version>
```

The [Event](#) Coordinates

```
<groupId>org.ascension</groupId>
<artifactId>disruptorparticipants</artifactId>
<packaging>jar</packaging>
<version>1.0.0-SNAPSHOT</version>
```

The [Participant](#) Coordinates

```
<groupId>org.ascension</groupId>
<artifactId>disruptorapp</artifactId>
<packaging>jar</packaging>
<version>1.0.0-SNAPSHOT</version>
```

```
<dependency>
    <groupId>org.ascension</groupId>
    <artifactId>disruptorevent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>

<dependency>
    <groupId>org.ascension</groupId>
    <artifactId>disruptorparticipants</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

The [Exchanger](#) Coordinates, followed by its two (2) Dependency Coordinates

Application(s) Lifecycle, scripts

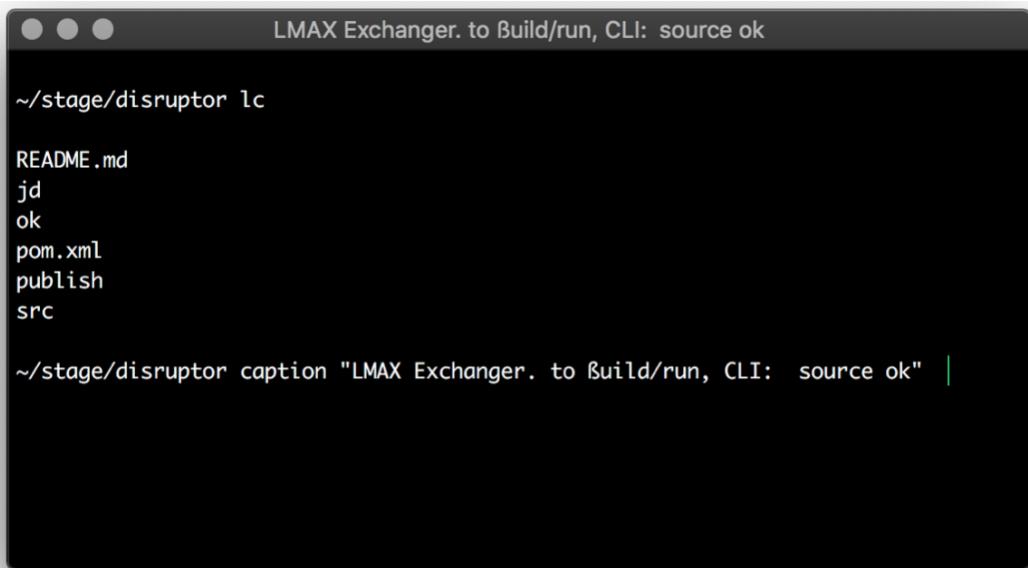
The three components/parts of the Event-Sourcing demo app all adopt the same build approach.

The [Disruptor](#) component, housing the `Exchanger.java` class, is the driver – it has a `main()` method. It has dependencies on both of the other components.

The other two application components – [Participants](#), [Event](#) – each have an additional script that “publishes” their Maven coordinates to the local repo, making them available to any other components (such as the [Disruptor/Exchanger](#) components). This script (w/o an extension) is named: `publish`

Each of the app components’ build scripts (w/o extension) are named: `ok`

The lifecycle scripts are shown immediately below.



A screenshot of a terminal window titled "LMAX Exchanger. to Build/run, CLI: source ok". The window shows the following directory listing:

```
~/stage/disruptor lc
README.md
jd
ok
pom.xml
publish
src
```

Below the listing, the command `~/stage/disruptor caption "LMAX Exchanger. to Build/run, CLI: source ok"` is entered, with the cursor at the end of the line.

Build/run – This is the main project (1/3), Exchanger (Disruptor).

This component of the application triad directly leverages LMAX technology.

The other two application components are: [Event](#), [Participants](#) (Producer/Consumer).

```
LMAX Event - to Build/run, CLI: source ok; source publish

~/stage/disruptorevent lc

README.md
html
jd
ok
pom.xml
publish
src

~/stage/disruptorevent caption "LMAX Event - to Build/run, CLI: source ok; source publish" |
```

Build/run – This is the [Event](#) project (2/3).

This part of the application triad is an intricate part of the LMAX technology; this event is an LMAX-aware event.

```
LMAX Producer/Consumer (Participants) - to Build/run, CLI: source ok; source publish

~/stage/disruptorparticipants lc

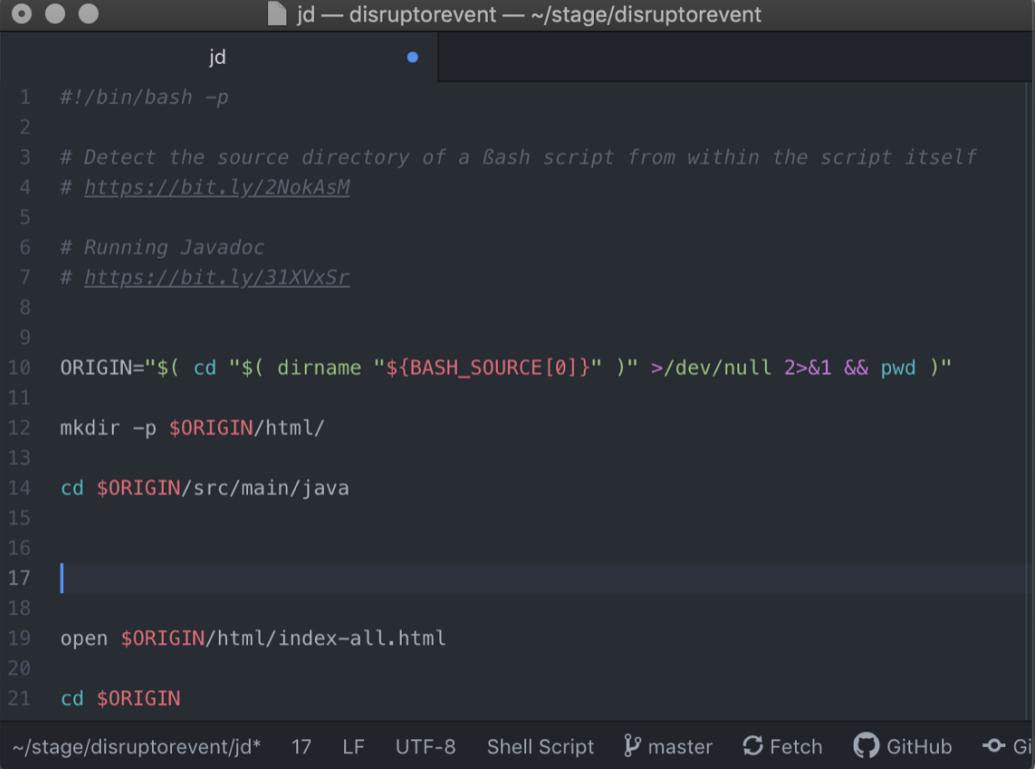
README.md
jd
ok
pom.xml
publish
src
Build.docx

~/stage/disruptorparticipants caption "LMAX Producer/Consumer (Participants) - to Build/run, CLI: source ok; source publish" |
```

Build/run – This is the [\(Producer/Consumer\)](#) Disruptor-centric part of the triad (3/3).

This part of the application triad dispatches/retrieves LMAX [Events](#).

Javadoc, CLI



```
jd — disruptorevent — ~/stage/disruptorevent
jd
1 #!/bin/bash -p
2
3 # Detect the source directory of a Bash script from within the script itself
4 # https://bit.ly/2NokAsM
5
6 # Running Javadoc
7 # https://bit.ly/31XVxSr
8
9
10 ORIGIN=$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )
11
12 mkdir -p $ORIGIN/html/
13
14 cd $ORIGIN/src/main/java
15
16
17 |
18
19 open $ORIGIN/html/index-all.html
20
21 cd $ORIGIN
```

~/stage/disruptorevent/jd* 17 LF UTF-8 Shell Script ⚡ master ⚡ Fetch ⚡ GitHub ⚡ Gi

Javadoc – Event component

```
jd
1 #!/bin/bash -p
2
3 # Detect the source directory of a Bash script from within the script itself
4 # https://bit.ly/2NokAsM
5
6 # Running Javadoc
7 # https://bit.ly/31XVxSr
8
9
10 ORIGIN=$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )"
11
12 cd $ORIGIN/src/main/java
13
14 javadoc -d $ORIGIN/html org.ascension.disruptor.event.producer org.ascension.disruptor.event.consumer
15
16 open $ORIGIN/src/main/java/html/index-all.html
17
18 cd $ORIGIN
19
```

~/stage/disruptorparticipants/jd* 18:12 LF UTF-8 Shell Script master Fetch GitHub Git (65)

Javadoc – Participants component

```
jd — disruptor — ~/stage/disruptor
jd

1 #!/bin/bash -p
2
3 v # Detect the source directory of a Bash script from within the script itself
4 # https://bit.ly/2NokAsM
5
6 v # Running Javadoc
7 # https://bit.ly/31XVxSr
8 |
9
10 ORIGIN=$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )"
11
12 cd $ORIGIN/src/main/java
13
14 javadoc -d $ORIGIN/html org.ascension.disruptor.event.engine
15
16 open $ORIGIN/src/main/java/html/index-all.html
17
18 cd $ORIGIN
19
```

~/stage/disruptor/jd 8:1 LF UTF-8 Shell Script ⚡ master ⚡ Fetch ⚡ GitHub ⚡ C

Javadoc – Exchanger component

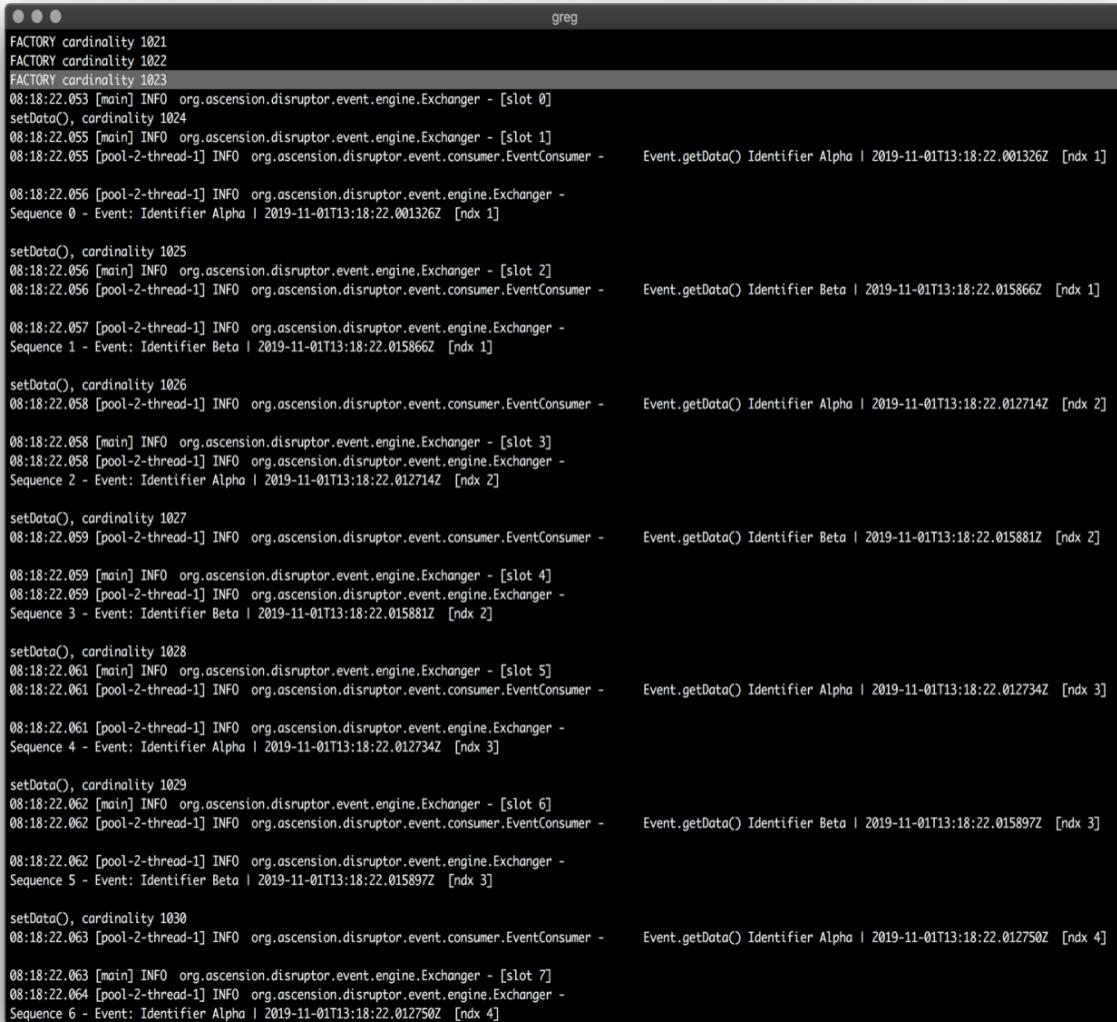
Application, running

The Disruptor first creates the specific number of slots [[Exchanger](#)].

As events stream into the [Exchanger](#), its internal instance of:

```
com.lmax.disruptor.EventHandler
```

... will acquire data from the slots which are populated w/ data from the [Exchanger](#)'s `com.lmax.disruptor.RingBuffer` performing polling (of a *Producer's* callback method) for `org.ascension.disruptor.event.Events`:



```
● ● ● greg
FACTORY cardinality 1021
FACTORY cardinality 1022
FACTORY cardinality 1023
08:18:22.053 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 0]
setData(), cardinality 1024
08:18:22.055 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 1]
08:18:22.055 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer - Event.getData() Identifier Alpha | 2019-11-01T13:18:22.001326Z [ndx 1]

08:18:22.056 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Sequence 0 - Event: Identifier Alpha | 2019-11-01T13:18:22.001326Z [ndx 1]

setData(), cardinality 1025
08:18:22.056 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 2]
08:18:22.056 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer - Event.getData() Identifier Beta | 2019-11-01T13:18:22.015866Z [ndx 1]

08:18:22.057 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Sequence 1 - Event: Identifier Beta | 2019-11-01T13:18:22.015866Z [ndx 1]

setData(), cardinality 1026
08:18:22.058 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer - Event.getData() Identifier Alpha | 2019-11-01T13:18:22.012714Z [ndx 2]

08:18:22.058 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 3]
08:18:22.058 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Sequence 2 - Event: Identifier Alpha | 2019-11-01T13:18:22.012714Z [ndx 2]

setData(), cardinality 1027
08:18:22.059 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer - Event.getData() Identifier Beta | 2019-11-01T13:18:22.015881Z [ndx 2]

08:18:22.059 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 4]
08:18:22.059 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Sequence 3 - Event: Identifier Beta | 2019-11-01T13:18:22.015881Z [ndx 2]

setData(), cardinality 1028
08:18:22.061 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 5]
08:18:22.061 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer - Event.getData() Identifier Alpha | 2019-11-01T13:18:22.012734Z [ndx 3]

08:18:22.061 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Sequence 4 - Event: Identifier Alpha | 2019-11-01T13:18:22.012734Z [ndx 3]

setData(), cardinality 1029
08:18:22.062 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 6]
08:18:22.062 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer - Event.getData() Identifier Beta | 2019-11-01T13:18:22.015897Z [ndx 3]

08:18:22.062 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Sequence 5 - Event: Identifier Beta | 2019-11-01T13:18:22.015897Z [ndx 3]

setData(), cardinality 1030
08:18:22.063 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer - Event.getData() Identifier Alpha | 2019-11-01T13:18:22.012750Z [ndx 4]

08:18:22.063 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 7]
08:18:22.064 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Sequence 6 - Event: Identifier Alpha | 2019-11-01T13:18:22.012750Z [ndx 4]
```

```

greg
Sequence 393 - Event:
setDataO, cardinality 1418
08:18:22.662 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 395]
08:18:22.662 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.662 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData() Identifier Alpha | 2019-11-01T13:18:22.015833Z [ndx 198]
Sequence 394 - Event: Identifier Alpha | 2019-11-01T13:18:22.015833Z [ndx 198]

setDataO, cardinality 1419
08:18:22.663 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 396]
08:18:22.663 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.663 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData()
Sequence 395 - Event:
setDataO, cardinality 1420
08:18:22.665 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 397]
08:18:22.665 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.665 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData() Identifier Alpha | 2019-11-01T13:18:22.015845Z [ndx 199]
Sequence 396 - Event: Identifier Alpha | 2019-11-01T13:18:22.015845Z [ndx 199]

setDataO, cardinality 1421
08:18:22.666 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 398]
08:18:22.666 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.666 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData()
Sequence 397 - Event:
setDataO, cardinality 1422
08:18:22.667 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 399]
08:18:22.667 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.667 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData()
Sequence 398 - Event:
setDataO, cardinality 1423
08:18:22.668 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 400]
08:18:22.668 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.668 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData()
Sequence 399 - Event:
setDataO, cardinality 1424
08:18:22.669 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 401]
08:18:22.669 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.669 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData()
Sequence 400 - Event:
setDataO, cardinality 1425
08:18:22.670 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 402]
08:18:22.670 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.670 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData()
Sequence 401 - Event:
setDataO, cardinality 1426
08:18:22.672 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 403]
08:18:22.672 [pool-2-thread-1] INFO org.ascension.disruptor.event.consumer.EventConsumer -
08:18:22.672 [pool-2-thread-1] INFO org.ascension.disruptor.event.engine.Exchanger - Event.getData()
Sequence 402 - Event:
setDataO, cardinality 1427
08:18:22.673 [main] INFO org.ascension.disruptor.event.engine.Exchanger - [slot 404]

```

As [Events](#) stop coming into the RingBuffer, its slots go empty (but the polling continues to fill slots, now with empty values). The memory for the slots is allocated at program inception, so it is only CPU cycles that are “wasted”.

CTL-Z or CTL-C terminates the sole thread-of-execution, but there is a `panic()` method that is unused, but that can gracefully cleanup application components and release memory.

Appendix

* The LMAX platform is able to process market activity on the scale of 100K messages/events per second. LMAX's internal latency is 80µs, yielding a nominal trade latency of 3ms. (user experience or wall time).

† Async logging, log4j2
<https://logging.apache.org/log4j/2.x/manual/async.html#UnderTheHood>

‡ Async logging is a related topic; however, it is not used in the demo app. If utilized by the demo app, async logging would need a Correlation ID (CID) to maintain a semblance of continuity.

The demo app could create such a CID by importing Java's UUID:

```
import java.util.UUID;
```

<https://mvnrepository.com/artifact/com.lmax/disruptor/3.2.1>

<https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html#method.summary>

§ Provides an adaptation of the Disruptor Pattern to broker an Event-driven Producer/Consumer app.

|| The interleaving behavior is "baked into" the polling logic/strategy of the [Exchanger](#). This strategy, as code in the demo app, is *Round-Robin* and is determined by modulo arithmetic; the divisor is based upon the number of *Producers*.

¶ The padding provision – innate to the Disruptor – prevents "corruption" of the contents of cache lines, which occurs when multiple (client) threads contend-for/share a part of a host machine's CPUs cache lines.

To compound matters, there are typically at least four (4) cores per CPU, and there may be multiple CPUs per machine.

When two or more variables share part of a cache line, and values of one or more of the other variables is invalidated, the entire cache line's contents **must** be invalidated too; this affects the cache lines in all CPUs, all of which must then be invalidated.

This antipattern is known as False Sharing.

The value of this meta-variable is architecture-dependent – so, it is important to choose a value the padding that is a multiple of the largest payload (Event) that the app (will allow, by convention) to be processed.

(continues)

(continued)

NB

While constructing an instance of the `com.lmax.disruptor.dsl.Disruptor.java` class, note that the implementation of the `ExecutorService` interface is deprecated in favor of an instance of a `ThreadFactory` instance; this is because a `ThreadFactory` reports when it's unable to construct a *Producer* Thread.

References

- <https://dzone.com/articles/ring-buffer-a-data-structure-behind-disruptor>
- https://trishagee.github.io/post/dissecting_the_disruptor_demystifying_memory_barriers/
- http://trishagee.github.io/post/dissecting_the_disruptor_writing_to_the_ring_buffer/
- <http://mechanitis.blogspot.com/2011/06/dissecting-disruptor-how-do-i-read-from.html>
- <https://lmax-exchange.github.io/disruptor/docs/com/lmax/disruptor/EventHandler.html#method.summary>