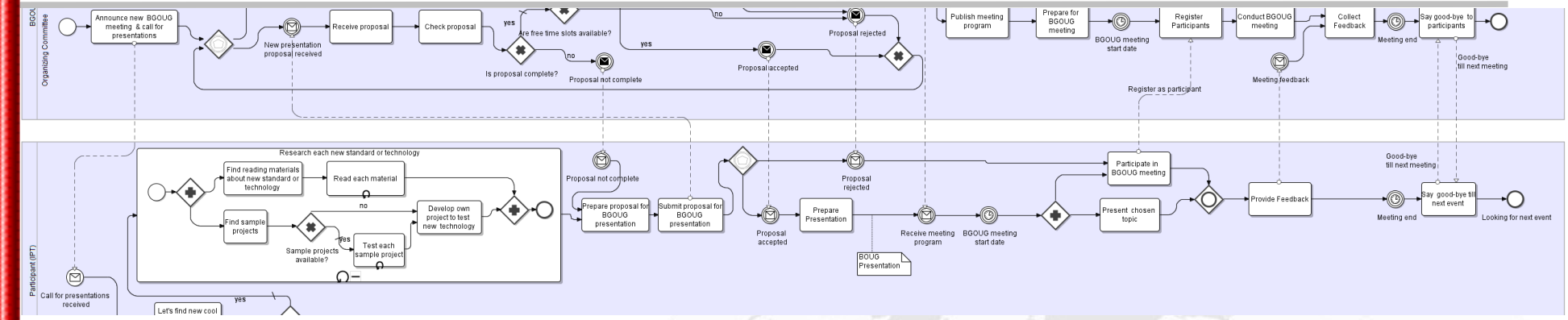
Java High Performance Reactive Programming



Trayan Iliev

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: http://iproduct.org

Agenda

1. Reactive programming. Reactive Streams Specification. Functional Reactive Programing.
2. Low latency and high throughput programming in Java.
3. Main performance factors – CPU architecture, memory hierarchies, lock contention, false sharing.
4. Single writer designs. The LMAX Disruptor (RingBuffer) high performance inter-thread messaging library.
5. Reactor & Proactor design patterns.
6. Building high-performance non-blocking asynchronous applications on the JVM using Reactor project.
7. RxJava –Java ReactiveX (Reactive Extensions)

Disclaimer

All information presented in this document and all supplementary materials and programming code represent only my personal opinion and current understanding and has not received any endorsement or approval by IPT - Intellectual Products and Technologies or any third party. It should not be taken as any kind of advice, and should not be used for making any kind of decisions with potential commercial impact. The information and code presented may be incorrect or incomplete. It is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. In no event shall the author or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the information, materials or code presented or the use or other dealings with this information or programming code.

About

Trayan Iliev

IPT – Intellectual Products & Technologies

IT Education Company specialized in Java™ and Java EE/Web and JS trainings

You are welcome!
http://iproduct.org/

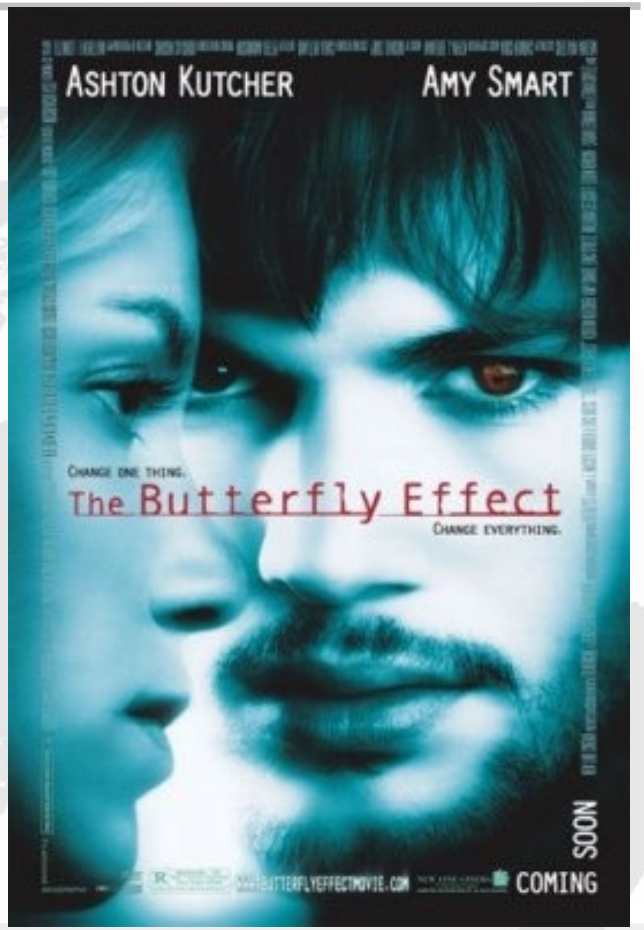
Being Reactive - What It Really Means?

My Favourite Definition of
Reactive Streams :)

https://youtu.be/qybUFnY7Y8w

We Live in a Connected Universe

The title refers to the butterfly effect, a popular hypothetical example of chaos theory which illustrates how small initial differences may activate chains of events leading to large and often unforeseen consequences in the future...



Source: https://en.wikipedia.org/wiki/The_Butterfly_Effect#/media/File:Butterflyeffect_poster.jpg
Fair use, File:Butterflyeffect poster.jpg, Uploaded by Yaminator, Uploaded: 8 July 2008

We Live in a Connected Universe

... there is hypothesis that all the things in the Universe are intimately connected, and you can not change a bit without changing all.

Action – Reaction principle is the essence of how Universe behaves.

Reactive Programming. Functional Programing

- **Reactive Programming** [Wikipedia]: a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow. Ex: **a := b + c**
- **Functional Programming** [Wikipedia]: a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm. Eliminating side effects can make it much easier to understand and predict the program behavior. Ex: **book -> book.getAuthor().fullName()**

Functional Reactive Programming: StackOverflow

*According to **Connal Elliot**'s answer in Stack Overflow (ground-breaking paper @ Conference on Functional Programming, **1997**):*

I'm glad you're starting by asking about a specification rather than implementation first. There are a lot of ideas floating around about what *FRP* is. For me it's always been two things: (a) ***denotative*** and (b) ***temporally continuous***. Many folks drop both of these properties and identify *FRP* with various implementation notions, all of which are beside the point in my perspective.

"***functional reactive programming***" = "***denotative, continuous-time programming***" (***DCTP***)

Denotative, Continuous-Time Programming according to Connal Elliot @StackOverflow

By **"denotative"**, I mean founded on a precise, simple, implementation-independent, ***compositional semantics*** that exactly specifies the meaning of each ***type and building block***. The compositional nature of the semantics then determines the meaning of all ***type-correct combinations*** of the building blocks.

About ***continuous time***, see the post *Why program with continuous time?*

- Using ***lazy functional languages***, we casually program with ***infinite data*** on ***finite machines***.
- I like my ***programs*** to reflect ***how I think about the problem space*** rather than ***the machine that executes the programs***, and I tend to expect other ***high-level language*** programmers to share that preference.

Functional Reactive Programming: @MEAP by Stephen Blackheath & Anthony Jones

Functional Reactive Programming

Stephen Blackheath and *Anthony Jones*

MEAP began November 2014 Publication in February 2016 (estimated)

ISBN 9781633430105

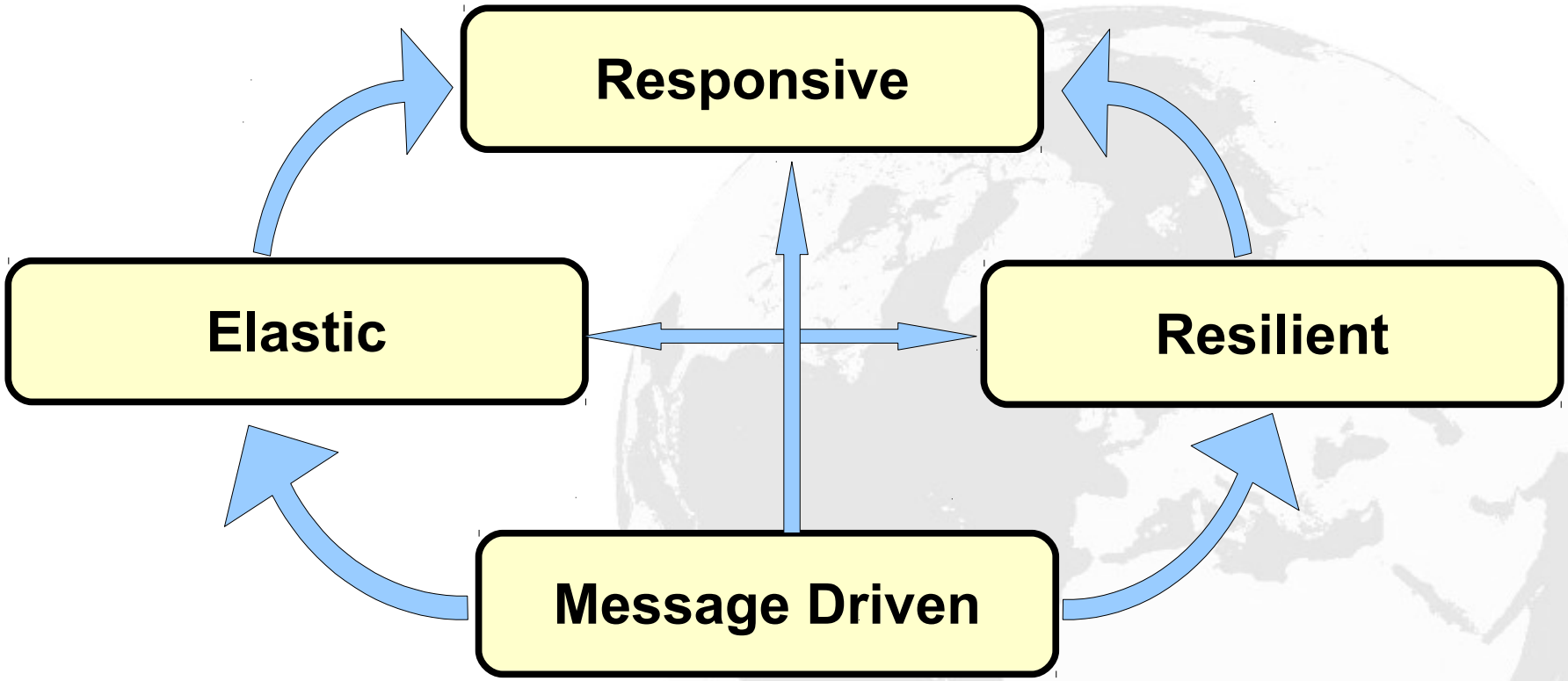
245 pages (estimated)

DEFINITION [Denotational semantics]

is a mathematical expression of the formal meaning of a programming language. For an FRP system, it provides both a formal specification of the system, and a proof that the important property of compositionality holds for all building blocks in all cases

https://github.com/SodiumFRP/sodium/

Reactive Manifesto
[http://www.reactivemanifesto.org/]



Other Definitions of Reactive Programming

- Microsoft® opens source polyglot project ***ReactiveX*** (Reactive Extensions) [http://reactivex.io]:

  Rx = Observables + LINQ + Schedulers :)

  - Supported Languages – Java: RxJava, JavaScript: RxJS, C#: Rx.NET, C#(Unity): UniRx, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Ruby: Rx.rb, Python: RxPY, Groovy: RxGroovy, JRuby: RxJRuby, Kotlin: RxKotlin, Swift: RxSwift
  - ReactiveX for platforms and frameworks: RxNetty, RxAndroid, RxCocoa
- ***Reactive Streams*** Specification [http://www.reactive-streams.org/] used by ***Project Reactor*** [http://projectreactor.io/, https://github.com/reactor/reactor]

Reactive Streams Specification
[http://www.reactive-streams.org/]

- Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM & JavaScript) as well as network protocols.
- The scope of Reactive Streams is to find a minimal set of interfaces, methods and protocols that will describe the necessary operations and entities to achieve the goal – asynchronous streams of data with non-blocking back pressure.
- As of April 30, 2015 have been released version 1.0.0 of **Reactive Streams for the JVM**, including Java API, a textual Specification, a **TCK and implementation examples**.

Reactive Streams Specification

[https://github.com/reactive-streams/reactive-streams-jvm]

- **Publisher** – provider of potentially unbounded number of sequenced elements, according to Subscriber(s) demand. After invoking Publisher.subscribe(Subscriber). Subscriber methods protocol is: onSubscribe onNext* (onError | onComplete)?
- **Subscriber** – receives call to onSubscribe(Subscription) once after passing an instance to Publisher.subscribe(Subscriber). No further notifications until Subscription.request(long) is called.
- **Subscription** – represents one-to-one lifecycle of a Subscriber subscribing to a Publisher. It is used to both signal desire for data and cancel demand (and allow resource cleanup).
- **Processor** -represents a processing stage, which is both a Subscriber and Publisher and obeys the contracts of both.

Reactive Programming = Programming with Asynchronous Data Streams

- **Functional Reactive Programming (FRP)** [Wikipedia]: a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter). FRP has been used for programming graphical user interfaces (GUIs), robotics, and music, aiming to simplify these problems by explicitly modeling time. Example (RxJava):

```
Observable.from(new String[]{"Reactive", "Extensions", "Java"})
    .take(2).map(s -> s + " : on " + new Date())
    .subscribe(s -> System.out.println(s));
```

```
Result: Reactive : on Wed Jun 17 21:54:02 GMT+02:00 2015
        Extensions : on Wed Jun 17 21:54:02 GMT+02:00 2015
```

Good intro tutorial in RP using RxJS by Andre Staltz see: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
JS Fiddle of the demo: http://jsfiddle.net/staltz/8jFJH/48/

What Is Low Latency? - Martin Thompson: http://www.infoq.com/articles/low-latency-vp

- **Performance** is about 2 things - **throughput**, e.g. units per second, and response time otherwise know as **latency**. It is important to define the units and not just say something should be "fast". Real-time has a very specific definition and is often misused. Real-time is to do with systems that have a real time constraint from input event to response time regardless of system load. In a ***hard real-time system*** if this constraint is not honored then a ***total system failure can occur***. Good examples are heart pacemakers or missile control systems.

(- continued on next slide -)

What Is Low Latency? - Martin Thompson :

- … With trading systems, real-time tends to have a different meaning in that the system must have high throughput and react as quickly as possible to an event, which can be considered ***"low latency"***. Missing a trading opportunity is typically not a total system failure so you cannot really call this real-time.

  A good trading system will have a high quality of execution for which one aspect is to have a ***low latency response with little deviation in response time***.

  [Martin Thompson]

What Is Low Latency? - Answers by Experts: http://www.infoq.com/articles/low-latency-vp

- A system with a measured latency requirement which is too fast to see. This could be anywhere from 100 nano-seconds to 100 milli-seconds. [Peter Lawrey]
- Real-time and low latency can be quite different. The majority view on "real-time" would be determinism over pure speed with very closely controlled, or even bounded, outliers. However, "low latency" typically implies that pure speed is given much higher priority and some outliers may be, however slightly, more tolerable. This is certainly the case when thinking about hard real-time. One of the key pre-requisites for low latency is a keen eye for efficiency. From a system view, this efficiency

(- continued on next slide -)

What Is Low Latency? - Todd L. Montgomery :

- ... must permeate the entire application stack, the OS, and the network. This means that low latency systems have to have a high degree of mechanical sympathy to all those components. In addition, many of the techniques that have emerged in low latency systems over the last several years have come from high performance techniques in OSs, languages, VMs, protocols, other system development areas, and even hardware design.

[Todd L. Montgomery]

What Is Low Latency? - Andy Piper :

- Latency is simply the **delay between decision and action**. In the context of high performance computing, low latency has typically meant that transmission delays across a network are low or that the overall delays from request to response are low. What defines "low" depends on the context – low latency over the Internet might be **200ms** whereas low latency in a trading application might by **2µs**. Technically low latency is not the same as real-time – low latency typically is measured as percentiles where the outliers (situations in which latency has not been low) are extremely important to know about. With real-time, guarantees are made about the behavior of the system –

(- continued on next slide -)

What Is Low Latency? - Andy Piper :

- … so instead of measuring percentile delays you are enforcing a maximum delay. You can see how a real-time system is also likely to be a low latency system, whereas the converse is not necessarily true. Today however, the notion of enforcement is gradually being lost so that many people now use the terms interchangeably.

  If latency is the overall delay from request to response then it is obvious that many things contribute to this delay – CPU, network, OS, application, even the laws of physics! Thus low latency systems typically require **high-performance code** so that **software elements of latency can be reduced**.

  [Dr. Andy Piper]

Low Latency Software System

- **Low latency software system** is a system in which the hardware (CPU, cache, memory, IO, Network), low-level operating system, language specific implementation platform (e.g. JVM), and application level code are working in harmony to **minimize the time** needed for event (request, message) processing. This synergistic property of different system layers is sometimes called **Mechanical Sympathy**.
- In order to achieve low latency we have to minimize the time one component is waiting unnecessary for another component to finish, shortening the **critical path**.
- This is achieved by making informed decisions during phases of system design, implementation, configuration and testing.

What Is Throughput?

- Number of events (requests, messages, bytes) that are processed by the system per second.
- Does high throughput imply low latency?
- Not necessarily – e.g. bus vs. car traveling:
  - Which has the higher throughput?
  - Which has the lower latency?
- **Throughput** ~ **System Capacity** / **Latency**
- System Capacity = Number of units processed in parallel
- Achieving **Low Latency** may mean additional work done by system => lowered **System Capacity** and **Throughput**

According to Peter Lawrey:

- Critical operations can be modeled as a series of asynchronous events, which can be recorded, knowing critical system state
- Horizontal scalability is valuable for high throughput. For low latency, you need simplicity - the less to do the less time it takes
- The key driver for low latency is **how easy is it to take out redundant operations from the critical path**.
- You have to understand how all different layers interact for the critical code and often combine layers to simplify the task.
- You can use natural Java for non critical code - the majority. For critical sections you need a subset of Java and **libraires** which are suitable for low latency. => 10% / 90% principle

InfoQ: How Well Is Java Suited for Low Latency? http://www.infoq.com/articles/low-latency-vp

- **Lawrey:** If your application spends 90% of the time in 10% of your code, Java makes optimising that 10% harder, but writing and maintaining 90% of your code easier; especially for teams of mixed ability.
- **Montgomery:** ... I think currently, the difference in performance between Java and C++ is so close that it's not a black and white decision based solely on speed. Improvements in GC techniques, JIT optimizations, and managed runtimes have made traditional Java weaknesses with respect to performance into some very compelling strengths that are not easy to ignore.

InfoQ: How Well Is Java Suited for Low Latency? http://www.infoq.com/articles/low-latency-vp

- **Thompson:** Low latency systems written in Java tend to not use 3rd party or even standard libraries for two major reasons. Firstly, many libraries have not been written with performance in mind and often do not have sufficient throughput or response time. Secondly, they tend to **use locks when concurrent**, and they **generate a lot of garbage**. Both of these contribute to highly variable response times, due to **lock-contention** and **garbage collection** respectively.

  Java has some of the **best tooling** support of any language which results in significant productivity gains. Time to market is often a key requirement when building trading systems, and Java can often get you there sooner.

InfoQ: How Well Is Java Suited for Low Latency? http://www.infoq.com/articles/low-latency-vp

- **Piper:** … writing good low latency code in Java is relatively hard since the developer is insulated from the guarantees of the hardware by the JVM itself. The good news is that this is changing, not only are JVMs constantly getting faster and more predictable but developers are now able to take advantage of hardware guarantees through a detailed understanding of the way that Java works – in particular the **Java Memory Model** - and how it maps to the underlying hardware ... A good example is the **lock-free**, **wait-free** techniques ... as these techniques become more mainstream we are starting to see their uptake in standard libraries (e.g. the **Disruptor**) so that developers can adopt the techniques without needing such a detailed understanding of the underlying behaviour.

InfoQ: How Well Is Java Suited for Low Latency? http://www.infoq.com/articles/low-latency-vp

- **Piper** (- continued -): Even without these techniques the safety advantages of Java (memory management, thread management etc.) can often outweigh the perceived performance advantages of C++, and of course JVM vendors have claimed for some time that modern **JVMs are often faster than custom C++ code** because of the **holistic optimizations** that they can apply across an application.

Low Latency: Things to Remember

- **Low garbage** through reusing existing objects - infrequent GC ideally when application not busy – can improve app 2 - 5x
  - In JVM using generational GC startegy it is ideal objects either to live very shortly (to be garbage collected in the next minor sweep) or be immortal (reused forever).
- **Non-blocking, lockless coding** – if there should be mutual exclusion it is preferred to use Compre-And-Swap – CAS (java.util.concurrent.atomic), than locks.
- For critical data structures – **direct memory access** using DirectByteBuffers or Unsafe => predictable memory layout and cache misses avoidance.

Low Latency: Things to Remember - II

- **Busy waiting** isolated critical threads – giving the CPU to the OS kernel slows your program by 2-5x (according to Lawrey) => avoid voluntary context switches.
- Write an algorithm that can **amortize the effect of expensive operations like IO** (including network) and **cache misses** – every IO operation that can block eventually will block – don't be surprised.
- Avoid **"false sharing"** - the situation that arises when two independent variables written concurrently by two different threads (and employing memory barrier e.g. using **volatile**) **share the same cache-line** (32-256 bytes, typically 64 bytes) if contention is to be minimized.

Parallelism & Concurrency

- Doing several tasks in **parallel** can increase your **Throughput** by increasing **System Capacity –** it is GOOD!
- But it usually comes hand-in-hand with **concurrent** access to shared resources => you have to provide **mutual exclusion (MutEx)** by parallel threads when changing the state of those resources (read only access can be shared by multiple threads)

Mutual exclusion can be achieved in several ways:

- **synchronized** – hardwired in HotSpot JVM, well optimized in J6
- **ReentrantLock, ReadWriteLock, StampedLock** → java.util.concurrent.locks.*
- **Optimistic Locking - tryLock()**, Using **Compare-And-Swap (CAS)** instructions → java.util.concurrent.atomic.*

Comparing Different Concurrent Implementations

- Simple problem: incrementing a long value 500 000 000 times.

9 implementations:

- SynchronousCounter – `while (counter++ < 500000000){}`
- SingleThreadSynchronizedCounter – 1T using **synchronized**
- TwoThreadsSynchronizedCounter – 2T using **synchronized**
- SingleThreadCASCounter – 1T using **`AtomicLong`**
- TwoThreadsCASCounter – 2T using **`AtomicLong`**
- TwoThreadsCASCounterLongAdder – 1T using **`LongAdder`**
- SingleThreadVolatileCounter – 1T, **memory barrier (volatile)**
- TwoThreadsVolatileCounter – 2T, **memory barrier (volatile)**

Comparing Different Concurrent Implementations

Test results (on my laptop - quad core Intel i7@2.2GHz):

- SynchronousCounter – 190ms
- SingleThreadSynchronizedCounter – 15000 ms
- TwoThreadsSynchronizedCounter – 21000 ms
- SingleThreadCASCounter – 4100 ms
- TwoThreadsCASCounter – 12000 ms
- TwoThreadsCASCounterLongAdder – 12800 ms
- SingleThreadVolatileCounter – 4100 ms
- TwoThreadsVolatileCounter – 20000 ms

Comparing Different Concurrent Implementations

For more complete microbenchmarking of different Mutex implementations see:

- http://blog.takipi.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized/
- http://www.slideshare.net/haimyadid/java-8-stamped-lock

Mutex Comparison => Conclusions

- **Non-blocking (synchronous)** implementation is **2 orders of magnitude** better then **synchronized**
- We should **try to avoid blocking and especially contended blocking** if want to achieve low latency
- If blocking is a must we have to **prefer CAS and optimistic concurrency** over blocking (but have in mind it always depends on concurrent problem at hand and how much contention do we experience – test early, test often, microbenchmarks are unreliable and highly platform dependent – **test real application with typical load patterns**)
- The real question is: Is it possible to build concurrency **without blocking?**

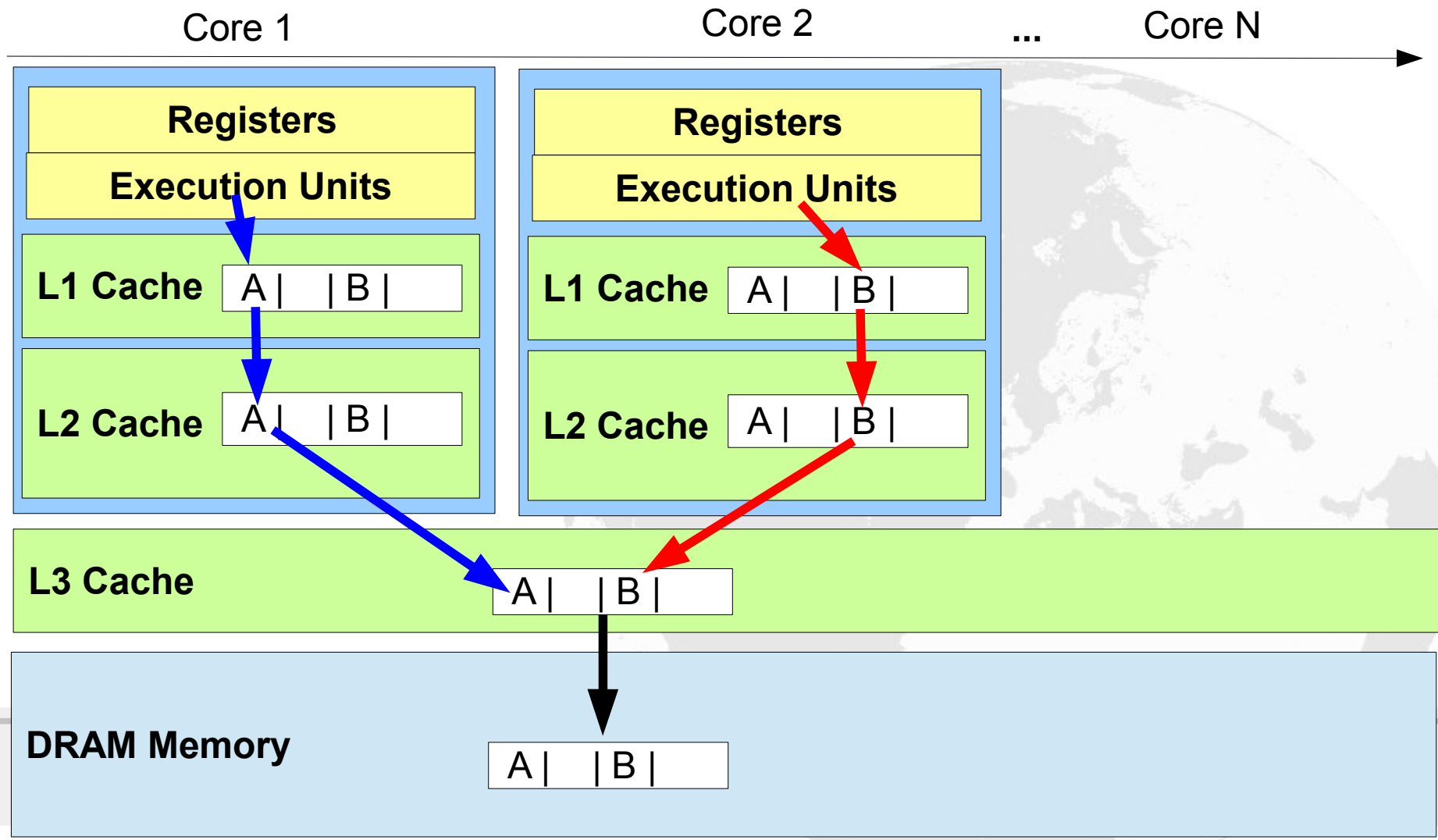
Main Factors Affecting Performance - I

- CPU architecture – multicore, hyperthreading
- Memory hierarchies – caches, predictive caching, memory access patterns: temporal, spatial, striding
- Garbage Collection – serial, parallel and concurrent GC strategies, genrational GC
- Lock contention –
  - uncontended locks - JVM executes fast path code to acquire and release thin lock ownership. The fast path typically involves one or two compare-exchange instructions (lock;cmpxchg on x86 and cas on SPARC) ~ 10 – 20 CPU cycles after Nehalem architecture
  - contended locks - much more expensive because the thread has to be parked and the control has to be released to OS kernel

Main Factors Affecting Performance - II

- False sharing – when two independent variables written concurrently by two different threads share the same cache-line (typically 64 bytes) => has the same effect as two threads contending on single variable

CPU Cache Architecture – False Sharing



Building Scalable, Massively Concurrent Computation Architectures

- Message Driven – asynchronous message-passing allows to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages [Reactive Manifesto].
- The main idea is to separate concurrent producer and consumer workers by using message queues.
- Message queues can be unbounded or bounded (limited max number of messages)
- Unbounded message queues can present memory allocation problem in case the producers outrun the consumers for a long period → OutOfMemoryError

Message Driven Designs

Message-driven architecture has been employed in a number of designs at different levels of the system stack – examples:

- Field-programmable gate arrays (FPGA) – different concurrently working MicroBlaze™ cores (Xilinx®) separated by bounded queues passing data between cores.
- Message-Oriented Middleware (MOM) – e.g. Java Message Service (JMS) & Message Driven Beans (MDB) as part of JavaEE specification.
- Service Oriented Architecture (SOA) – SOAP (XML) and REST (JSON, BSON, XML, etc.)
- Actor Model (http://dspace.mit.edu/handle/1721.1/6952), Akka
- Microservices (or even Nano Services, are there Piko? :)

Disadvantages of Traditional Queues [http://lmax-exchange.github.com/disruptor/files/Disruptor-1.0.pdf]

- Queues typically use either linked-lists or arrays for the underlying storage of elements. Linked lists are not „mechanically sympathetic" – there is no predictable **caching "stride"** (should be less than 2048 bytes in each direction).
- Bounded queues often experience write **contention** on head, tail, and size variables. Even if head and tail separated using CAS, they usually are in the **same cache-line**.
- Queues produce **much garbage**.
- Typical queues **conflate a number of different concerns** – e.g. producer and consumer synchronization and data storage

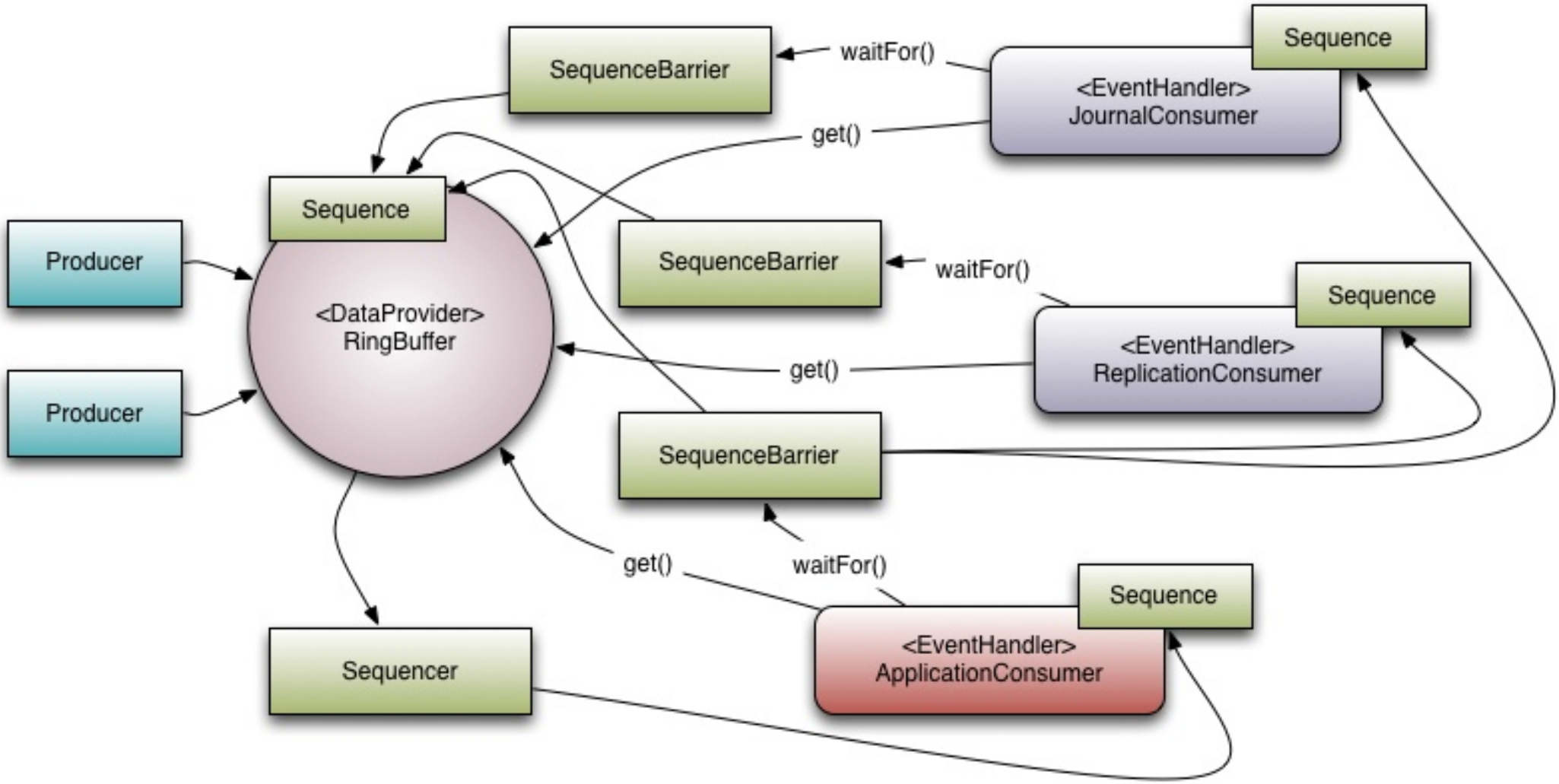
Single Writer Designs. LMAX Disruptor (RingBuffer) High Performance Inter-Thread Messaging Library

[http://lmax-exchange.github.com/disruptor/files/Disruptor-1.0.pdf]

- Instead of conflation synchronization and storage the **LMAX Disruptor** design pattern separates different concerns in a "mechanically sympathetic" way:
  - Storage of items being exchanged
  - Coordination of producers claiming the next sequence for exchange
  - Coordination of consumers being notified that a new item is available
- **Single Writer** principle is employed when writing data in the Ring Buffer from single producer thread only (no contention)

LMAX Disruptor (RingBuffer) High Performance [http://lmax-exchange.github.com/disruptor/files/Disruptor-1.0.pdf]

- In more exotic case of multiple producers which race with each other to claim the next entry in the ring-buffer, simple CAS operation on the sequence number for that slot is employed.
- Consumers wait for a sequence to become available in the ring buffer before they read the entry using loop checking with or without yielding (trading CPU for latency), or if CPU resource precious can wait on condition within a producer signaled lock.
- All the memory for storing entries is **pre-allocated on starup** as a cache friendly array with predictable stride and there is effectively **no garbage produced** (slots are reused) during the operation. The size should be power of 2 (faster reminder calc)



Source: LMAX Disruptor github wiki - https://raw.githubusercontent.com/wiki/LMAX-Exchange/disruptor/images/Models.png

LMAX Disruptor DSL & API
[https://github.com/LMAX-Exchange/disruptor/wiki/Introduction]

- **Ring Buffer** – in Disruptor 3.0 the Ring Buffer is only responsible for the storing and updating the data (Events)
- **Sequence** – used to identify where a particular component is up to in the buffer. Each consumer (EventProcessor) maintains a Sequence as does the Disruptor itself. Sequences supports features of an AtomicLong + false sharing prevention.
- **Sequencer** – real Disruptor core: 2 implementations (single producer, multi-producer) implementing concurrent algorithms for fast passing of data between producers and consumers.
- **Sequence Barrier** – produced by the Sequencer, contains references to the published Sequences from the Sequencer and dependent consumers, determines if Events to consume.

LMAX Disruptor DSL & API
[https://github.com/LMAX-Exchange/disruptor/wiki/Introduction]

- **Wait Strategy** – the Wait Strategy determines how a consumer will wait for events to be placed into the Disruptor by a producer; optionally lock-free.
- Event – user defined data unit passed from producer to consumer.
- **EventProcessor** – main event loop handling Disruptor events; owner of consumer's Sequence => **BatchEventProcessor**
- **EventHandler** – an interface that is implemented by the user and represents a consumer for the Disruptor.
- Producer – the user defined code that calls the Disruptor to enqueue Events.

LMAX Disruptor Example – Single Producer & Single Consumer - I

```
public final class ValueEvent {
    private long value;
    public long getValue(){
        return value;
    }
    public void setValue(final long value){
        this.value = value;
    }
    public final static EventFactory<ValueEvent> EVENT_FACTORY =
        new EventFactory<ValueEvent>(){
            public ValueEvent newInstance(){
                return new ValueEvent();
            }
        };
}
```

LMAX Disruptor Example – Single Producer & Single Consumer - II

```
public class Demo1P1C {
   public static final int RING_SIZE = 128;
   public static final int SAMPLES_SIZE = 500000;
   public static final int NUMBER_CONSUMERS = 1;
   public static long start, end;
   public static final ExecutorService EXECUTOR =
      Executors.newFixedThreadPool(NUMBER_CONSUMERS);

   public static void main(String[] args) {
      final EventHandler<ValueEvent> handler =
      new EventHandler<ValueEvent>() {
         private BitSet bset = new BitSet(SAMPLES_SIZE);
         public void onEvent(final ValueEvent event, final long
         sequence, final boolean endOfBatch) throws Exception {
                              ( - continues -)
```


LMAX Disruptor Example – Single Producer & Single Consumer - III

```
            bset.set((int)(event.getValue()));
            if(event.getValue() == SAMPLES_SIZE - 1) {
               end = System.nanoTime();
               System.out.println("Number samples received: " +
                  bset.cardinality());
               System.out.println((end - start) / 1000000d + "ms");
            }
         }
      };

      // Create single producer
      RingBuffer<ValueEvent> ringBuffer =
         RingBuffer.createSingleProducer(ValueEvent.EVENT_FACTORY,
            RING_SIZE, new SleepingWaitStrategy());
                              ( - continues -)
```

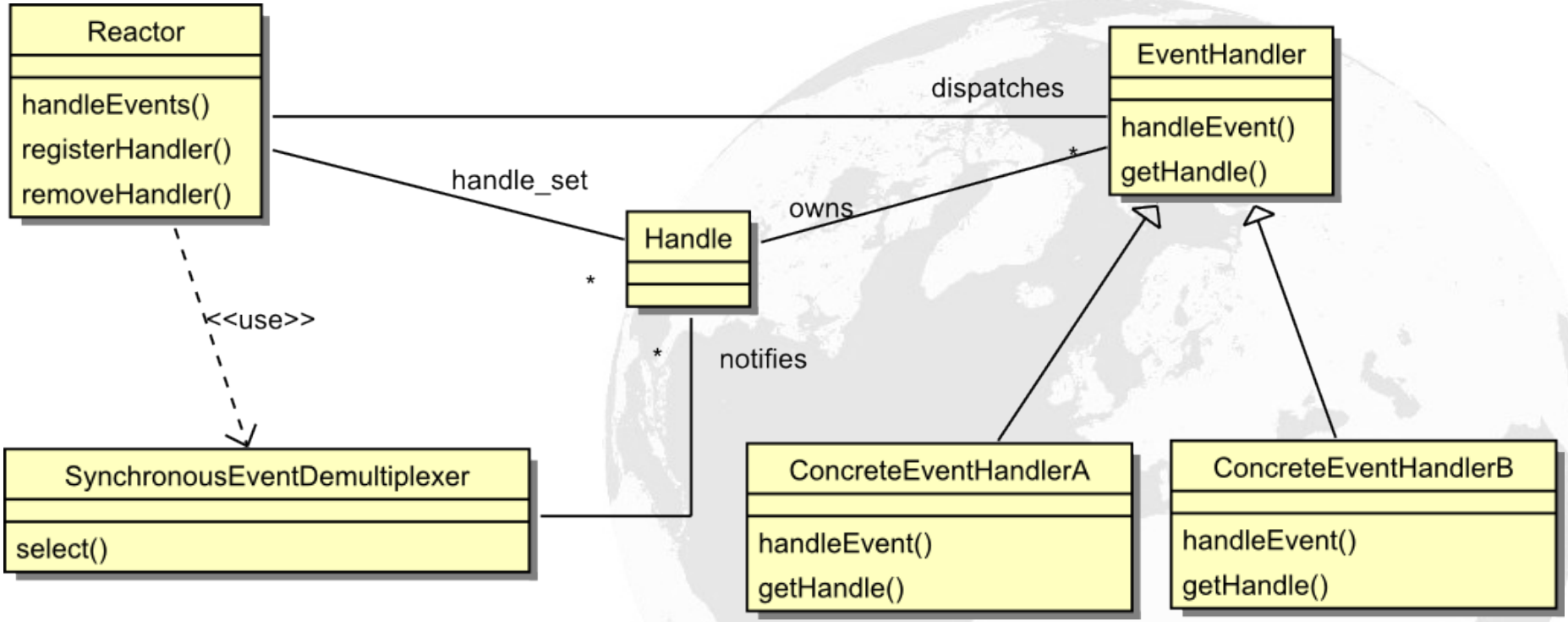

LMAX Disruptor Example – Single Producer & Single Consumer - IV

```
        SequenceBarrier barrier = ringBuffer.newBarrier();
        BatchEventProcessor<ValueEvent> eventProcessor =
           new  BatchEventProcessor<ValueEvent>(
              ringBuffer, barrier, handler);
      ringBuffer.addGatingSequences(eventProcessor.getSequence());
        EXECUTOR.submit(eventProcessor); //run on separate thread
        start = System.nanoTime();
        for(int i = 0;  i  < SAMPLES_SIZE; i++) {
            long sequence = ringBuffer.next();// claim event
        ValueEvent event = ringBuffer.get(sequence);
        event.setValue(i);
        ringBuffer.publish(sequence); //supply event
    }
}}
```

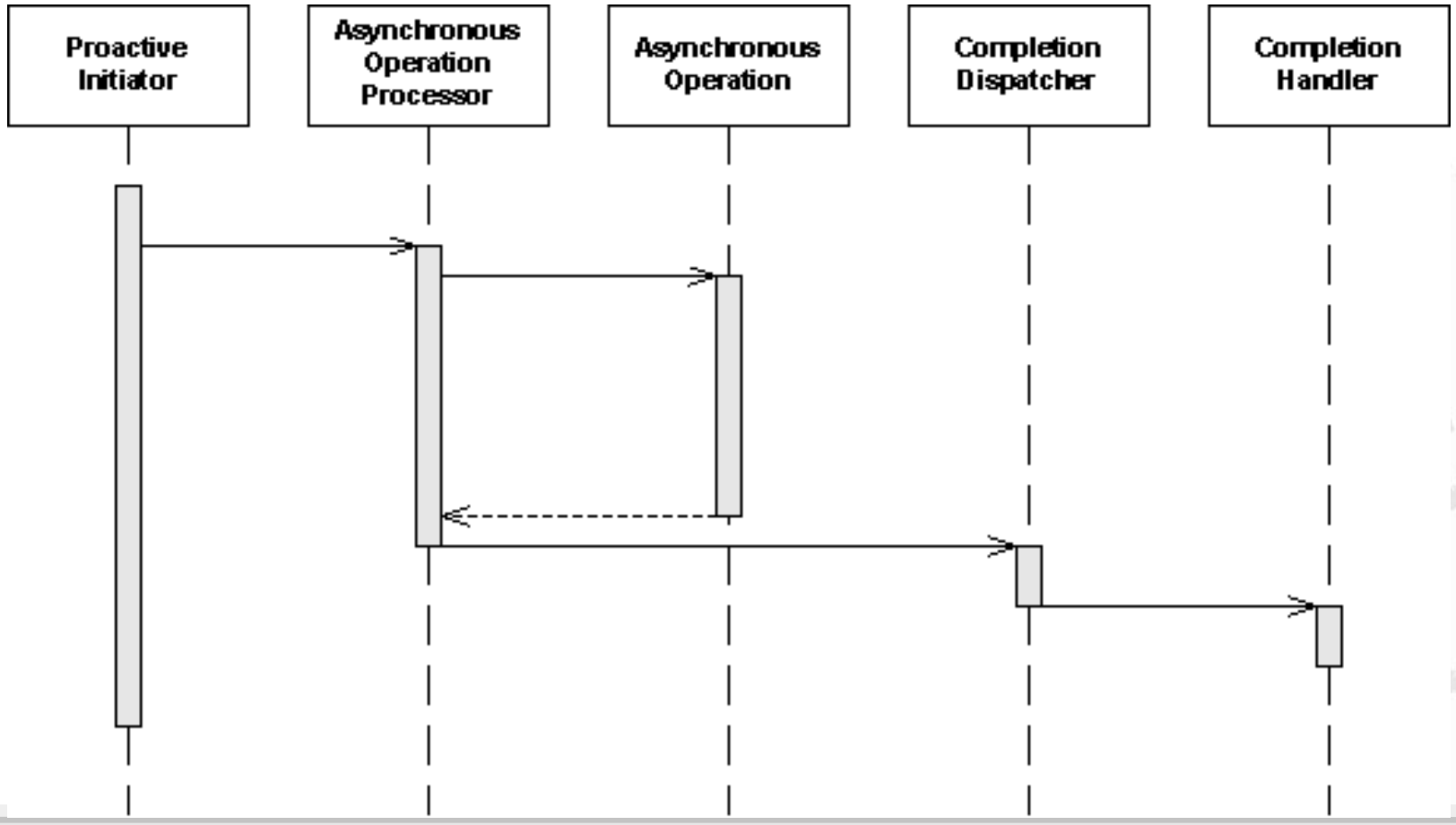
Reactor & Proactor Design Patterns

- The **Reactor** design pattern is an event handling pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers. [Wikipedia]
- **Proactor** is a software design pattern for event handling in which long running activities are running in an asynchronous part. A completion handler is called after the asynchronous part has terminated. The proactor pattern can be considered to be an asynchronous variant of the synchronous reactor pattern. [Wikipedia]

Reactor Design Pattern



Proactor Design Pattern



Source: Wikipedia
https://upload.wikimedia.org/wikipedia/commons/6/61/Proactor.VSD_SequenceDiagram.png
Author: UlrichAAB, License: Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)

Project Reactor

[http://projectreactor.io/, https://github.com/reactor/reactor]

- Reactor project allows building high-performance (low latency and high throughput) non-blocking asynchronous applications on the JVM.
- Reactor is designed to be extraordinarily **fast** and can sustain throughput rates on the order of 10's of millions of operations per second.
- Reactor has **powerful API** for declaring data transformations and functional composition.
- Makes use of the concept of **Mechanical Sympathy** by building on top of the Disruptor RingBuffer.
- **Fully Reactive** — Reactor is designed to be functional and reactive to allow for easy composition of operations.

Reactor Main Advantages
(According to http://projectreactor.io/)

- Pre-allocation at startup-time;
- Message-passing structures are bounded;
- Using Reactive and Event-Driven Architecture patterns => non-blocking end-to-end flows, including replies;
- Implement Reactive Streams Specification, to make bounded structures efficient by not requesting more than their capacity;
- Applies above features to IPC and provides non-blocking IO drivers that are flow-control aware;
- Expose a Functional API to help developers organize their code in a side-effect free way, which helps you determine you are thread-safe and fault-tolerant.

Project Reactor – HelloWorld Example

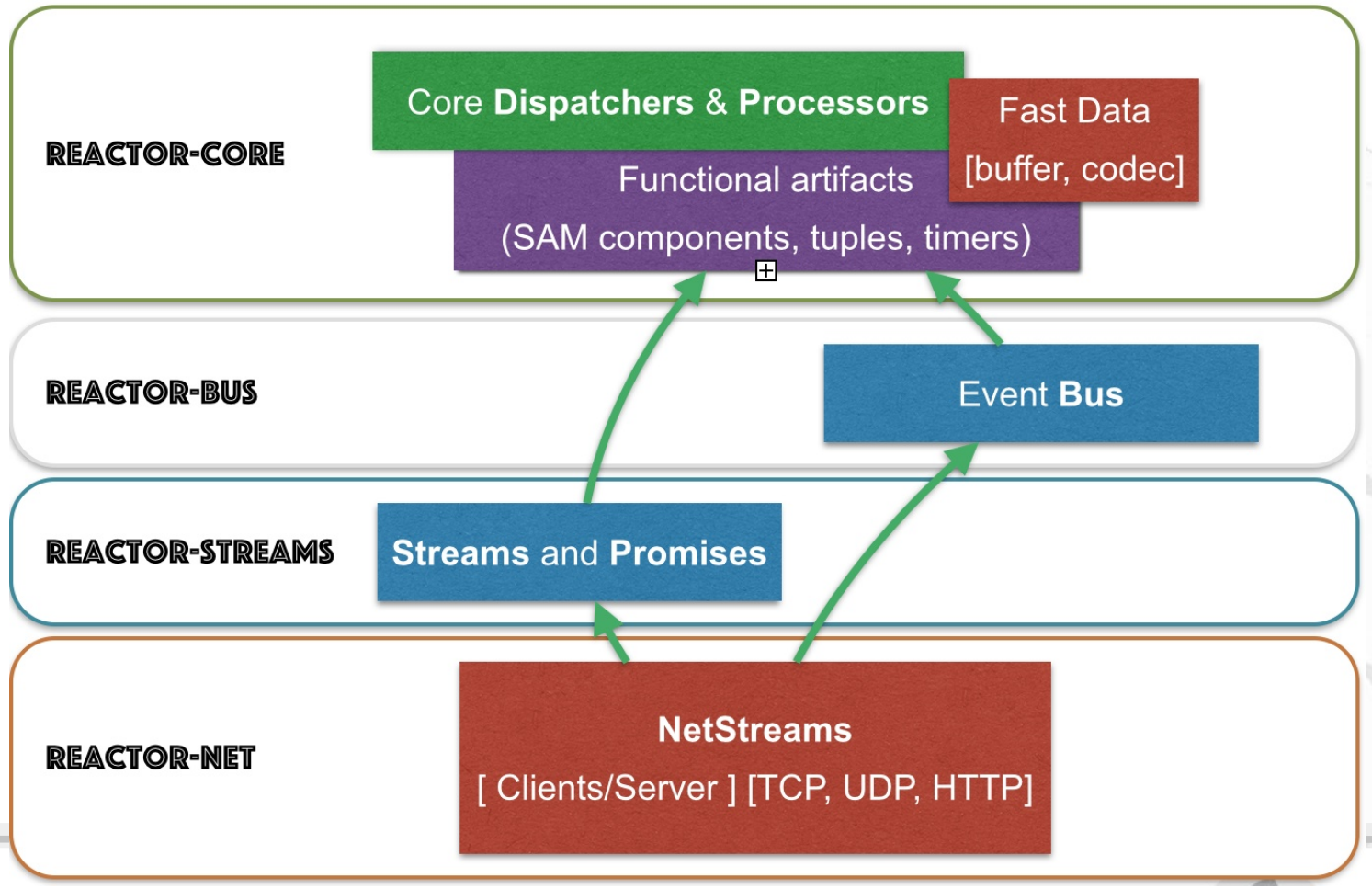
[http://projectreactor.io/, https://github.com/reactor/reactor]

```
import reactor.Environment;
import reactor.rx.broadcast.Broadcaster;

public class ReactorHelloWorld {
  public static void main(String... args) throws
                                          InterruptedException {
    Environment.initialize();
    Broadcaster<String> sink =
          Broadcaster.create(Environment.get());
    sink.dispatchOn(Environment.cachedDispatcher())
        .map(String::toUpperCase)
        .filter(s -> s.startsWith("HELLO"))
        .consume(s -> System.out.printf("s=%s%n", s));
    sink.onNext("Hello World!"); sink.onNext("Goodbye World!");
    Thread.sleep(500);
  }
}
```

Project Reactor Basic Architecture

[http://projectreactor.io/, https://github.com/reactor/reactor]



Source: Project Reactor - http://projectreactor.io/docs/reference/#_about_the_project

RxJava – Java ReactiveX (Reactive Extensions)

[http://reactivex.io, https://github.com/ReactiveX/RxJava/]

- **ReactiveX** is a polyglot library for composing asynchronous and event-based programs by using observable sequences.
- It extends the **observer pattern** to support sequences of data and/or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.
- Allow composing flows and sequences of asynchronous data.
- Observables can be implemented using thread-pools, event loops, non-blocking I/O, actors (such as Akka). Client code treats all of its interactions with Observables as asynchronous, whether your underlying implementation is blocking or non.

RxJava – Hello World Example (Java 8)

[http://reactivex.io, https://github.com/ReactiveX/RxJava/]

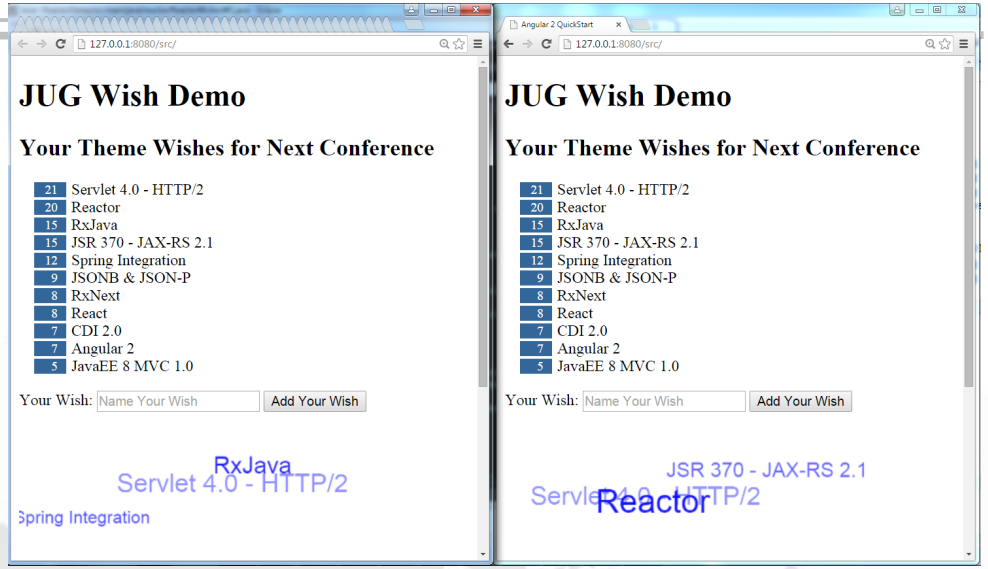
```
import java.util.Date;
import rx.Observable;

public class HelloRxJava2 {
    public static void helloLambda(String... names) {
        Observable.from(names)
            .take(2).map(s -> s + " : on " + new Date())
            .subscribe(System.out::println);
    }
    public static void main(String[] args) {
      helloLambda("Reactive", "Extensions", "Java");
    }
}
```

rx	reactor-stream	Comment
Observable	reactor.rx.Stream	Reflect the implementation of the Reactive Stream Publisher
Operator	reactor.rx.action.Action	Reflect the implementation of the Reactive Stream Processor
Observable with 1 data at most	reactor.rx.Promise	Type a unique result, reflect the implementation of the Reactive Stream Processor and provides for optional asynchronous dispatching.
Factory API (just, from, merge….)	reactor.rx.Streams	Aligned with a core data-focused subset, return Stream
Functional API (map, filter, take….)	reactor.rx.Stream	Aligned with a core data-focused subset, return Stream
Schedulers	reactor.core.Dispatcher, org.reactivestreams.Processor	Reactor Streams compute operations with unbounded shared Dispatchers or bounded Processors
Observable.observeOn()	Stream.dispatchOn()	Just an adapted naming for the dispatcher argument

Source: Project Reactor - http://projectreactor.io/docs/reference/#_about_the_project

IPT JUG Wish Demo - Lets See Some Code :)



Code available

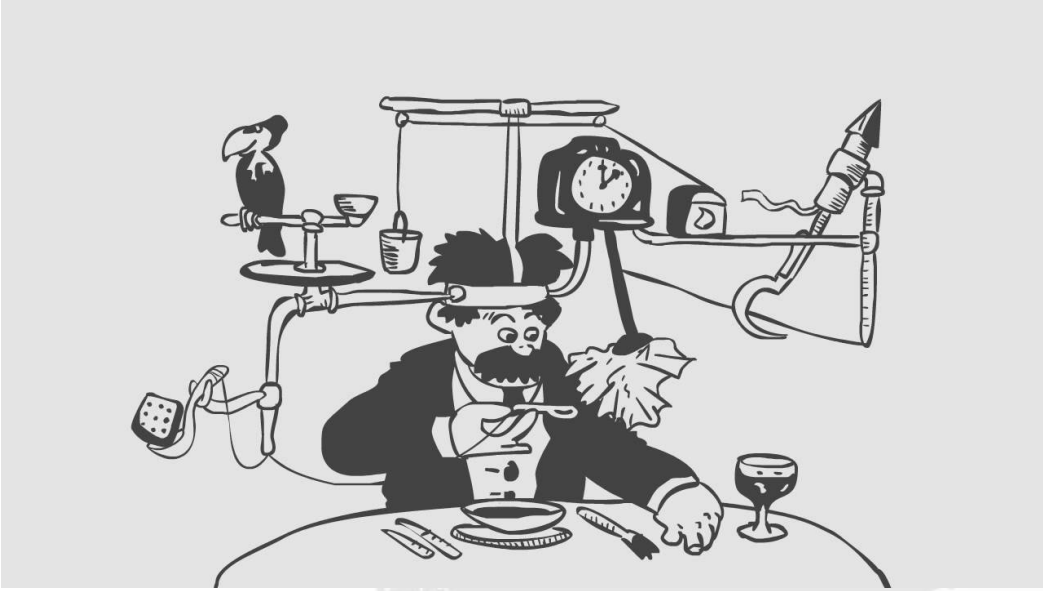
@ GitHub:

https://github.com/iproduct/ipt-angular2-reactive-websocket-demo

WebSocket endpoint java class: reactor.ReactorWishesWS

WebSocket web client: src/main/webapp (Angular2 + RxJS)

Want to Learn More: Welcome to IPT Reactive Programming Workshop



High Performance Reactive Programming with Java™ 8 and JavaScript – February 6, 2016

http://iproduct.org/en/course-reactive-java-js/

Thanks for Your Attention!

Questions?

