June 2, 2017
IPT – Intellectual
Products & Technologies

Asynchronous Data Stream Processing Using CompletableFuture and Flow in Java 9

Trayan Iliev

tiliev@iproduct.org
http://iproduct.org

Trademarks

Oracle®, Java™ and JavaScript™ are trademarks or registered trademarks of Oracle and/or its affiliates.

Other names may be trademarks of their respective owners.

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.

IPT - Intellectual Products & Technologies

Since 2003 we provide trainings and share skills in JS/ TypeScript/ Node/ Express/ Socket.IO/ NoSQL/ Angular/ React / Java SE/ EE/ Web/ REST SOA:

- Node.js + Express/ hapi + React.js + Redux + GraphQL
- Angular + TypeScript + Redux (ngrx)
- Java EE6/7, Spring, JSF, Portals: Liferay, GateIn
- Reactive IoT with Reactor / RxJava / RxJS
- SOA & Distributed Hypermedia APIs (REST)
- Domain Driven Design & Reactive Microservices

Stream Processing with JAVA 9

- Stream based data / event / message processing for real-time distributed SOA / microservice / database architectures.
- PUSH (hot) and PULL (cold) event streams in Java
- CompletableFuture & CompletionStage non-blocking, asynchronous hot event stream composition
- Reactive programming. Design patterns. Reactive Streams (java.util.concurrent.Flow)
- Novelties in Java 9 CompletableFuture
- Examples for (reactive) hot event streams processing

Where to Find the Demo Code?

CompletableFuture and Flow demos are available @ GitHub:

https://github.com/iproduct/reactive-demos-java-9

Data / Event / Message Streams

“Conceptually, a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.”

Apache Flink: Dataflow Programming Model

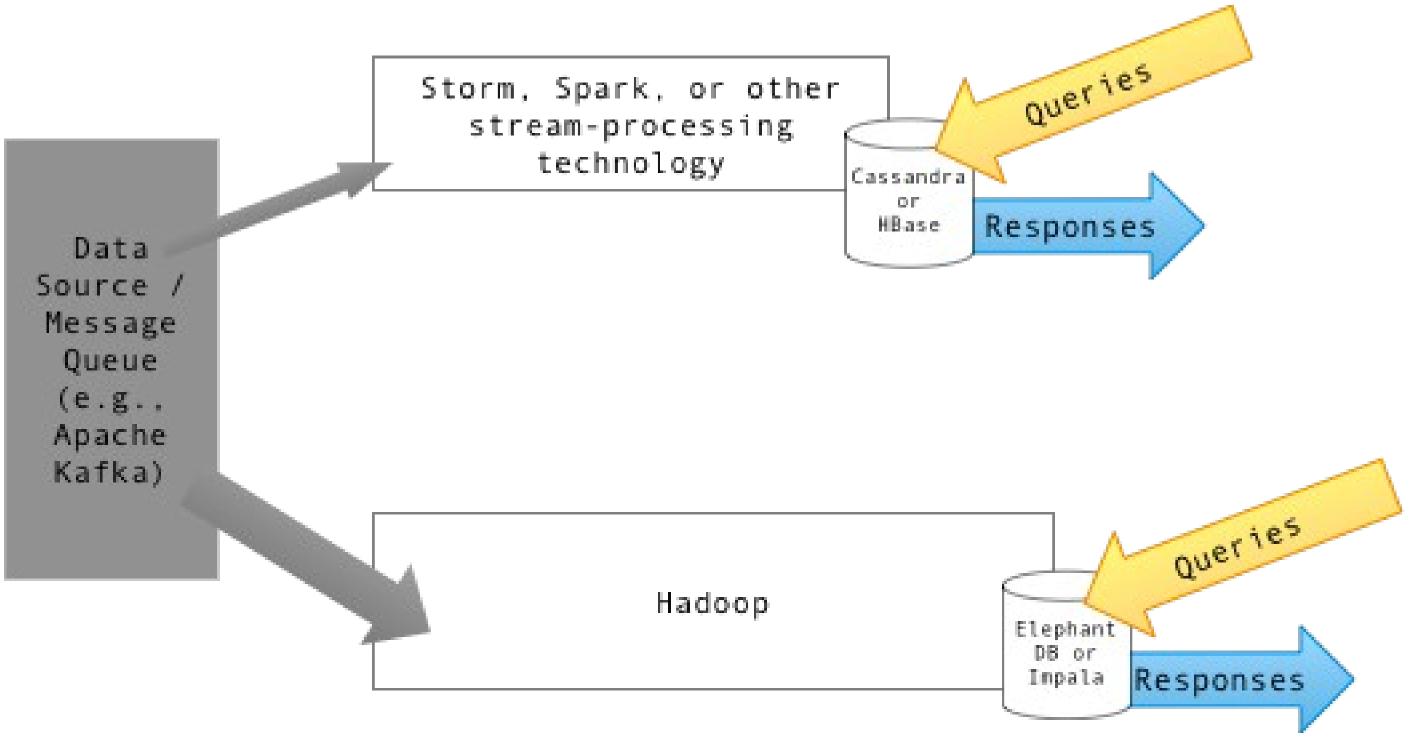
Data Stream Programming

The idea of abstracting logic from execution is hardly new -- it was the dream of SOA. And the recent emergence of microservices and containers shows that the dream still lives on.

For developers, the question is whether they want to learn yet one more layer of abstraction to their coding. On one hand, there's the elusive promise of a common API to streaming engines that in theory should let you mix and match, or swap in and swap out.

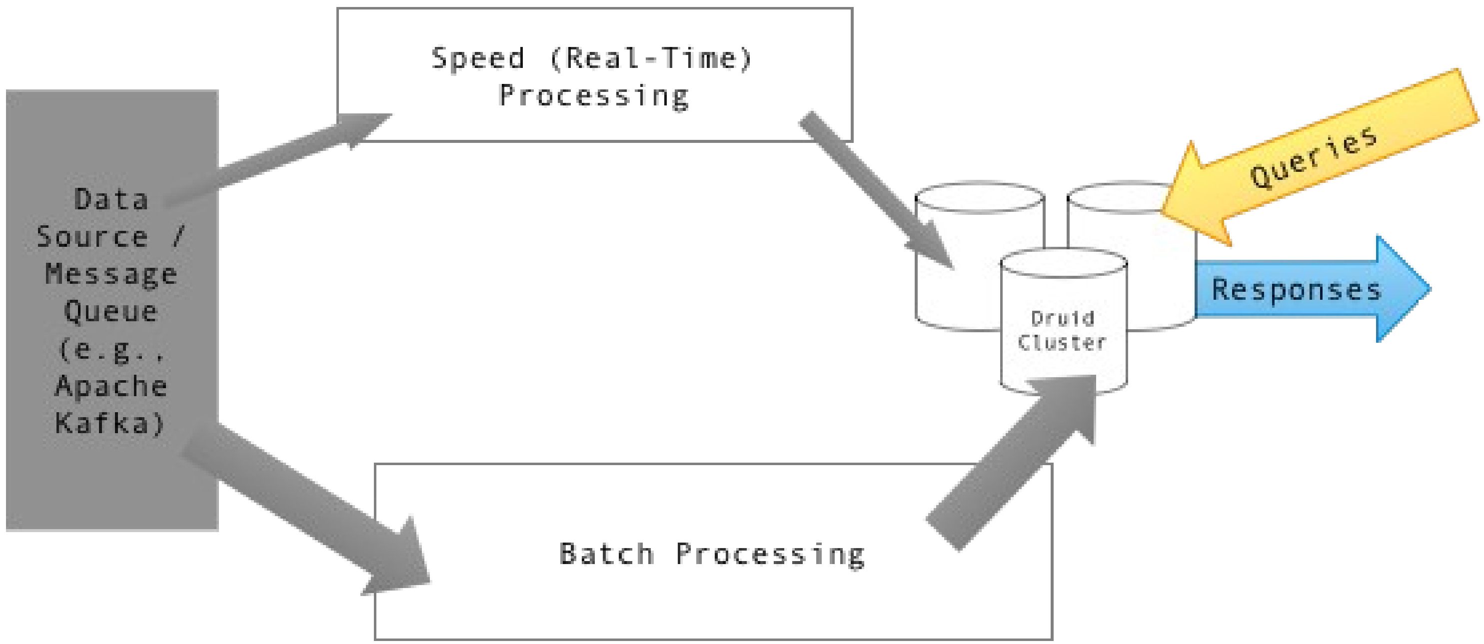
Tony Baer (Ovum) @ ZDNet - Apache Beam and Spark: New coopetition for squashing the Lambda Architecture?

Lambda Architecture - I



https://commons.wikimedia.org/w/index.php?curid=34963986, By Textractor - Own work, CC BY-SA 4

Lambda Architecture - II

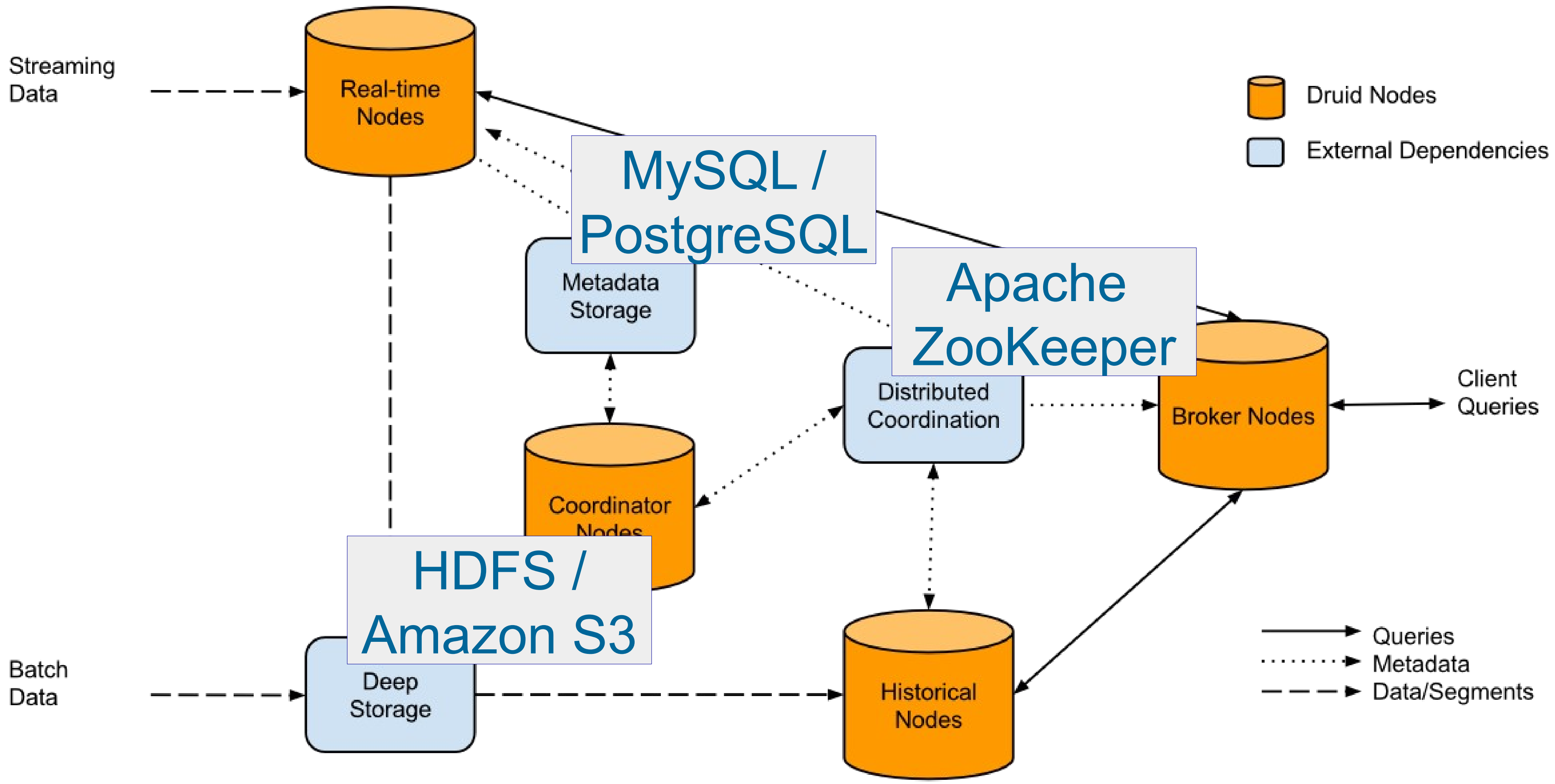


https://commons.wikimedia.org/w/index.php?curid=34963987, By Textractor - Own work, CC BY-SA 4

Lambda Architecture - III

- Data-processing architecture designed to handle massive quantities of data by using both *batch*- and *stream-processing* methods
- Balances *latency, throughput, fault-tolerance, big data, real-time analytics*, mitigates the latencies of map-reduce
- Data model with an append-only, *immutable data* source that serves as a system of record
- Ingesting and processing *timestamped events* that are appended to existing events. State is determined from the *natural time-based ordering* of the data.

Druid Distributed Data Store (Java)



https://commons.wikimedia.org/w/index.php?curid=33899448 By Fangjin Yang - sent to me personally, GFDL

Lambda Architecture: Projects - I

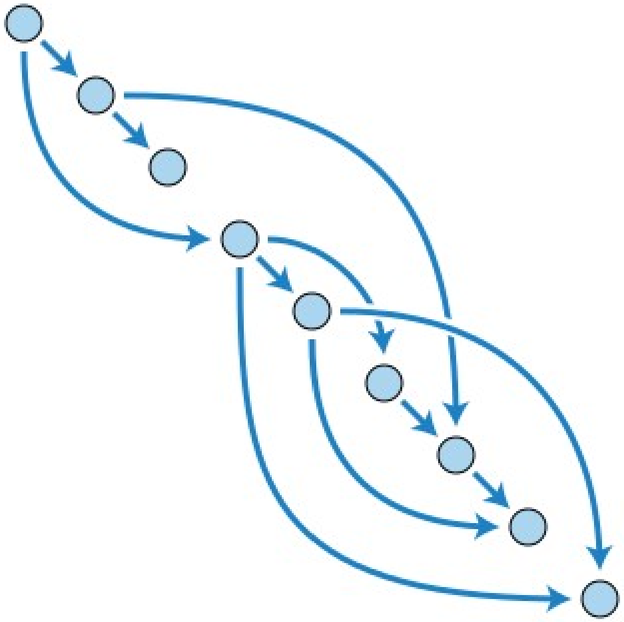
- Apache Spark is an open-source cluster-computing framework. Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. Spark MLlib - a distributed machine learning lib.
- Apache Storm is a distributed stream processing computation framework – uses streams as DAG
- Apache Apex™ unified stream and batch processing engine.

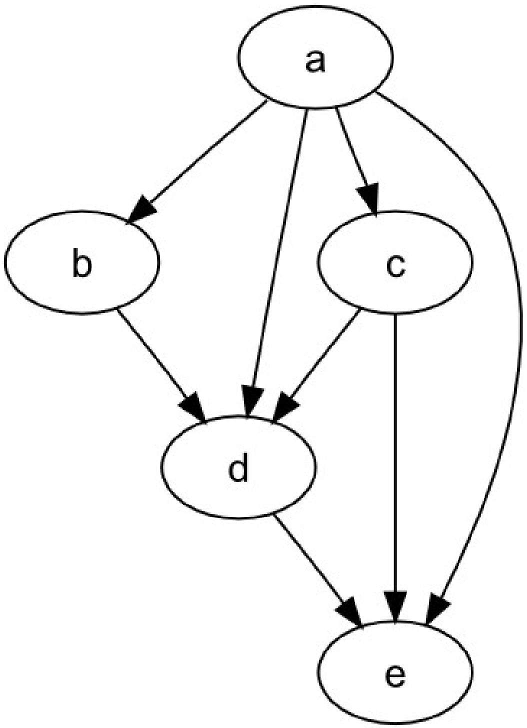
Lambda Architecture: Projects - II

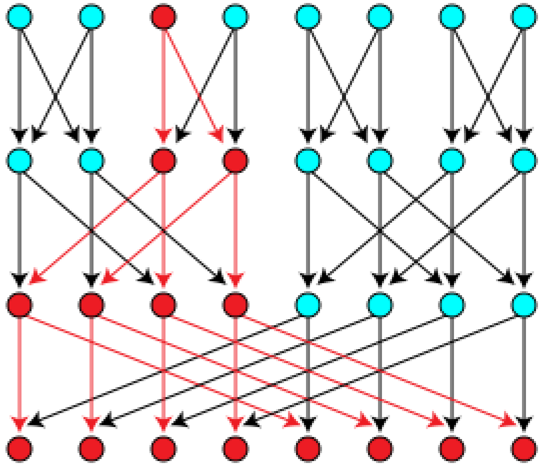
- Apache Flink - open source stream processing framework – Java, Scala
- Apache Beam – unified batch and streaming, portable, extensible
- Apache Kafka - open-source stream processing, real-time, low-latency, unified, high-throughput, massively scalable pub/sub message queue architecture as distributed transaction log - Kafka Streams, a Java library

Beam

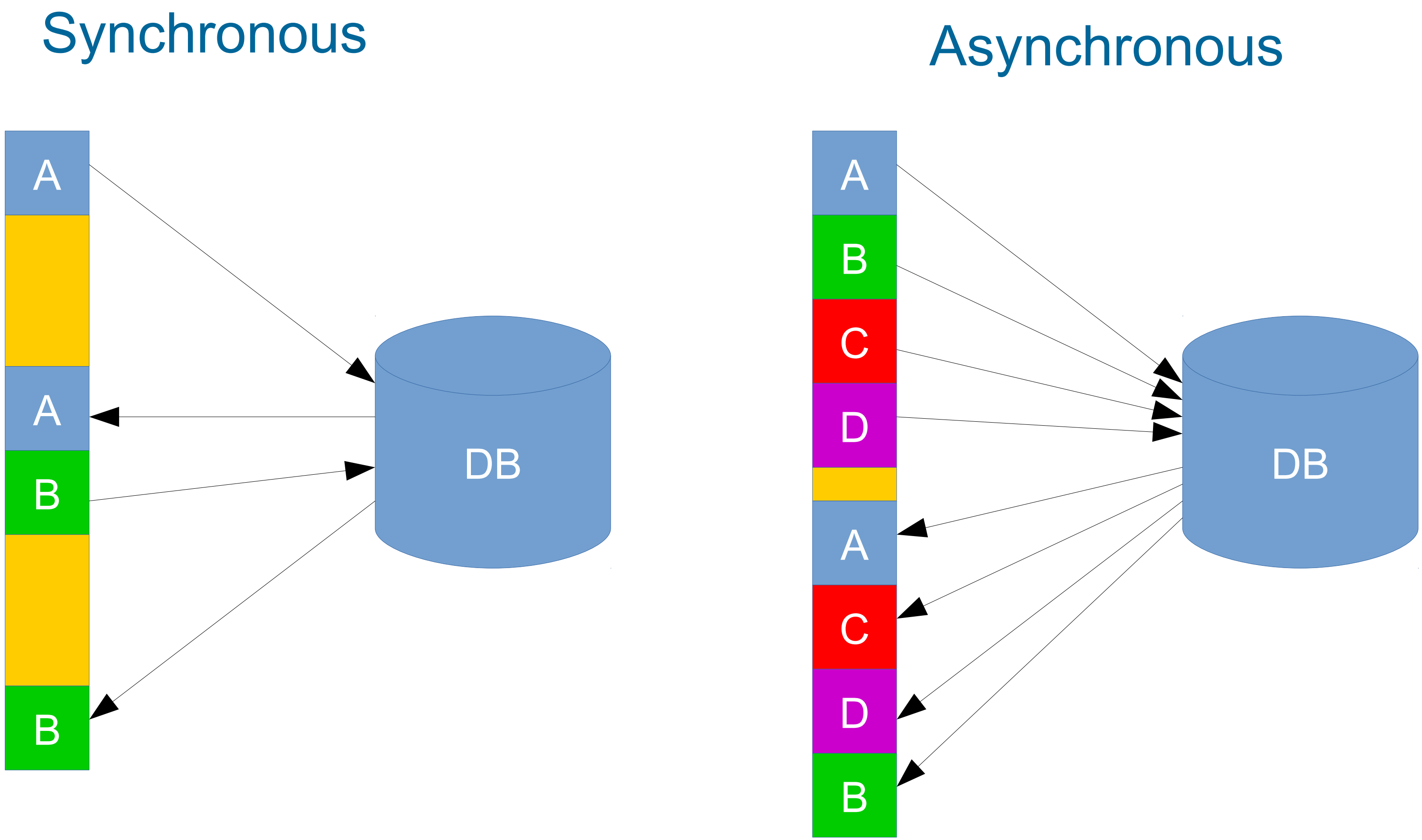
Direct Acyclic Graphs - DAG



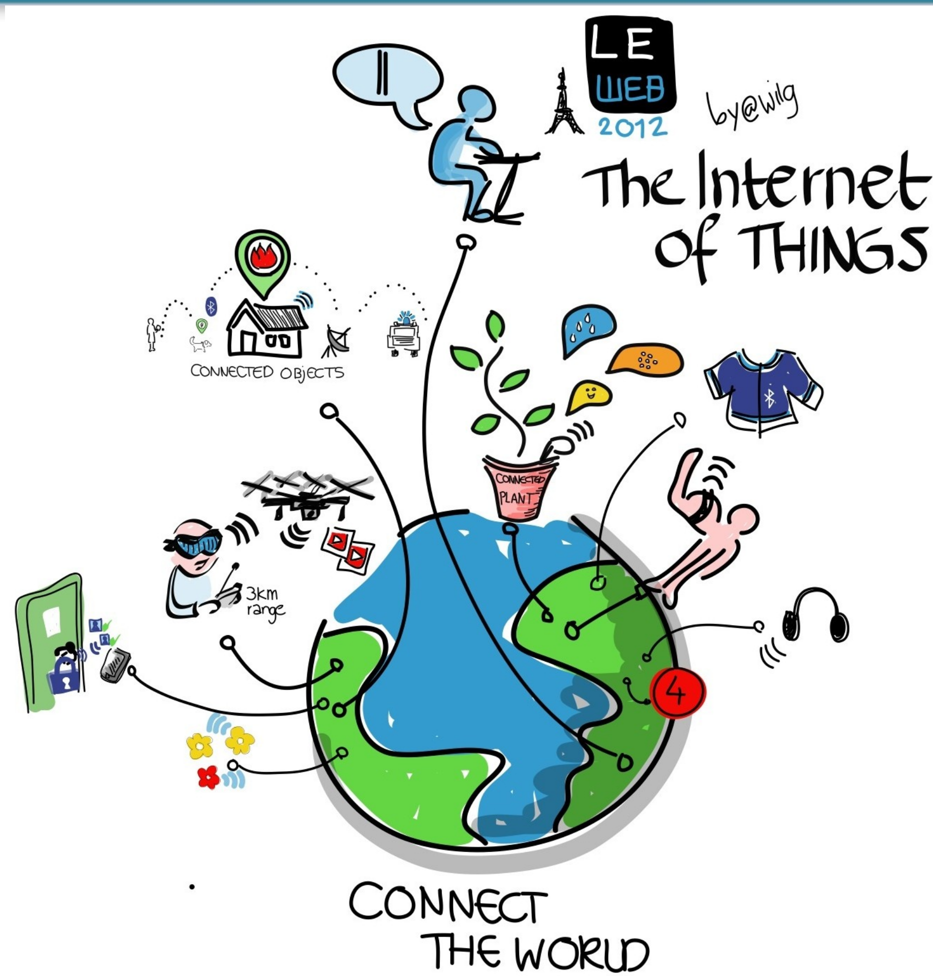




Synchronous vs. Asynchronous IO



Example: Internet of Things (IoT)



Radar, GPS, lidar for navigation and obstacle avoidance (2007 DARPA Urban Challenge)

IoT Services Architecture

Web/ Mobile Portal

Dashboard

PaaS

Cloud (Micro)Service Mng.
Docker, Kubernetes/ Apache Brooklyn

PaaS API: Event Processing Services, Analytics

HTTP, AMQP

Aggregation/ Bus: ESB, Message Broker

M2M: HTTP(/2) / WS / MQTT / CoAP
Management: TR-069 / OMA-DM / OMA LWM2M

Device Gateway: Local Coordination and Event Aggregation

UART/ I2C/ 2G/ 3G/ LTE/ ZigBee/ 6LowPan/ BLE

Devices: Hardware + Embedded Software + Firmware

All Sensors Provide Hot Streams

What's High Performance?

- **Performance** is about 2 things (Martin Thompson – http://www.infoq.com/articles/low-latency-vp):
  - **Throughput** – units per second, and
  - **Latency** – response time
- **Real-time** – time constraint from input to response regardless of system load.
- **Hard real-time system** if this constraint is not honored then a total system failure can occur.
- **Soft real-time system** – low latency response with little deviation in response time
- **100 nano-seconds** to **100 milli-seconds**. [Peter Lawrey]

Low Latency: Things to Remember

- **Low garbage** by reusing existing objects + infrequent GC when application not busy – **can improve app 2 - 5x**
- JVM generational GC startegy – ideal for **objects living very shortly** (garbage collected next minor sweep) **or be immortal**
- **Non-blocking, lockless** coding or **CAS**
- Critical data structures – direct memory access using **DirectByteBuffers** or **Unsafe** => predictable memory layout and cache misses avoidance
- **Busy waiting** – giving the CPU to OS kernel slows program 2-5x => avoid context switches
- Amortize the effect of expensive **IO - blocking**

Mutex Comparison => Conclusions

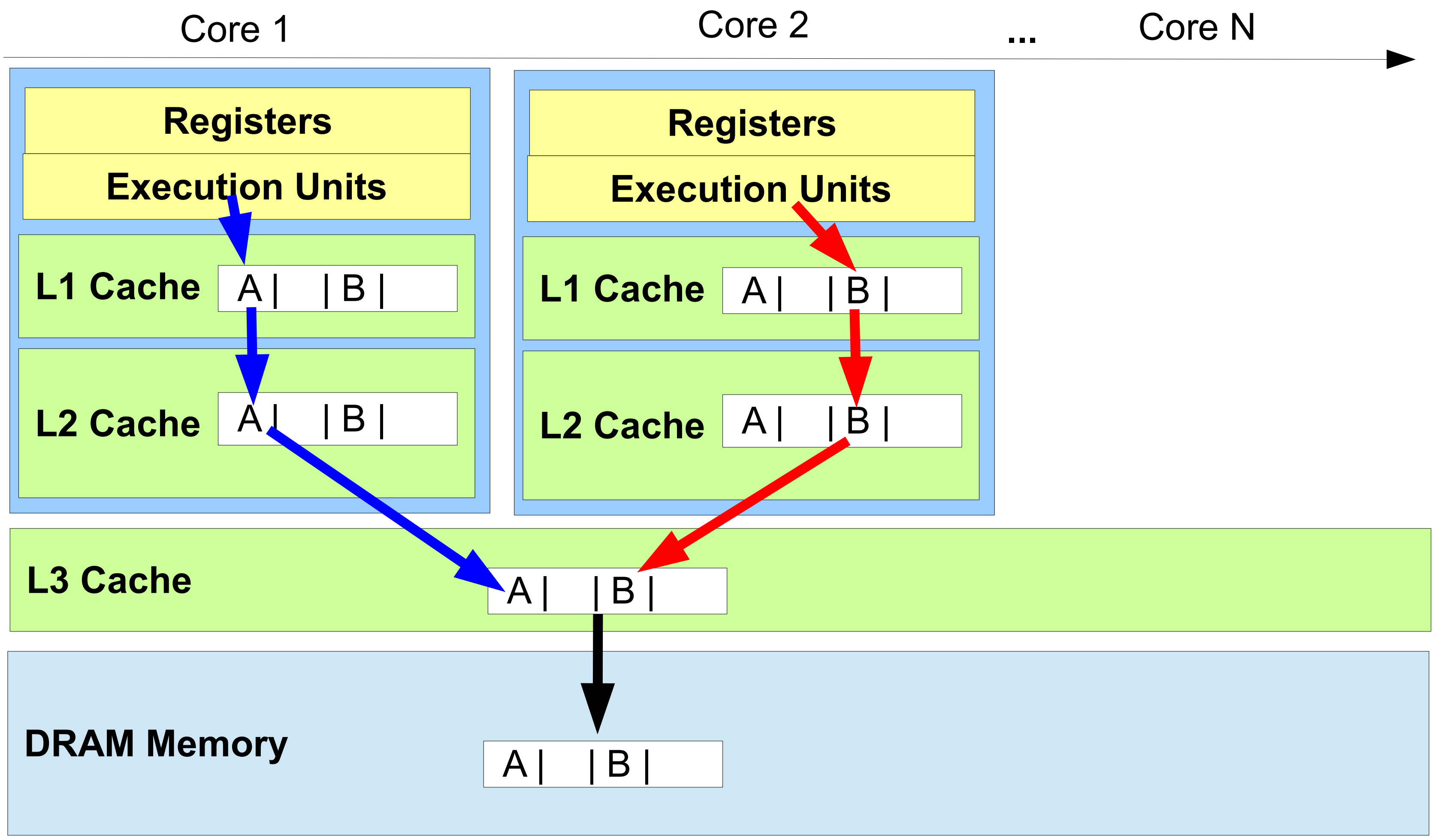
- **Non-blocking (synchronous)** implementation is **2 orders of magnitude** better then **synchronized**
- We should **try to avoid blocking and especially contended blocking** if want to achieve low latency
- If blocking is a must we have to **prefer CAS and optimistic concurrency** over blocking (but 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: HOW is is possible to build concurrency **without blocking?**

Blocking Queues Disadvantages

[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** – producer and consumer **synchronization** and **data storage**

CPU Cache – False Sharing



Tracking Complexity

We need tools to cope with all that complexity inherent in robotics and IoT domains.

Simple solutions are needed – cope with problems through divide and concur on different levels of abstraction:

Domain Driven Design (DDD) – back to basics: domain objects, data and logic.

Described by Eric Evans in his book:
Domain Driven Design: Tackling Complexity in the Heart of Software, 2004

Domain Driven Design

Main concepts:

- Entities, value objects and modules
- Aggregates and Aggregate Roots [Haywood]:

value < entity < aggregate < module < BC

- Aggregate Roots are exposed as **Open Host Services**
- Repositories, Factories and Services:

application services <-> **domain services**

- Separating interface from implementation

Microservices and DDD

Actually DDD require additional efforts (as most other divide and concur modeling approaches :)

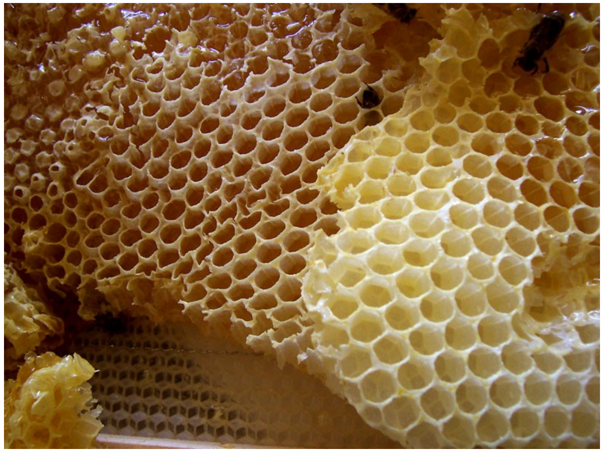
- Ubiquitous language and Bounded Contexts
- DDD Application Layers:

  Infrastructure, Domain, Application, Presentation
- Hexagonal architecture :

OUTSIDE <-> transformer <->

(application <-> domain)

[A. Cockburn]

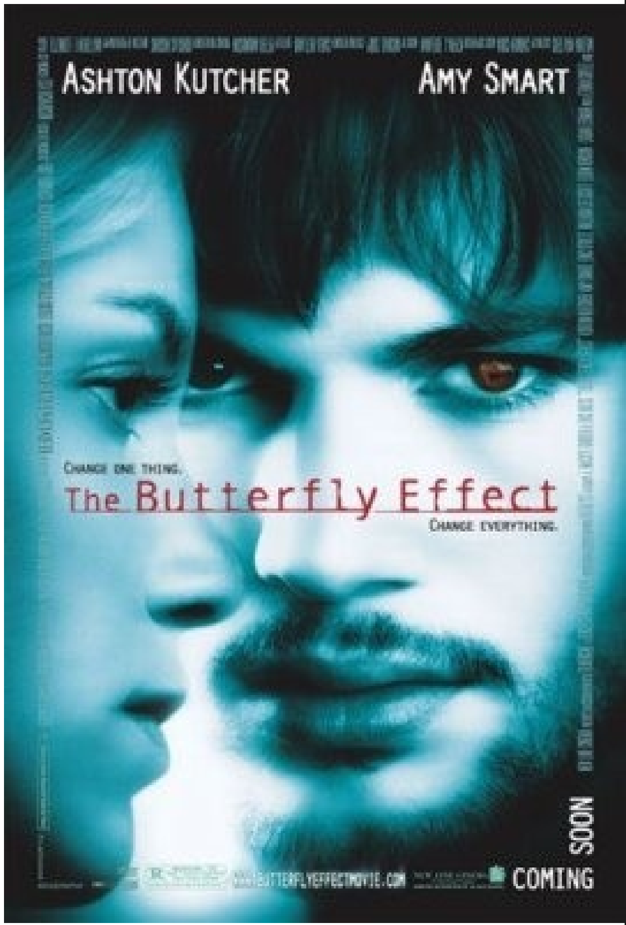


Imperative and Reactive

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.



Imperative and Reactive

- **Reactive Programming:** using static or dynamic data flows and propagation of change

Example: a := b + c

- **Functional Programming:** evaluation of mathematical functions,
  - Avoids changing-state and mutable data, declarative programming
  - Side effects free => much easier to understand and predict the program behavior.

Example: books.stream().filter(book -> book.getYear() > 2010) .forEach(System.out::println)

Functional Reactive (FRP)

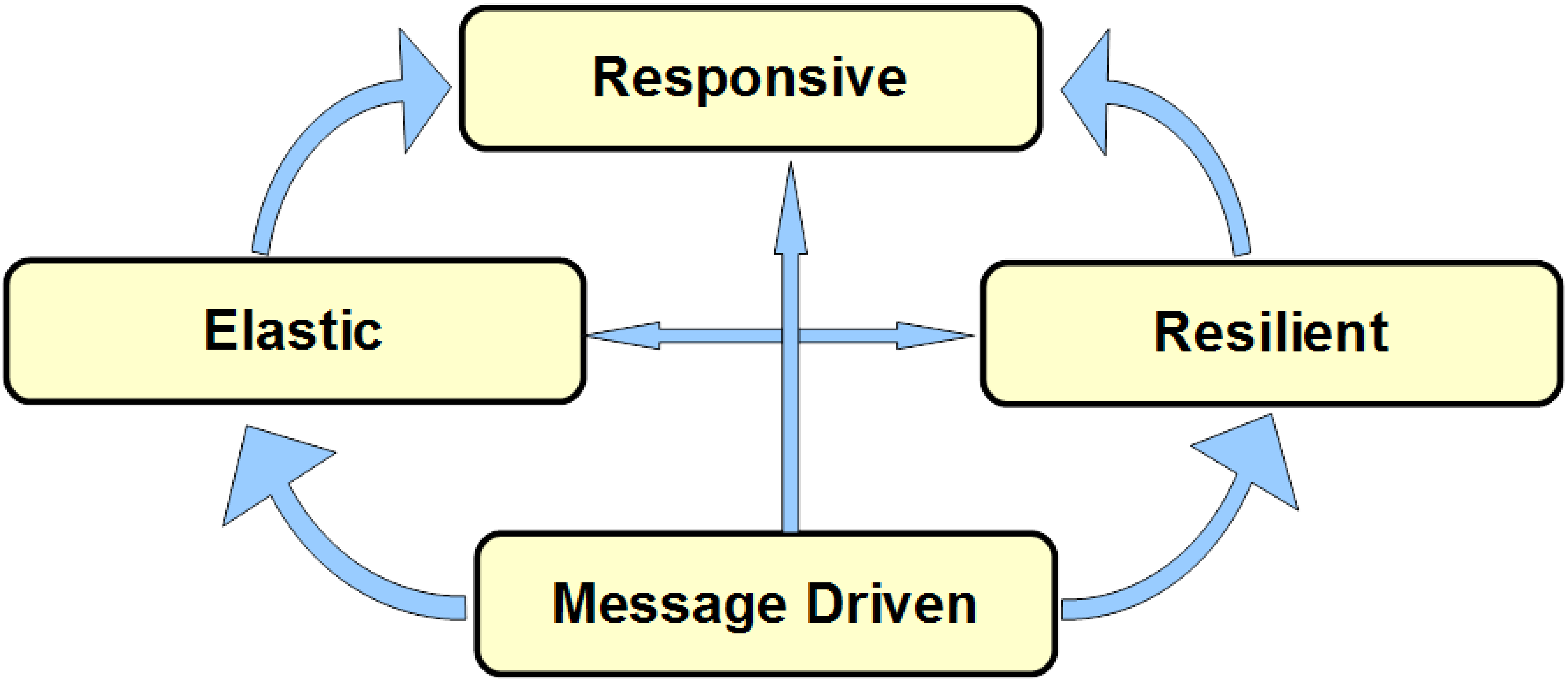
According to **Connal Elliot's** (ground-breaking paper @ Conference on Functional Programming, 1997), **FRP** is:

(a) Denotative

(b) Temporally continuous

Reactive Manifesto

[http://www.reactivemanifesto.org]



Scalable, Massively Concurrent

- **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**

Reactive Programming

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

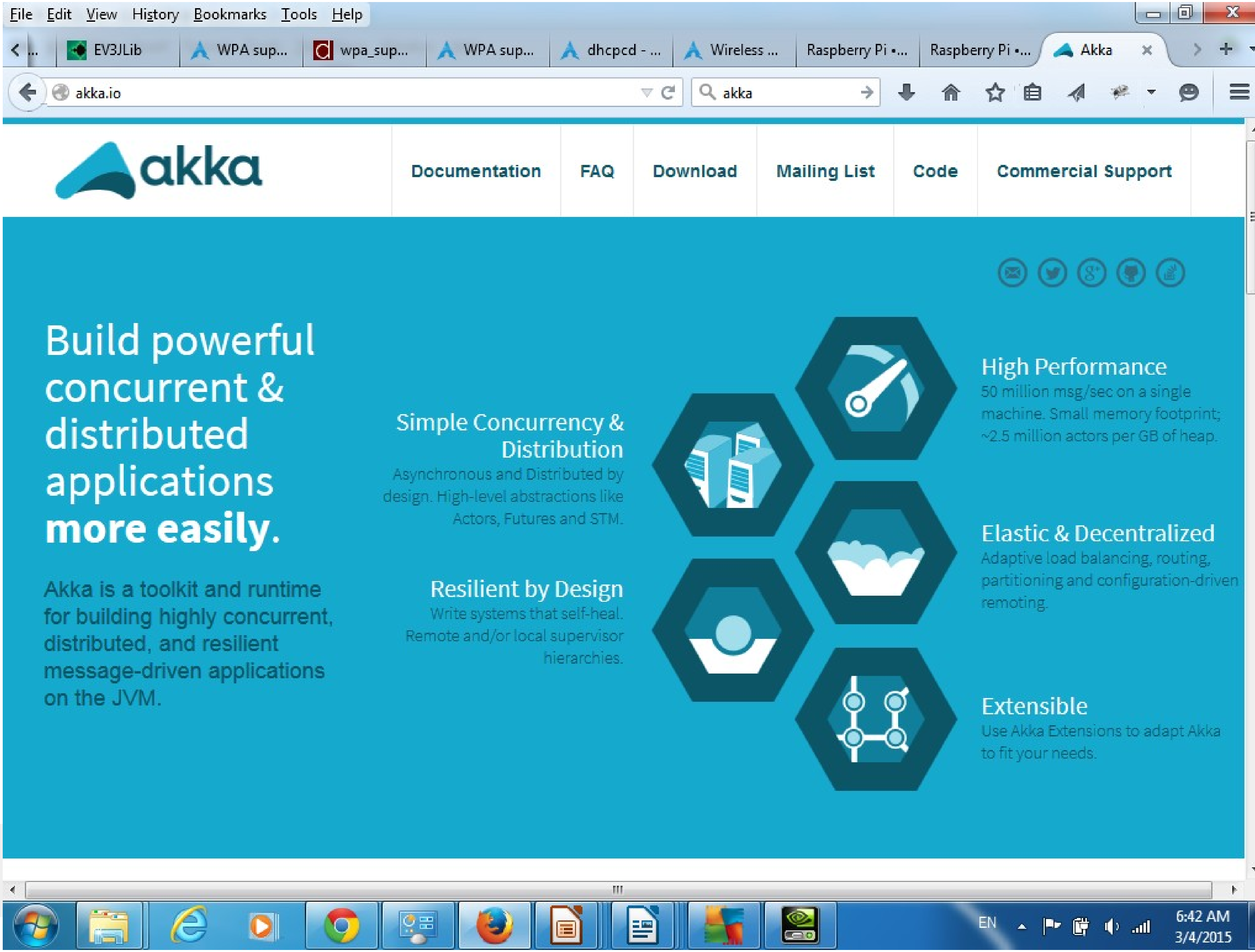
  Rx = Observables + LINQ + Schedulers :)

  Java: RxJava, JavaScript: RxJS, C#: Rx.NET, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Ruby: Rx.rb, Python: RxPY, Groovy: RxGroovy, JRuby: RxJRuby, Kotlin: RxKotlin ...

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

- (Spring) Project Reactor [http://projectreactor.io/]

- Actor Model – Akka (Java, Scala) [http://akka.io/]


File Edit View History Bookmarks Tools Help
EV3JLib
WPA sup...
wpa_sup...
WPA sup...
dhcpcd - ...
Wireless ...
Raspberry Pi •...
Raspberry Pi •...
Akka
akka.io
akka
akka
Documentation
FAQ
Download
Mailing List
Code
Commercial Support
Build powerful concurrent & distributed applications more easily.
Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.
Simple Concurrency & Distribution
Asynchronous and Distributed by design. High-level abstractions like Actors, Futures and STM.
Resilient by Design
Write systems that self-heal. Remote and/or local supervisor hierarchies.
High Performance
50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.
Elastic & Decentralized
Adaptive load balancing, routing, partitioning and configuration-driven remoting.
Extensible
Use Akka Extensions to adapt Akka to fit your needs.
EN
6:42 AM
3/4/2015

Reactive Streams Spec.

- Reactive Streams – provides standard for asynchronous stream processing with non-blocking back pressure.
- Minimal set of interfaces, methods and protocols for asynchronous data streams
- April 30, 2015: has been released version 1.0.0 of Reactive Streams for the JVM (Java API, Specification, TCK and implementation examples)
- Java 9: java.util.concurrent.Flow

Reactive Streams Spec.

- **Publisher** – provider of potentially unbounded number of sequenced elements, according to Subscriber(s) demand.

Publisher.subscribe(Subscriber) => **onSubscribe onNext* (onError | onComplete)?**

- **Subscriber** – calls **Subscription.request(long)** to receive notifications
- **Subscription** – one-to-one **Subscriber** ↔ **Publisher**, request data and cancel demand (allow cleanup).
- **Processor** = **Subscriber** + **Publisher**

FRP = Async Data Streams

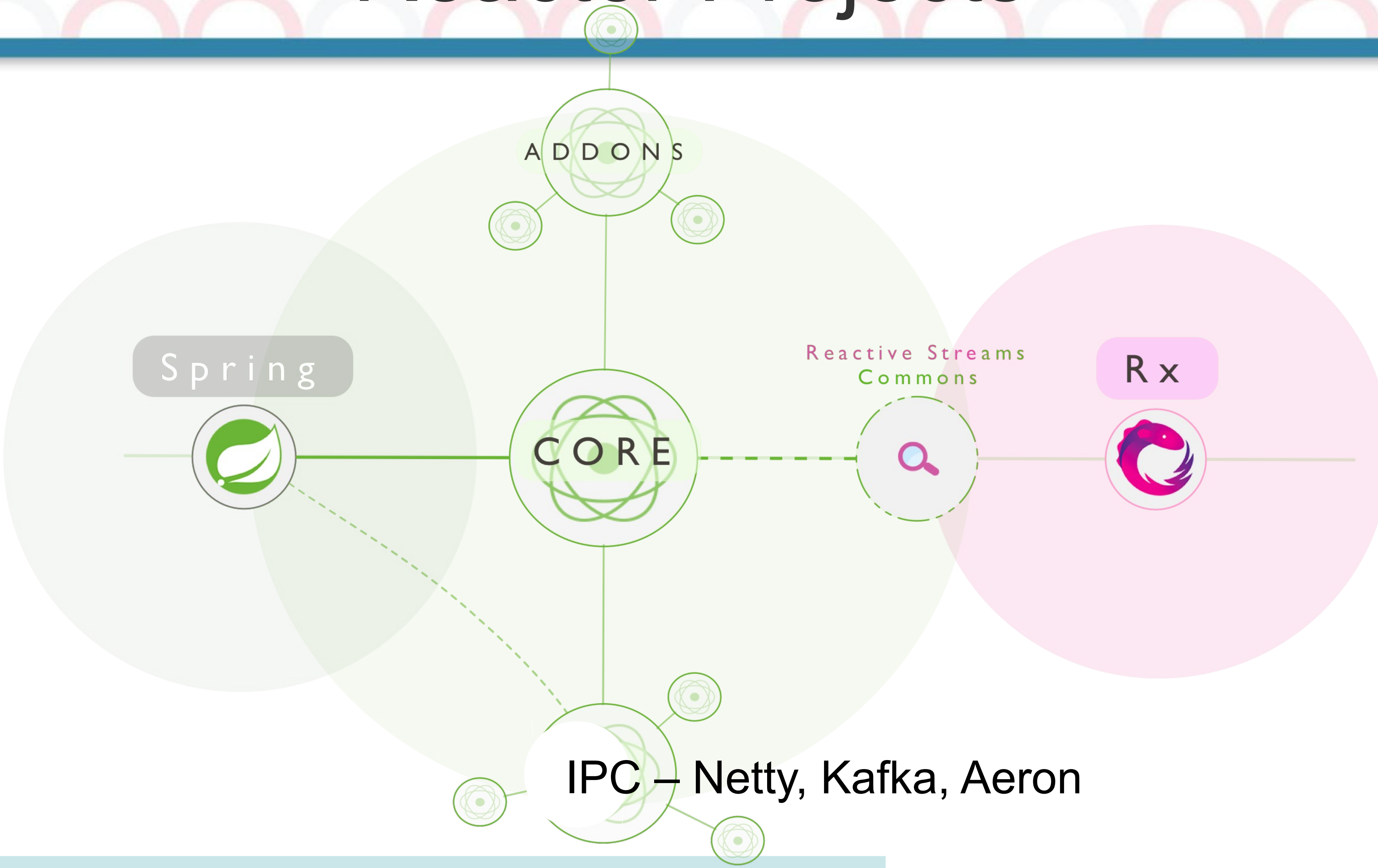
- **FRP** is asynchronous data-flow programming using the building blocks of functional programming (e.g. map, reduce, filter) and explicitly modeling time
- Used for GUIs, robotics, and music. Example (RxJava):

```
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
```

Project Reactor

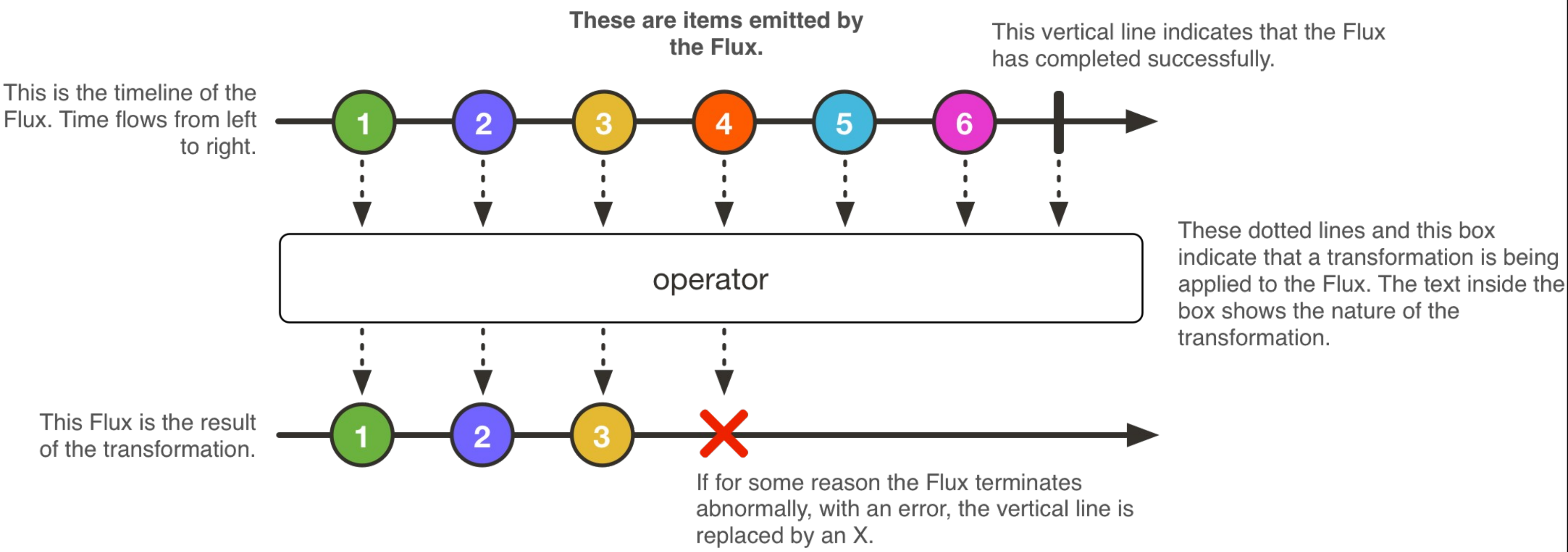
- Reactor project allows building high-performance (low latency high throughput) non-blocking asynchronous applications on JVM.
- Reactor is designed to be extraordinarily fast and can sustain throughput rates on order of 10's of millions of operations per second.
- Reactor has powerful API for declaring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

Reactor Projects

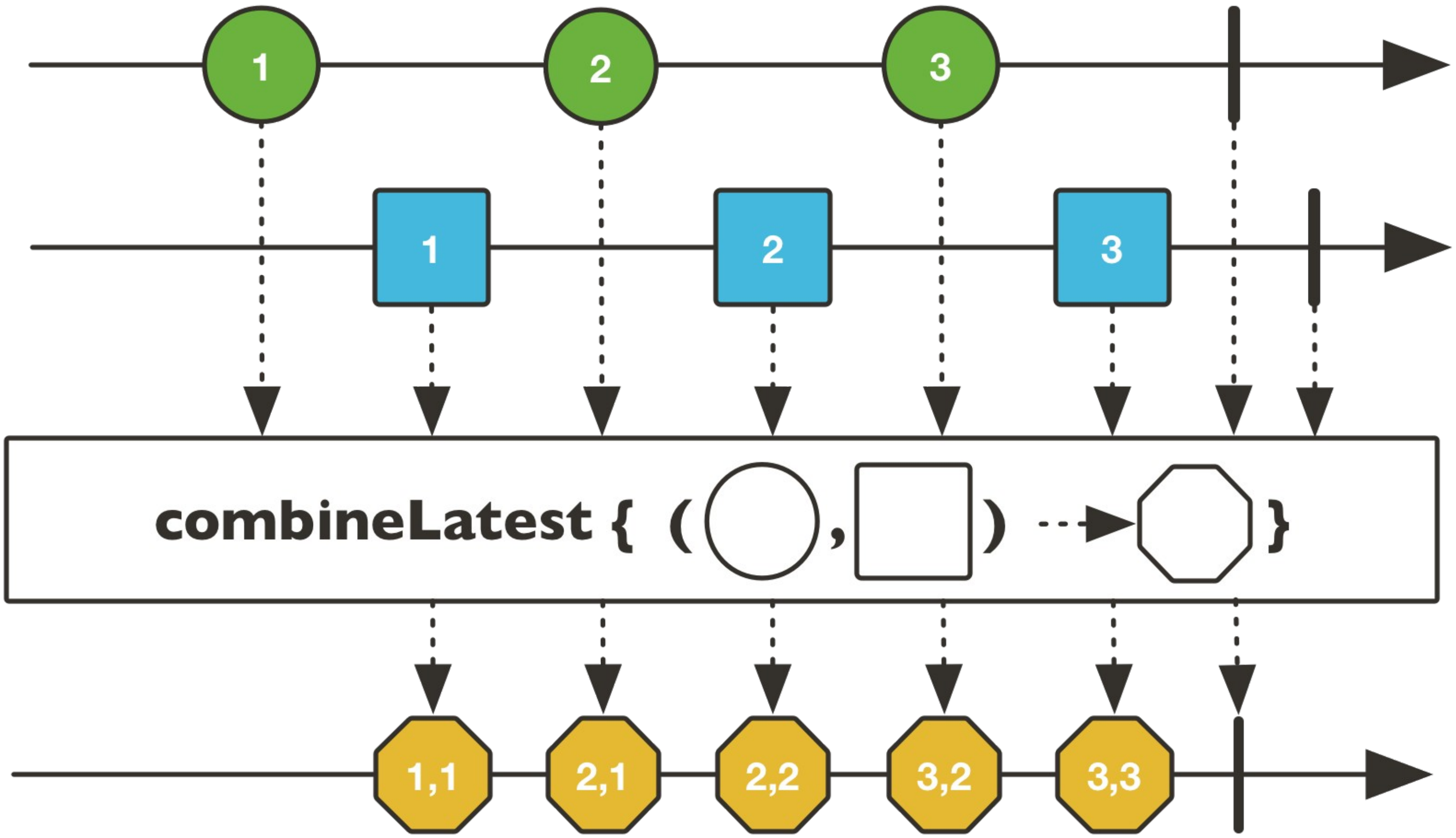


https://github.com/reactor/reactor, Apache Software License 2.0

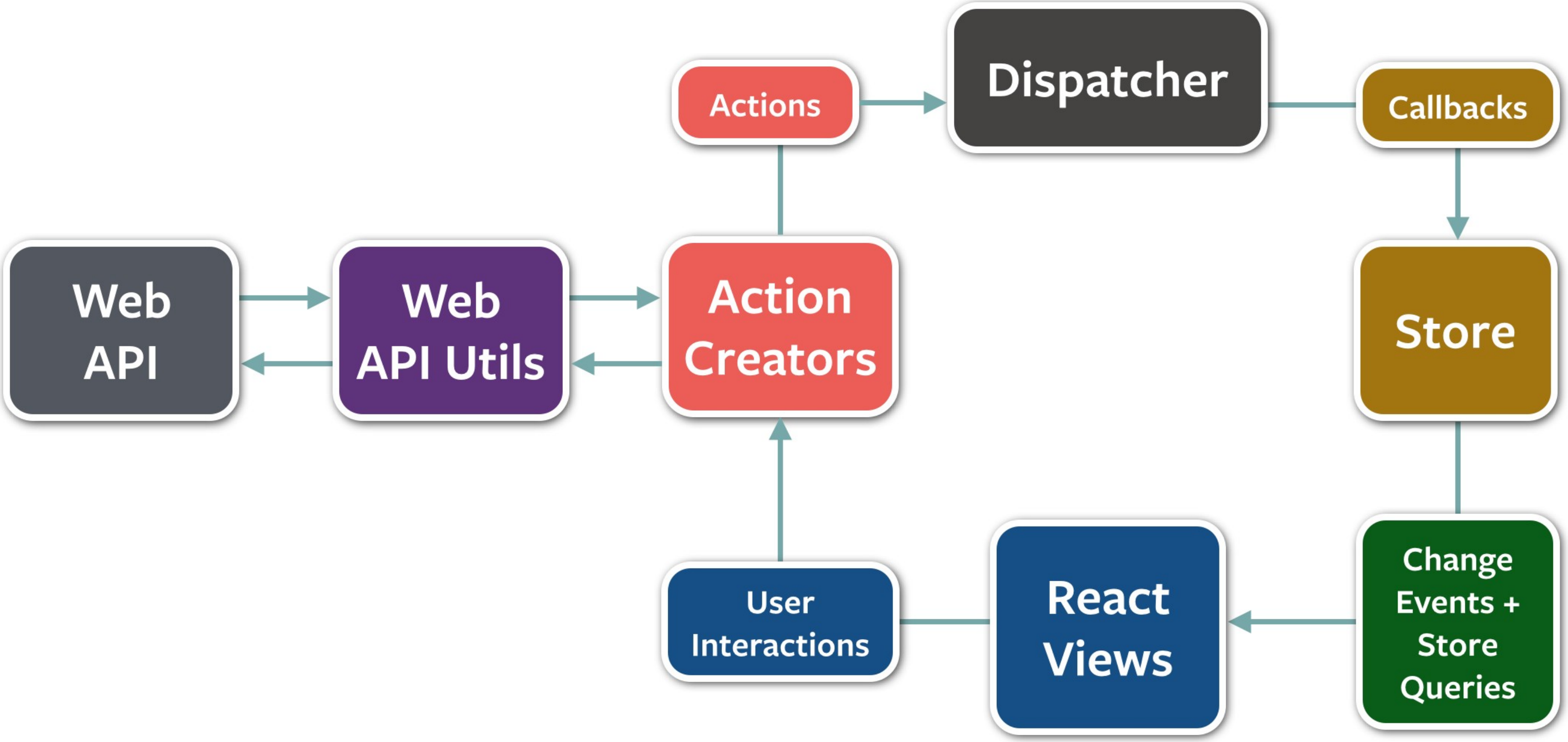
Reactor Flux



Example: Flux.combineLatest()

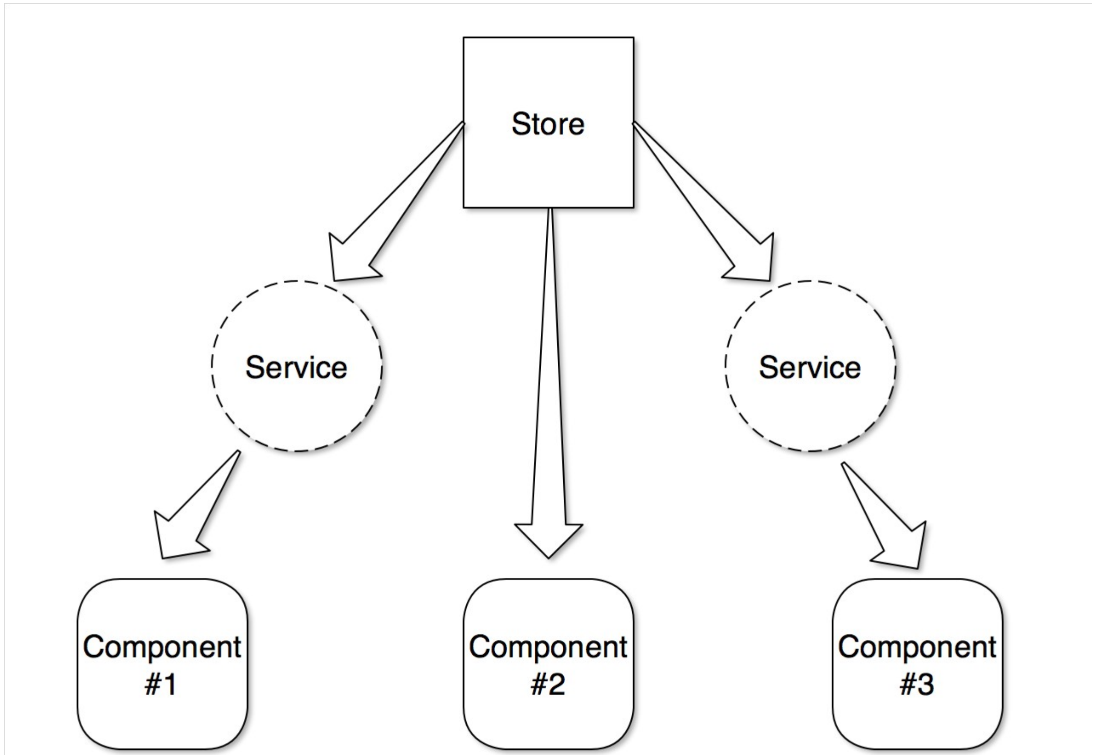


Flux Design Pattern



Source: Flux in GitHub, https://github.com/facebook/flux, License: BSD 3-clause "New" License

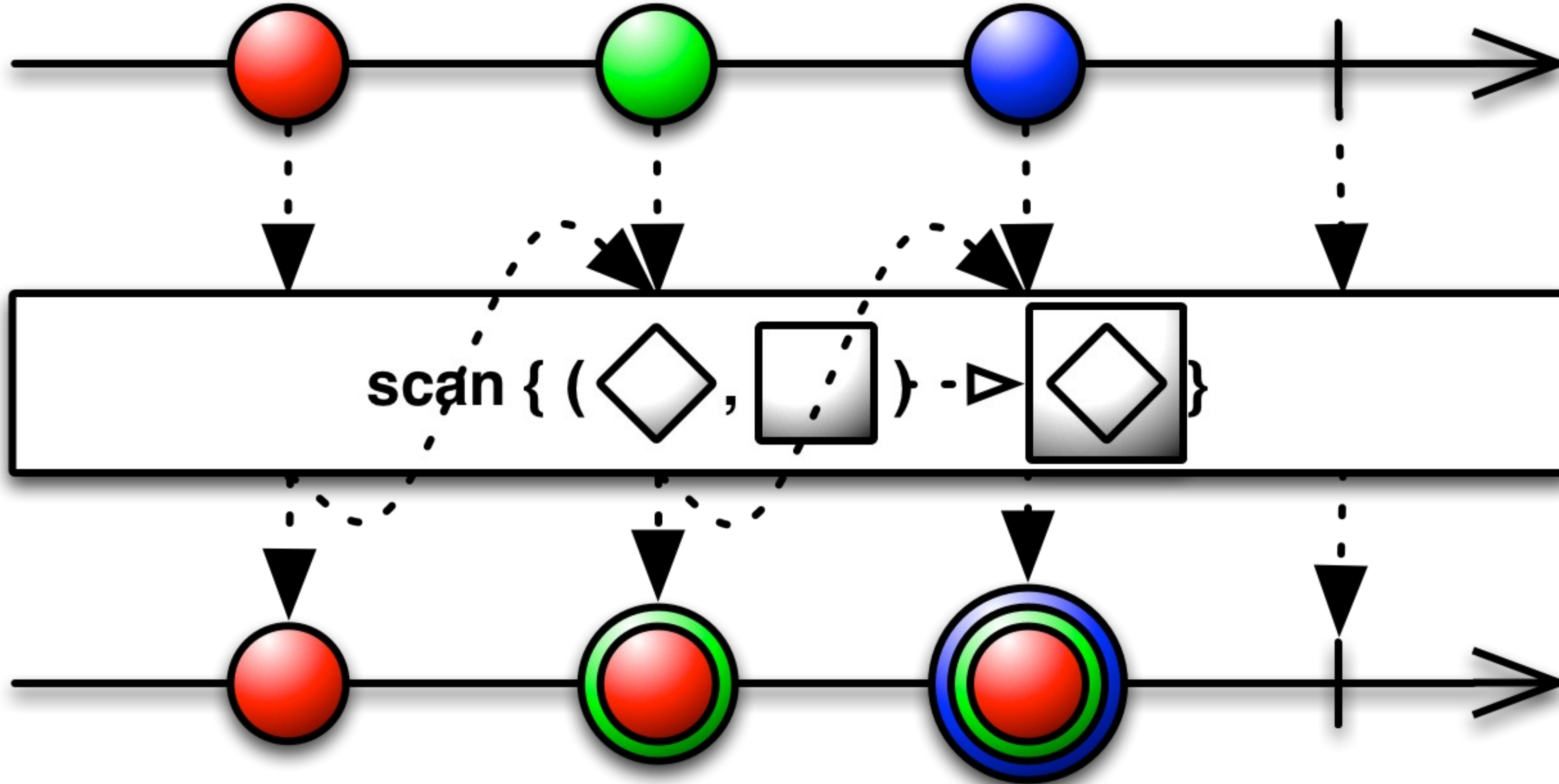
Redux Design Pattern



Linear flow: **Dispatch** ⇨ **Reducers** ⇨ **New State** ⇨ **Store**

Source: @ngrx/store in GitHub, https://gist.github.com/btroncone/a6e4347326749f938510

Redux == Rx Scan Opearator



Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/

Hot and Cold Event Streams

- **PULL-based (Cold Event Streams)** – Cold streams (e.g. RxJava Observable / Flowable or Reactor Flow / Mono) are streams that run their sequence when and if they are subscribed to. They present the sequence from the start to each subscriber.
- **PUSH-based (Hot Event Streams)** – Hot streams emit values independent of individual subscriptions. They have their own timeline and events occur whether someone is listening or not. An example of this is mouse events. A mouse is generating events regardless of whether there is a subscription. When subscription is made observer receives current events as they happen.

Cold RxJava 2 Flowable Example

```java
Flowable<String> cold = Flowable.just("Hello",
"Reactive", "World", "from", "RxJava", "!");
cold.subscribe(i -> System.out.println("First: " + i));
Thread.sleep(500);
cold.subscribe(i -> System.out.println("Second: " + i));
```

Results:

```
First: Hello
First: Reactive
First: World
First: from
First: RxJava
First: !
```

```
Second: Hello
Second: Reactive
Second: World
Second: from
Second: RxJava
Second: !
```

Cold RxJava Example 2

```
Flowable<Long> cold = Flowable.intervalRange(1,10,0,200,
  TimeUnit.MILLISECONDS);
cold.subscribe(i -> System.out.println("First: " + i));
Thread.sleep(500);
cold.subscribe(i -> System.out.println("Second: " + i));
Thread.sleep(3000);
```

Results:

```
First: 1
First: 2
First: 3
Second: 1
First: 4
Second: 2
First: 5
Second: 3
First: 6
Second: 4
First: 7
Second: 5
First: 8
Second: 6
First: 9
Second: 7
First: 10
Second: 8
Second: 9
Second: 10
```

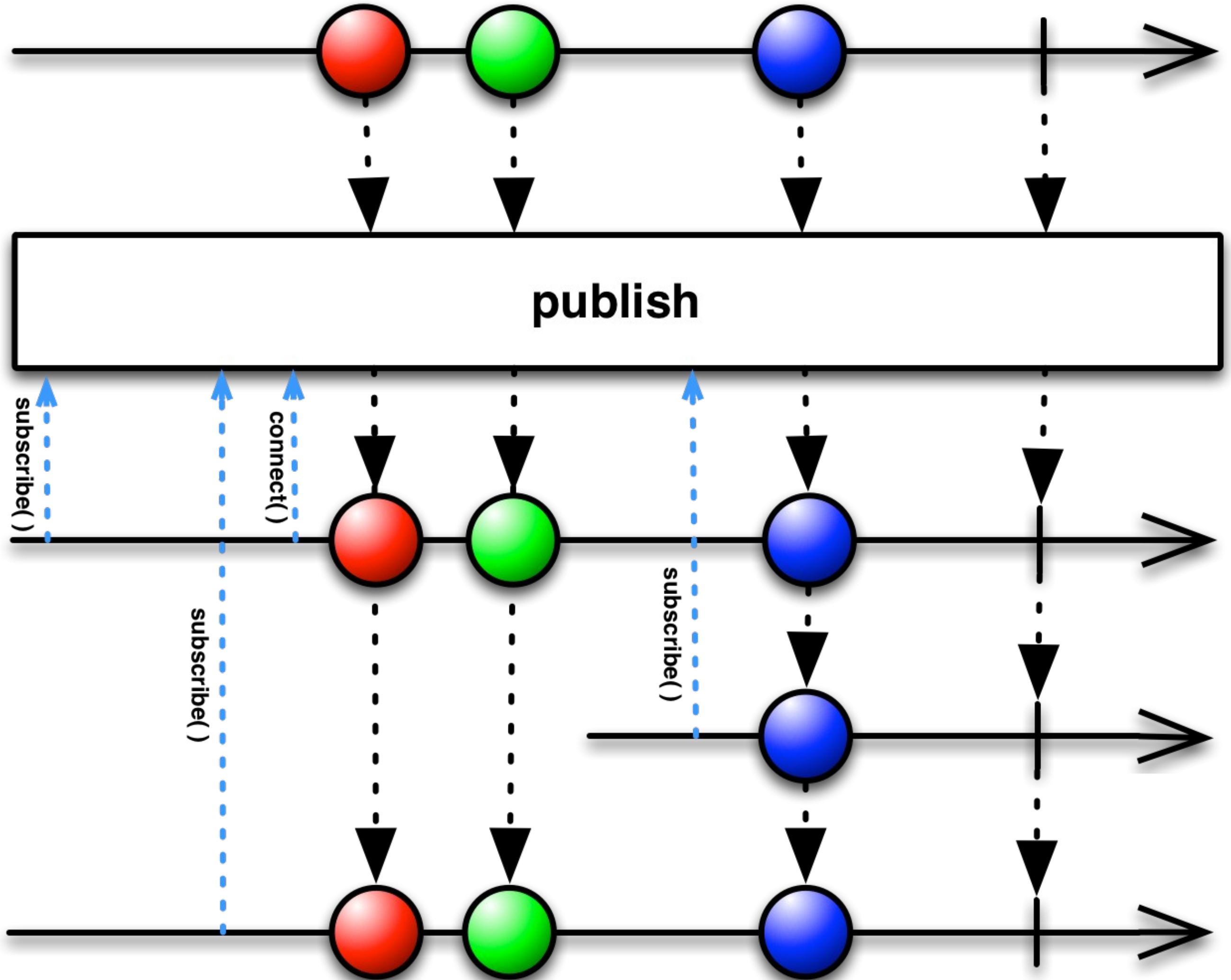
Hot Stream RxJava 2 Example

```
ConnectableFlowable<Long> hot =
Flowable.intervalRange(1,10,0,200,TimeUnit.MILLISECONDS)
  .publish();;
hot.connect(); // start emmiting Flowable -> Subscribers
hot.subscribe(i -> System.out.println("First: " + i));
Thread.sleep(500);
hot.subscribe(i -> System.out.println("Second: " + i));
Thread.sleep(3000);
```

Results:

```
First: 2
First: 3
First: 4
Second: 4
First: 5
Second: 5
First: 6
Second: 6
First: 7
Second: 7
First: 8
Second: 8
First: 9
Second: 9
First: 10
Second: 10
```

Converting Cold to Hot Stream



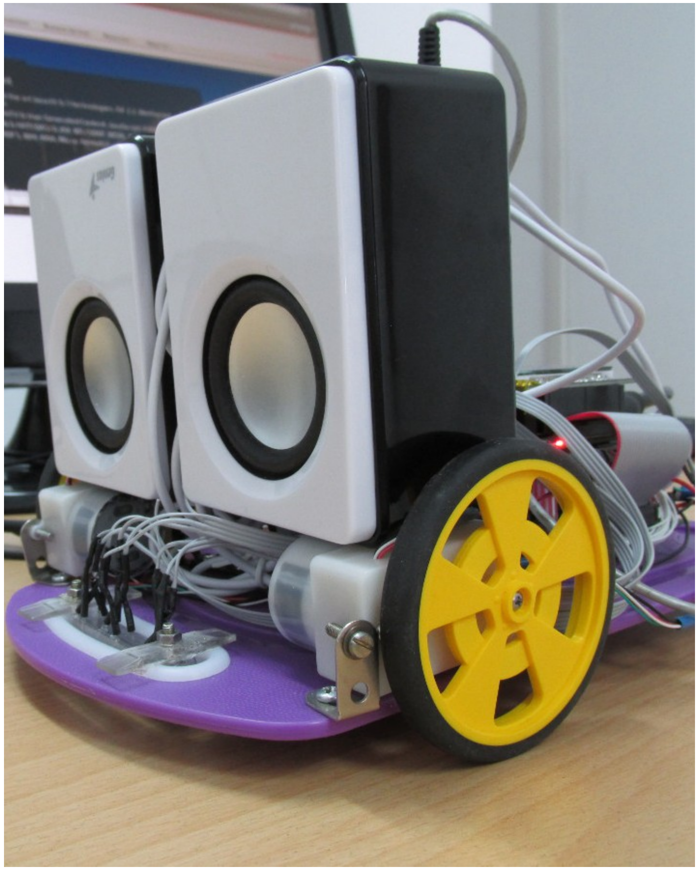
Cold Stream Example – Reactor

```
Flux.fromIterable(getSomeLongList())
    .mergeWith(Flux.interval(100))
    .doOnNext(serviceA::someObserver)
    .map(d -> d * 2)
    .take(3)
    .onErrorResumeWith(errorHandler::fallback)
    .doAfterTerminate(serviceM::incrementTerminate)
    .subscribe(System.out::println);
```

Hot Stream Example - Reactor

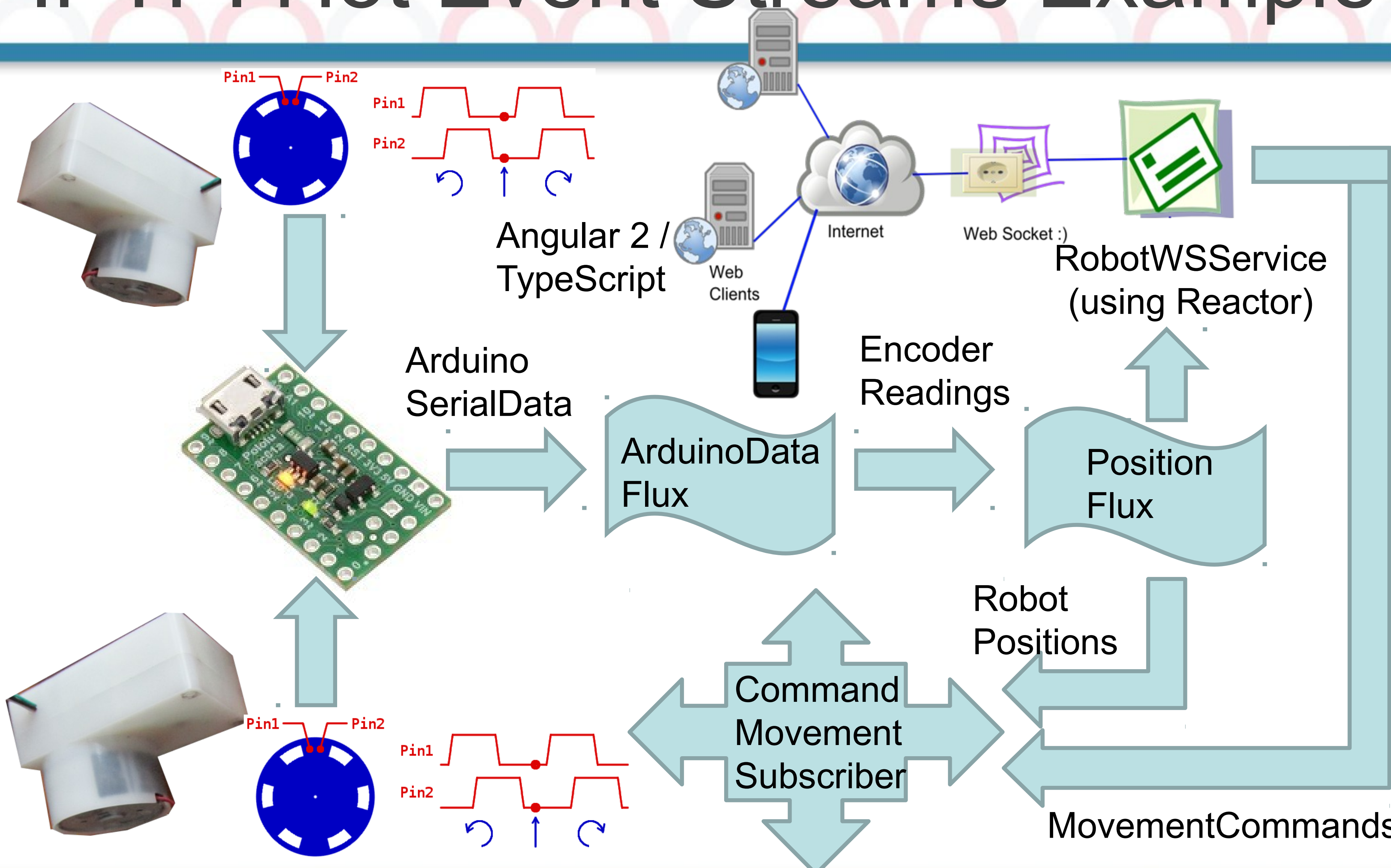
```
public static void main(String... args) throws
  InterruptedException {
  EmitterProcessor<String> emitter =
        EmitterProcessor.create();
  BlockingSink<String> sink = emitter.connectSink();
  emitter.publishOn(Schedulers.single())
   .map(String::toUpperCase)
   .filter(s → s.startsWith("HELLO"))
   .delayMillis(1000).subscribe(System.out::println);
  sink.submit("Hello World!"); // emit - non blocking
  sink.submit("Goodbye World!");
  sink.submit("Hello Trayan!");
  Thread.sleep(3000);
}
```

Example: IPTPI - RPi + Ardunio Robot



- Raspberry Pi 2 (quad-core ARMv7 @ 900MHz) + Arduino Leonardo cloneA-Star 32U4 Micro
- *Optical encoders* (custom), IR optical array, 3D accelerometers, gyros, and compass MinIMU-9 v2
- **IPTPI** is programmed in Java using Pi4J, Reactor, RxJava, Akka
- More information about IPTPI: http://robolearn.org/iptpi-robot/

IPTPI Hot Event Streams Example



Futures in Java 8 - I

- Future (implemented by FutureTask) – represents the result of an cancelable asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation (blocking till its ready).
- RunnableFuture – a Future that is Runnable. Successful execution of the run method causes Future completion, and allows access to its results.
- ScheduledFuture – delayed cancelable action that returns result. Usually a scheduled future is the result of scheduling a task with a ScheduledExecutorService

Future Use Example

```
Future<String> future = executor.submit(
    new Callable<String>() {
        public String call() {
            return searchService.findByTags(tags);
        }
    }
);

DoSomethingOther();

try {
    showResult(future.get()); // use future result
} catch (ExecutionException ex) { cleanup(); }
```

Futures in Java 8 - II

- CompletableFuture – a Future that may be explicitly completed (by setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.
- CompletionStage – a stage of possibly asynchronous computation, that is triggered by completion of previous stage or stages (CompletionStages form Direct Acyclic Graph – DAG). A stage performs an action or computes value and completes upon termination of its computation, which in turn triggers next dependent stages. Computation may be Function (apply), Consumer (accept), or Runnable (run).

CompletableFuture Example - I

```
private CompletableFuture<String>
  longCompletableFutureTask(int i, Executor executor) {
    return CompletableFuture.supplyAsync(() -> {
      try {
          Thread.sleep(1000); // long computation :)
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
      return i + "-" + "test";
  }, executor);
}
```

CompletableFuture Example - II

```
ExecutorService executor =  ForkJoinPool.commonPool();
//ExecutorService executor = Executors.newCachedThreadPool();
public void testlCompletableFutureSequence() {
List<CompletableFuture<String>> futuresList =
  IntStream.range(0, 20).boxed()
    .map(i -> longCompletableFutureTask(i, executor)
    .exceptionally(t -> t.getMessage()))
    .collect(Collectors.toList());
CompletableFuture<List<String>> results =
  CompletableFuture.allOf(
    futuresList.toArray(new CompletableFuture[0]))
    .thenApply(v -> futuresList.stream()
      .map(CompletableFuture::join)
      .collect(Collectors.toList())
    );
```

CompletableFuture Example - III

```
 try {
   System.out.println(results.get(10, TimeUnit.SECONDS));
 } catch (ExecutionException | TimeoutException
                | InterruptedException e) {
   e.printStackTrace();
 }
 executor.shutdown();
}


// OR just:
System.out.println(results.join());
executor.shutdown();
```

Which is better?

CompletionStage

- Computation may be Function (apply), Consumer (accept), or Runnable (run) – e.g.:

```
completionStage.thenApply( x -> x * x )
        .thenAccept(System.out::print )
        .thenRun( System.out::println )
```

- Stage computation can be triggered by completion of 1 (then), 2 (combine), or either 1 of 2 (either)
- Functional composition can be applied to stages themselves instead to their results using compose
- handle & whenComplete – support unconditional computation – both normal or exceptional triggering

CompletionStages Composition

```
public void testlCompletableFutureComposition() throws
InterruptedException, ExecutionException {
  Double priceInEuro = CompletableFuture.supplyAsync(()
      -> getStockPrice("GOOGL"))
    .thenCombine(CompletableFuture.supplyAsync(() ->
      getExchangeRate(USD, EUR)), this::convertPrice)
    .exceptionally(throwable -> {
      System.out.println("Error: " +
        throwable.getMessage());
          return -1d;
      }).get();

  System.out.println("GOOGL stock price in Euro: " +
      priceInEuro );
}
```

New in Java 9: CompletableFuture

- Executor **defaultExecutor()**
- CompletableFuture<U> **newIncompleteFuture()**
- CompletableFuture<T> **copy()**
- CompletionStage<T> **minimalCompletionStage()**
- CompletableFuture<T> **completeAsync(**
  Supplier<? extends T> supplier[, Executor executor]**)**
- CompletableFuture<T> **orTimeout(**
  long timeout, TimeUnit unit**)**
- CompletableFuture<T> **completeOnTimeout(**
  T value, long timeout, TimeUnit unit**)**

More Demos ...

CompletableFuture, Flow & RxJava2 @ GitHub:

https://github.com/iproduct/reactive-demos-java-9

- completable-future-demo – composition, delayed, ...
- flow-demo – custom Flow implementations using CFs
- rxjava2-demo – RxJava2 intro to reactive composition
- completable-future-jaxrs-cdi-cxf – async observers, ...
- completable-future-jaxrs-cdi-jersey
- completable-future-jaxrs-cdi-jersey-client

Ex.1: Async CDI Events with CF

```
@Inject @CpuProfiling private Event<CpuLoad> event; ...
IntervalPublisher.getDefaultIntervalPublisher(
  500, TimeUnit.MILLISECONDS) // Custom CF Flow Publisher
  .subscribe(new Subscriber<Integer>() {
    @Override public void onComplete() {}
    @Override public void onError(Throwable t) {}
    @Override public void onNext(Integer i) {
      event.fireAsync(new CpuLoad(
        System.currentTimeMillis(), getJavaCPULoad(),
        areProcessesChanged()))
      .thenAccept(event -> {
        logger.info("CPU load event fired: " + event);
      }); } //firing CDI async event returns CF
    @Override public void onSubscribe(Subscription
subscription) {subscription.request(Long.MAX_VALUE);} });
```

Ex.2: Reactive JAX-RS Client - CF

```
CompletionStage<List<ProcessInfo>> processesStage =
  processes.request().rx()
    .get(new GenericType<List<ProcessInfo>>() {})
    .exceptionally(throwable -> {
      logger.error("Error: " + throwable.getMessage());
      return Collections.emptyList();
    });

CompletionStage<Void> printProcessesStage =
  processesStage.thenApply(proc -> {
    System.out.println("Active JAVA Processes: " + proc);
    return null;
});
```

Ex.2: Reactive JAX-RS Client - CF

```
(- continues -)

printProcessesStage.thenRun( () -> {
  try (SseEventSource source =
    SseEventSource.target(stats).build()) {
      source.register(System.out::println);
      source.open();
      Thread.sleep(20000); // Consume events for 20 sec
  } catch (InterruptedException e) {
    logger.info("SSE consumer interrupted: " + e);
  }
})
.thenRun(() -> {System.exit(0);});
```

Thank's for Your Attention!

Trayan Iliev

CEO of IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD

https://plus.google.com/+IproductOrg