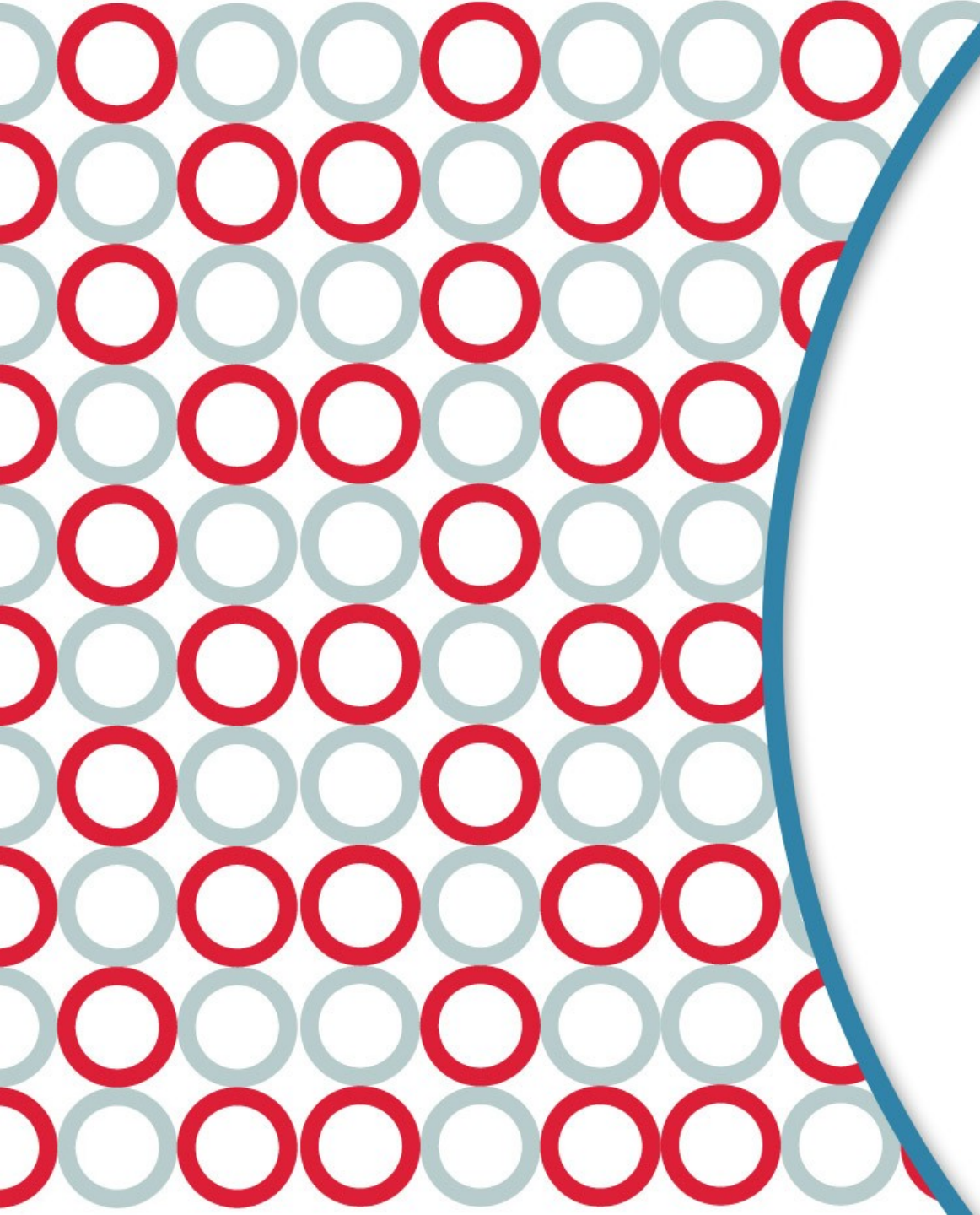


# Reactive Java and IoT

**Trayan Iliev**

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

# 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.

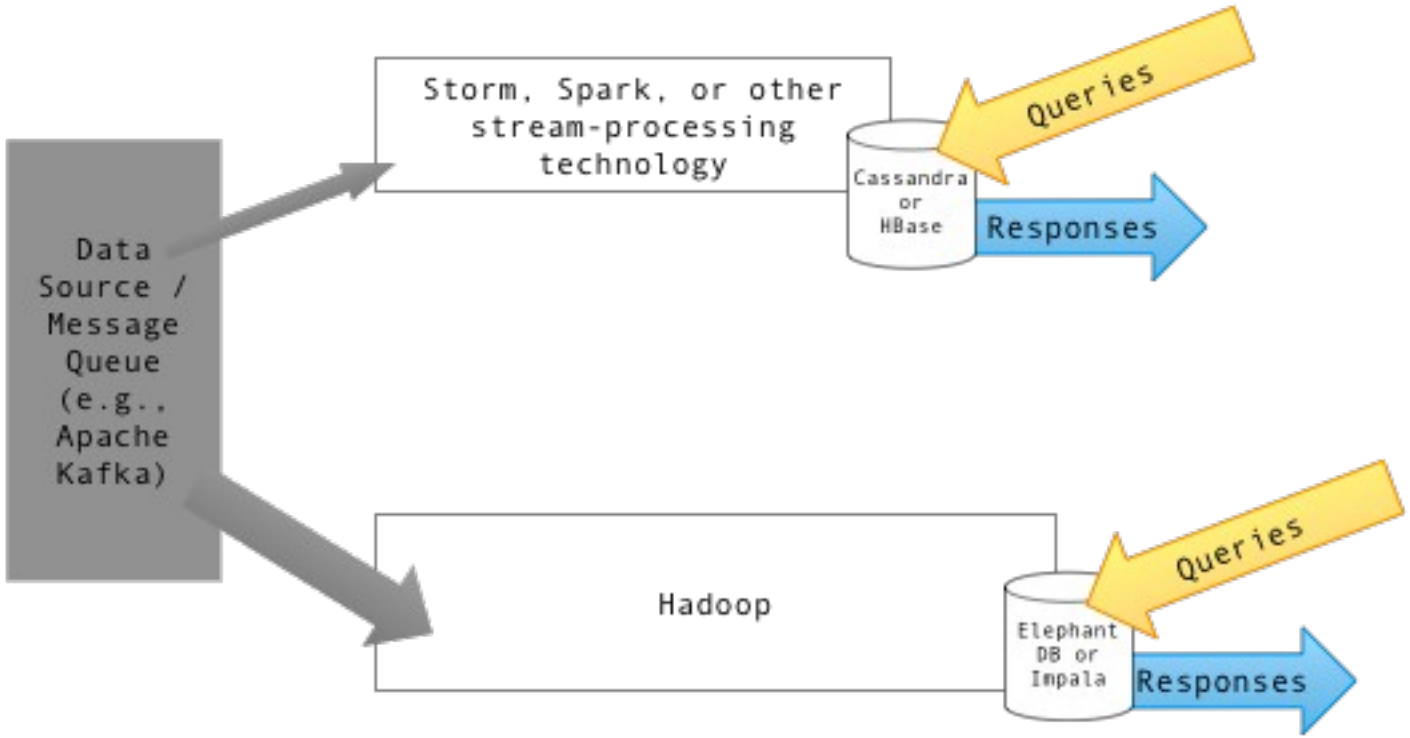
*Tony Baer (Ovum) @ ZDNet - Apache Beam and Spark: New competition for squashing the Lambda Architecture?*

# Realtime Event Processing

Distributed realtime event processing becomes a hot topic:

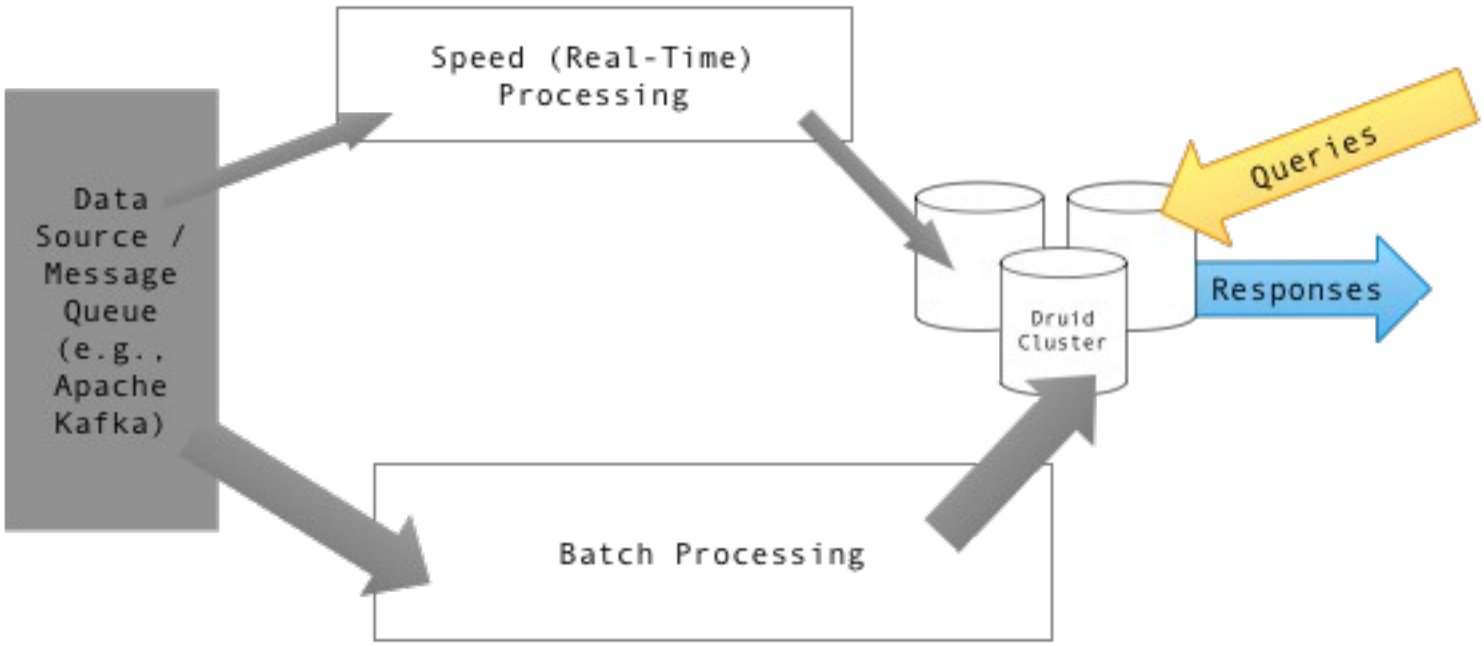
- IoT,
- Service/process monitoring,
- Realtime analytics, fraud detection
- Click stream analytics
- Stock-trading analysis
- Supply chain and transportation alerts
- ...

# 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.

# Lambda Architecture: Projects - I

- Apache Spark is an open-source cluster-computing framework. Spark Streaming, Spark Mllib


- Apache Storm is a distributed stream processing – streams DAG


- Apache Apex™ unified stream and batch processing engine.

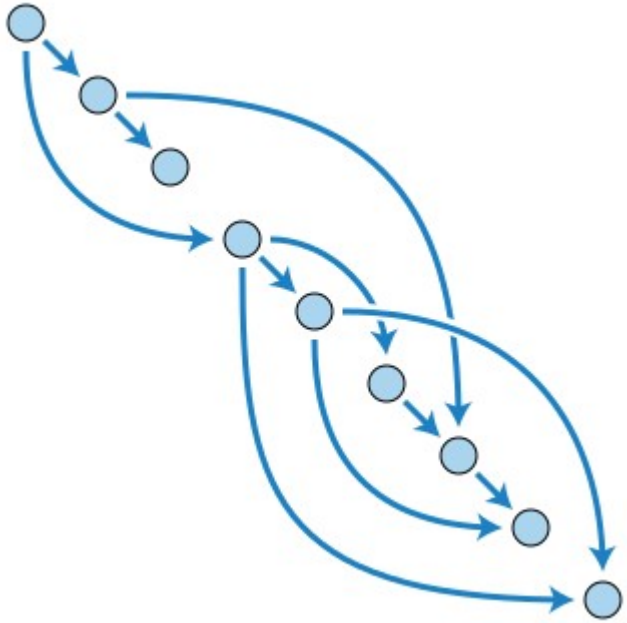
# Lambda Architecture: Projects - II

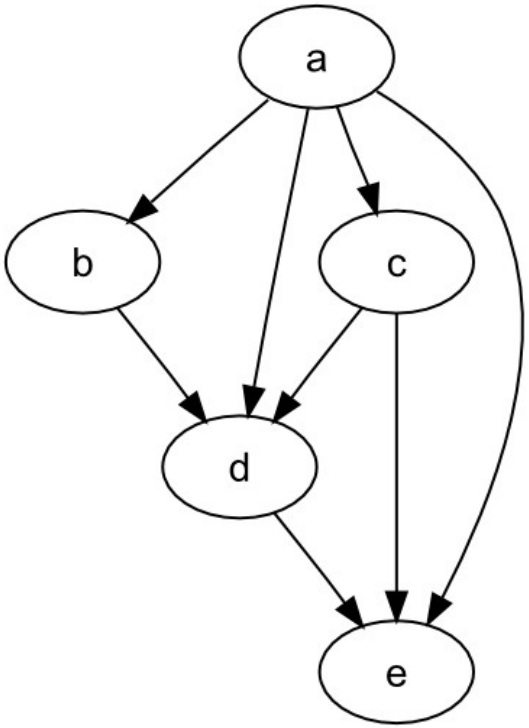
- Apache Flink - open source stream processing framework – Java, Scala

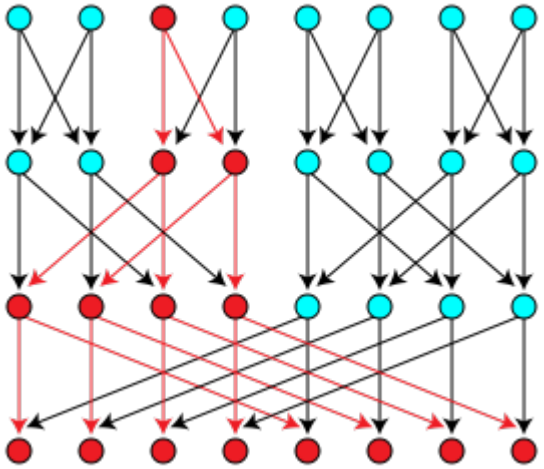
- Apache Kafka - open-source stream processing (Kafka Streams), real-time, low-latency, high-throughput, massively scalable pub/sub

- Apache Beam – unified batch and streaming, portable, extensible

# Direct Acyclic Graphs - DAG

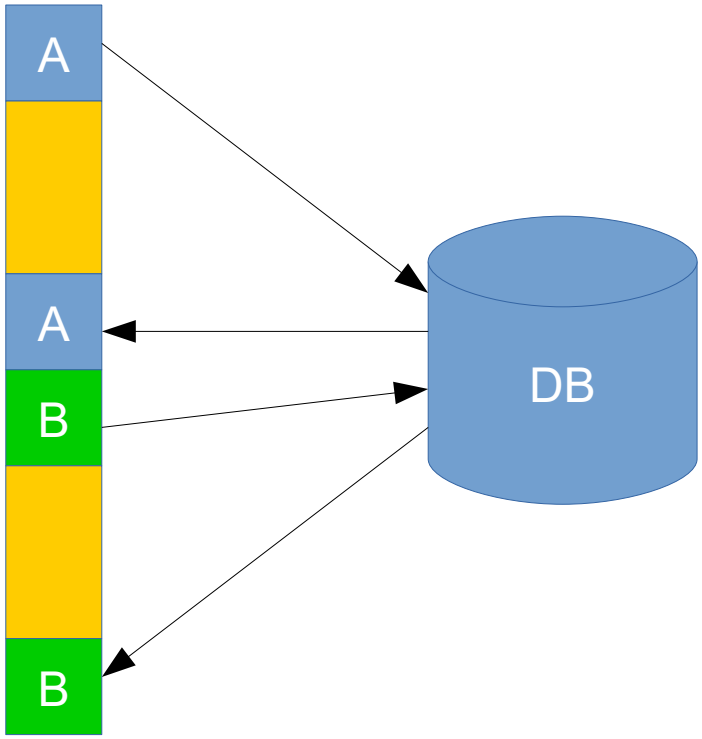




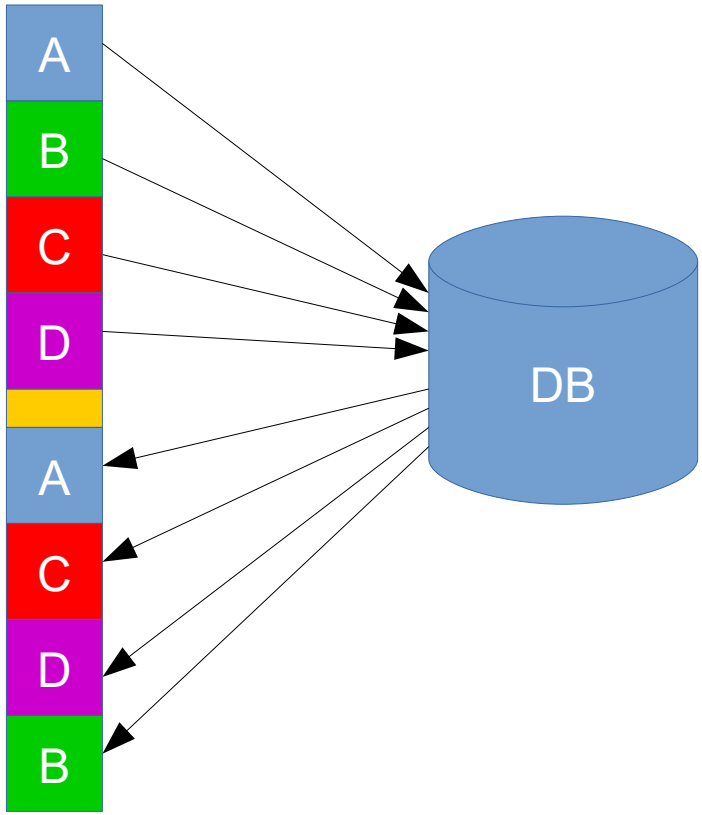



# Synchronous vs. Asynchronous IO

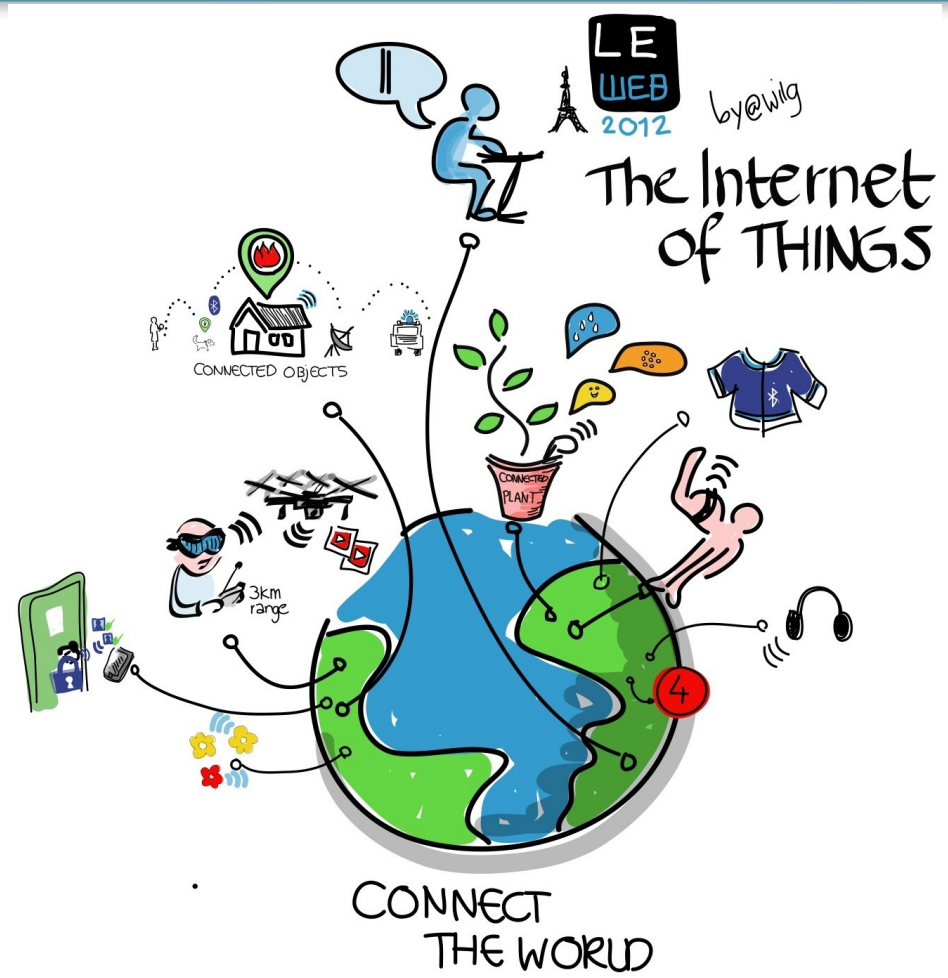
## Synchronous




## Asynchronous



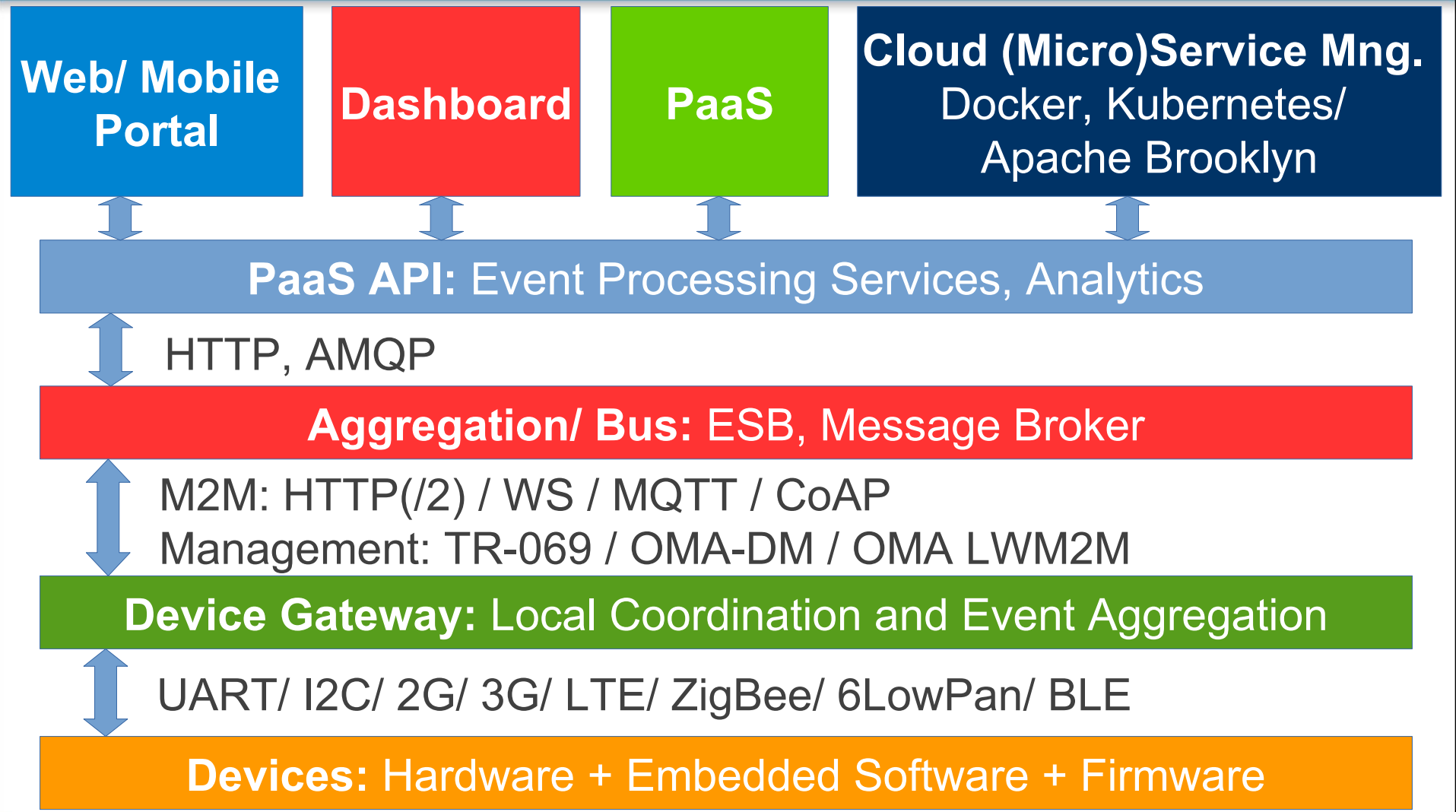
# Example: Internet of Things (IoT)




*CC BY 2.0, Source: https://www.flickr.com/photos/wilgengebroed/8249565455/*


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

# IoT Services Architecture



# 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]

# How to Do Async Programming?

- **Callbacks** – asynchronous methods do not have a return value but take an extra callback parameter (a lambda or anonymous class) that gets called when the result is available. Ex.: Swing's **EventListener**
- **Futures, Promises** – asynchronous methods return a **(Completable)Future<T>** immediately. The value is not immediately available, and the object can be polled Ex.: **Callable<T>** task
- **Reactive Streams (functional, non-blocking)** – **Observable (RxJava), Flowable (RxJava2), Flux & Mono (Project Reactor)**:
  - **Composability** and readability
  - **Data as a flow** manipulated with a rich vocabulary of operators
  - **Lazy evaluation** – nothing happens until you subscribe ()
  - **Backpressure** – consumer can signal to producer that the rate is high
  - **High level but high value abstraction** that is concurrency-agnostic



Source: *https://projectreactor.io/docs/core/release/reference*

# Listing Favs or Suggestions - Reactor

```
userService.getFavorites(userId)
     .timeout(Duration.ofMillis(800))
     .onErrorResume(cacheService.cachedFavoritesFor(userId))
     .flatMap(favoriteService::getDetails)
     .switchIfEmpty(suggestionService.getSuggestions())
     .take(5)
     .publishOn(UiUtils.uiThreadScheduler())
     .subscribe(uiList::show, UiUtils::errorPopup);
});
```

# Comb. Names & Stats – CompletableFuture

```
CompletableFuture<List<String>> ids = findIds();
CompletableFuture<List<String>> result = ids.thenComposeAsync(l -> {
  Stream<CompletableFuture<String>> zip =
    l.stream().map(i -> {
      CompletableFuture<String> nameTask = findName(i);
      CompletableFuture<Integer> statTask = findStat(i);
      return nameTask.thenCombineAsync(statTask,
        (name, stat) -> "Name " + name + " has stats " + stat); });
    List<CompletableFuture<String>> combineList =
      zip.collect(Collectors.toList());
    CompletableFuture<String>[] combineArray =
      combineList.toArray(new CompletableFuture[combineList.size()]);
    CompletableFuture<Void> allDone =
      CompletableFuture.allOf(combineArray);
    return allDone.thenApply(v -> combineList.stream()
        .map(CompletableFuture::join)
        .collect(Collectors.toList()));
});
List<String> results = result.join();
```

# Combined Names & Stats – Reactor

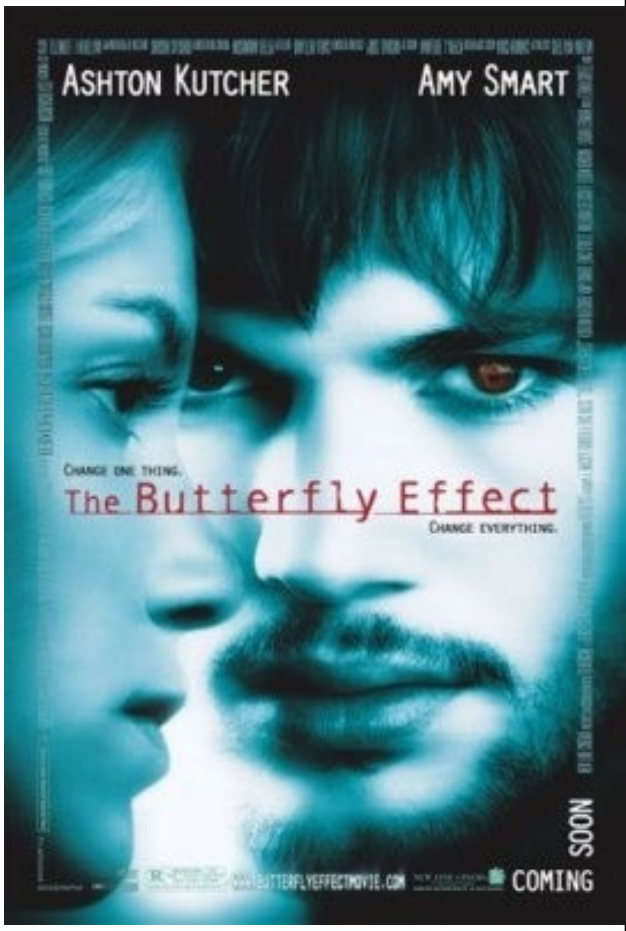
```
Flux<String> ids = findIds();

    Flux<String> combinations =
        ids.flatMap(id -> {
            Mono<String> nameTask = findName(id);
            Mono<Integer> statTask = findStat(id);

            return nameTask.zipWith(statTask,
                (name, stat) -> "Name " + name + " has stats " + stat);
        });

    Mono<List<String>> result = combinations.collectList();

    List<String> results = result.block();
}
```

# 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.**






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

# 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 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/]

# 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));
```

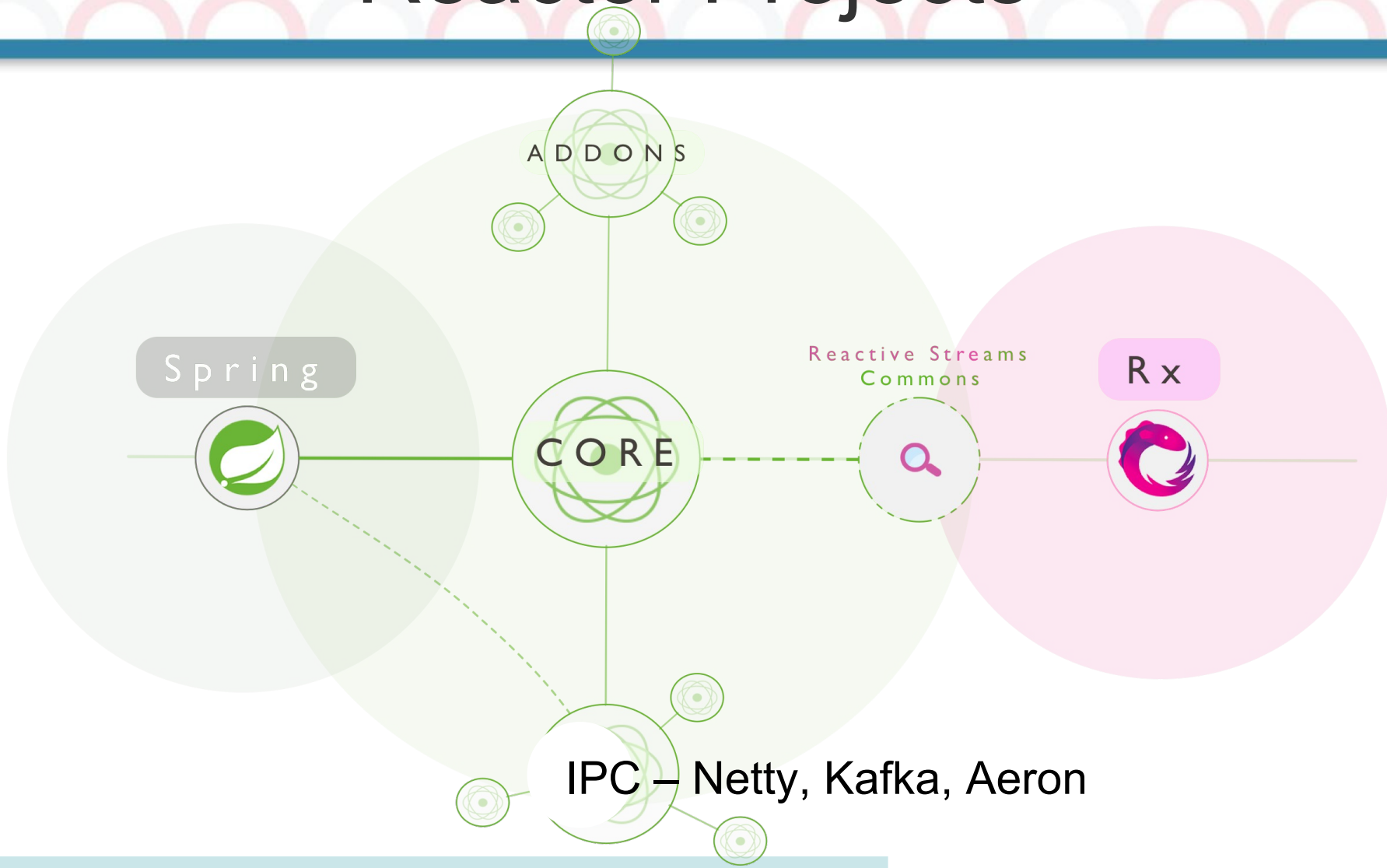
*Result:*

```
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

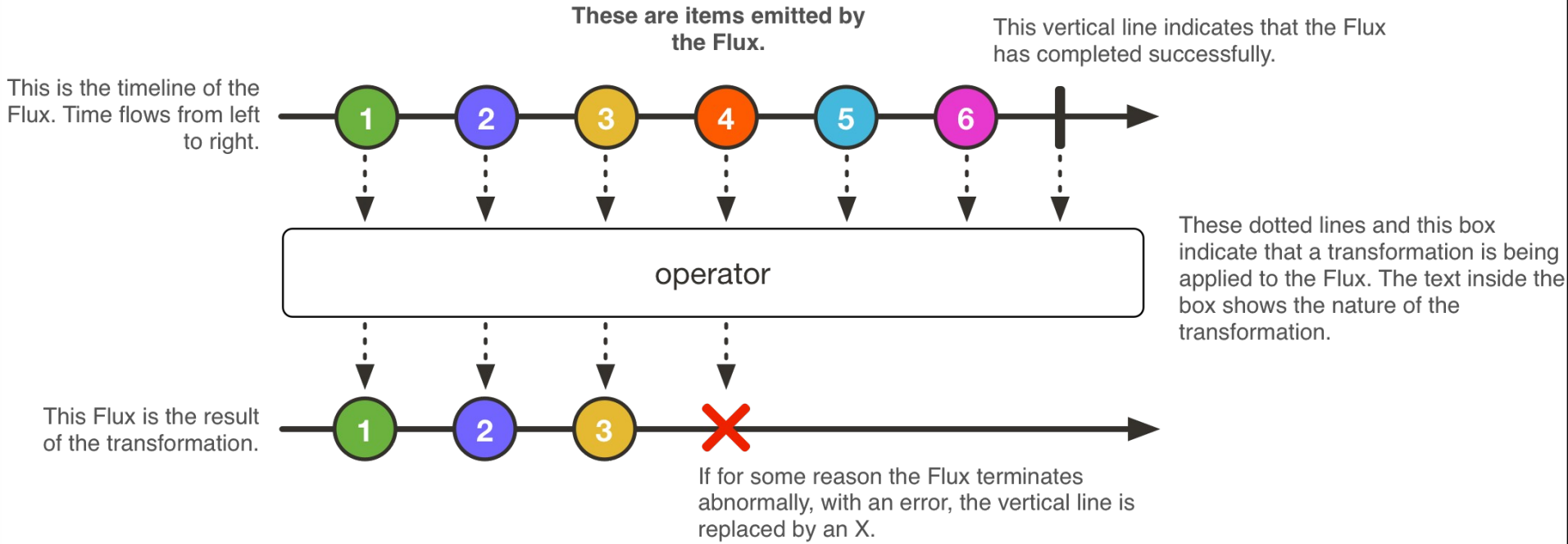
- Reactor project allows building high-performance (low latency high throughput) non-blocking asynchronous applications on JVM.
- Reactor is designed to be extraordinarily fast and can sustain throughput rates on order of 10's of millions of operations per second.
- Reactor has powerful API for declaring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

# Reactor Projects

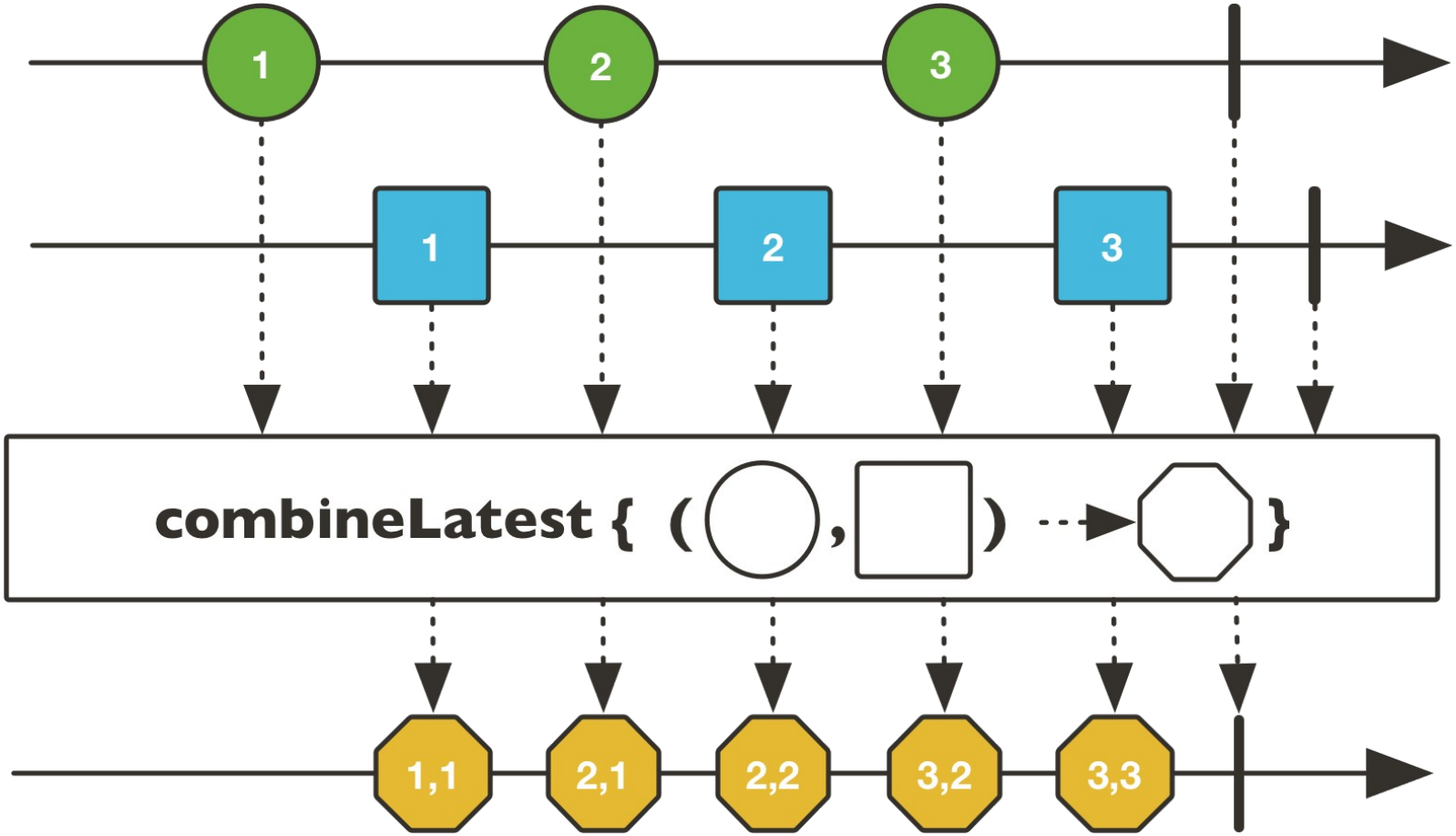



*https://github.com/reactor/reactor, Apache Software License 2.0*

# Reactor Flux

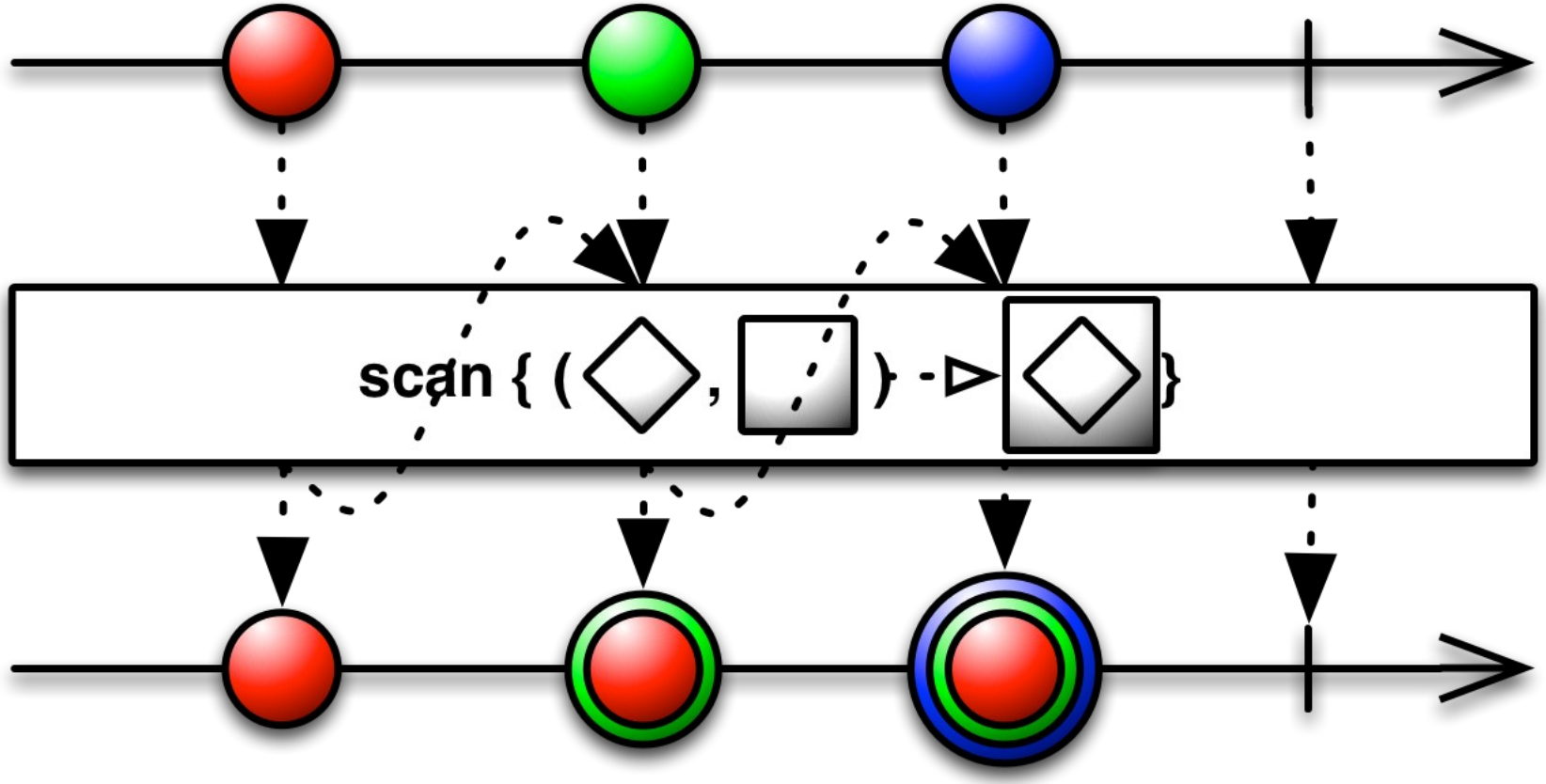


# Example: Flux.combineLatest()




*https://projectreactor.io/core/docs/api/, Apache Software License 2.0*

# Redux == Rx Scan Opearator

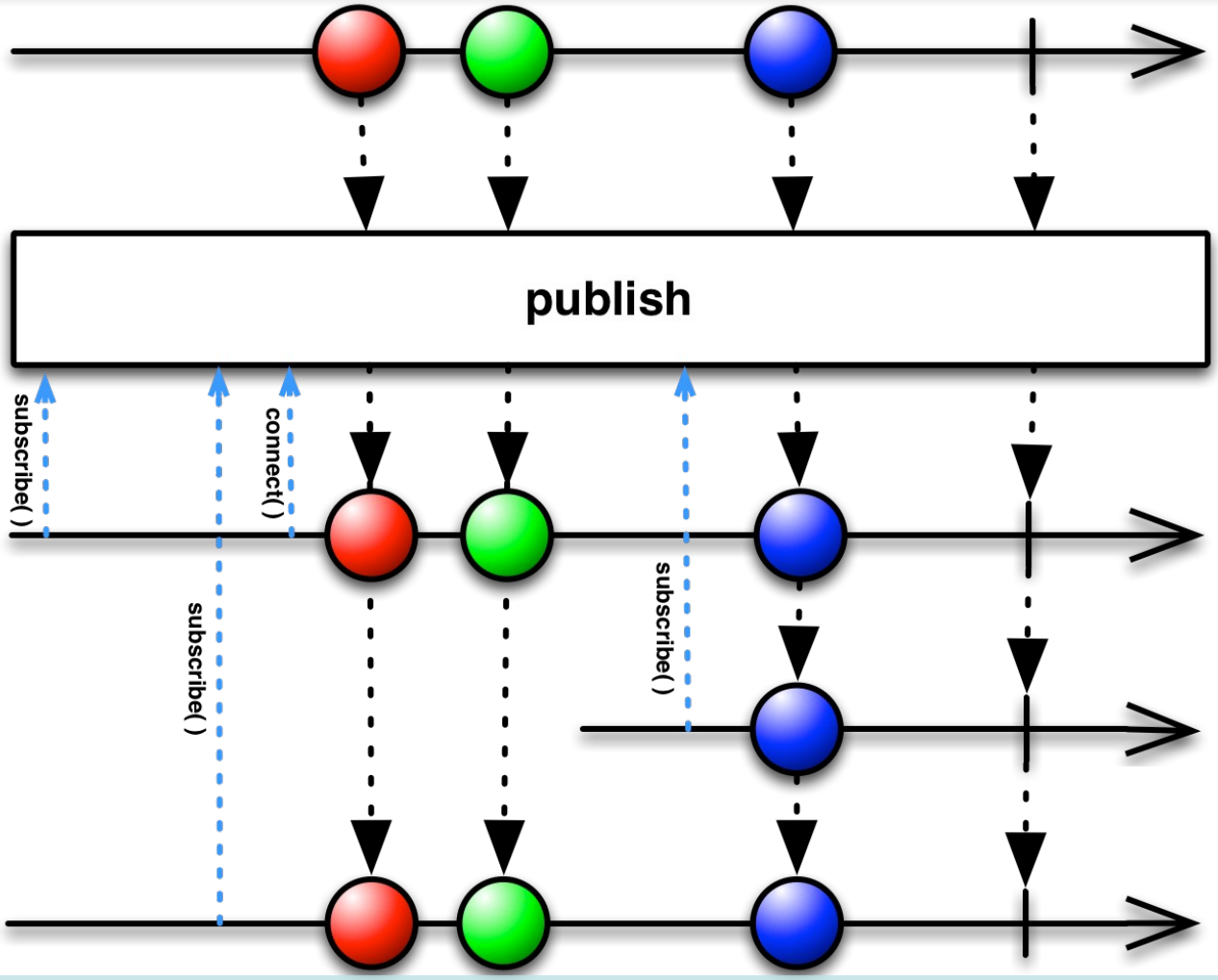



*Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/*

# 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) – emit values independent of individual subscriptions. They have their own timeline and events occur whether someone is listening or not. When subscription is made observer receives current events as they happen.
  *Example:* mouse events

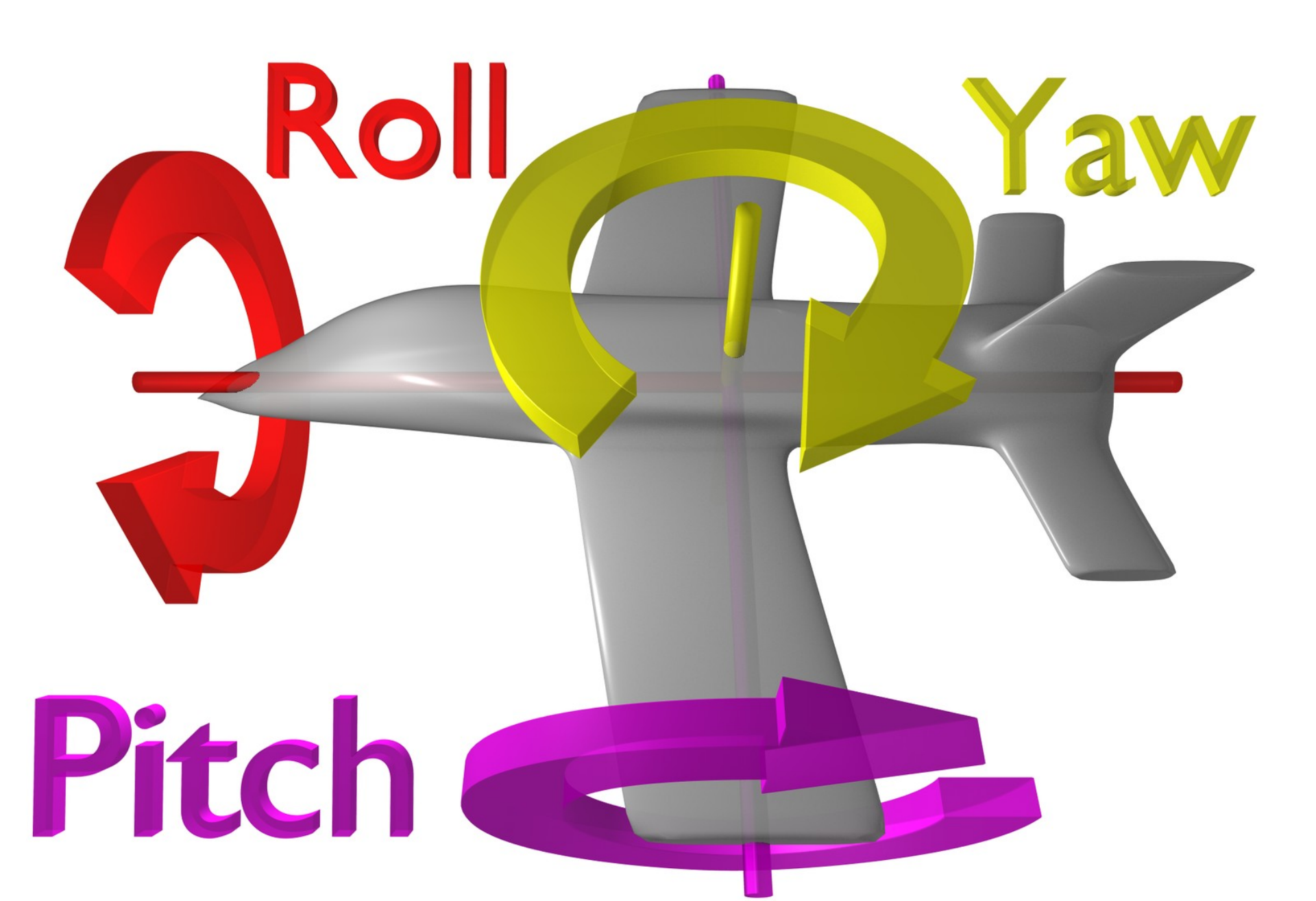
# Converting Cold to Hot Stream



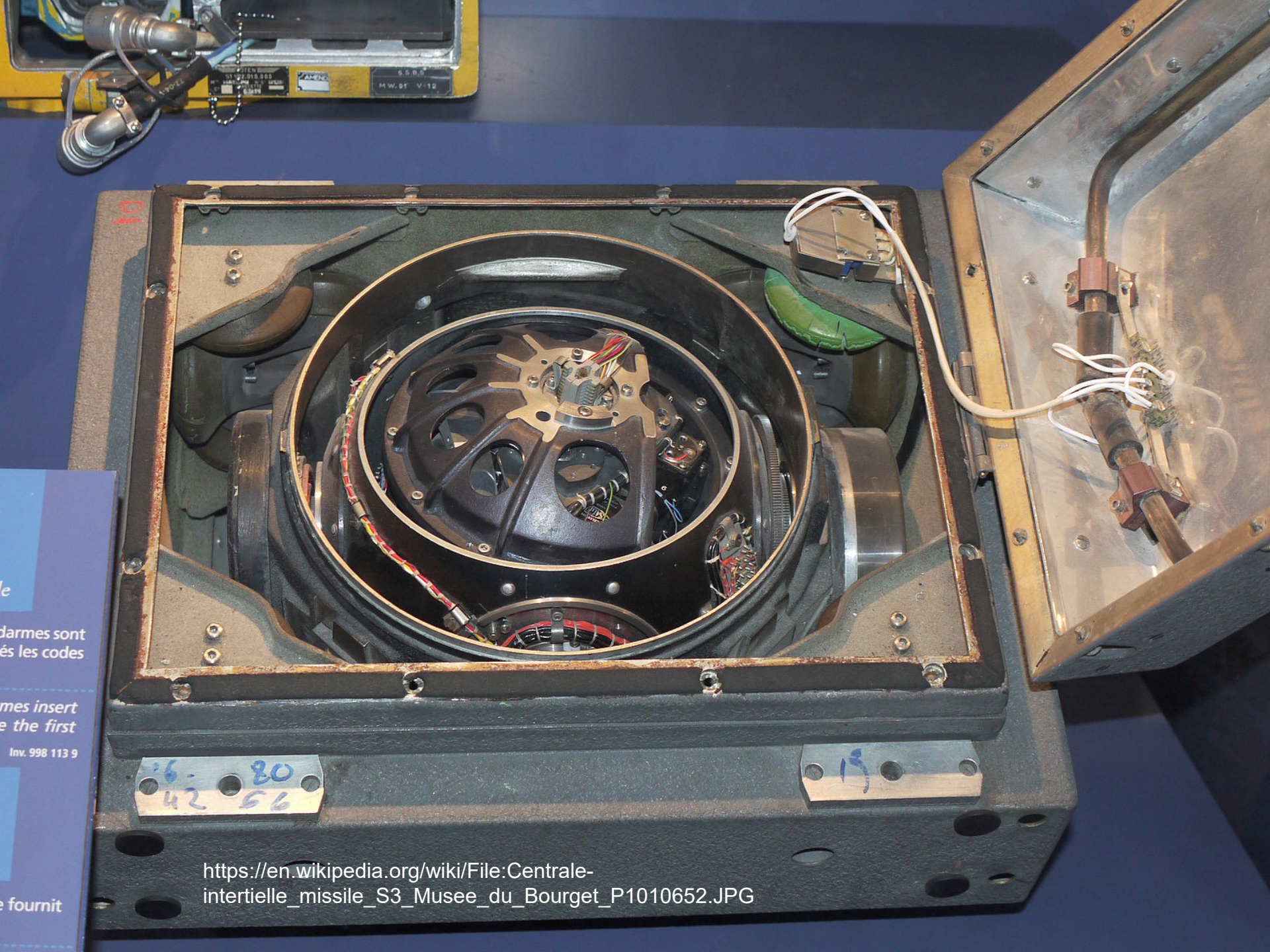

*Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/*

# Hot Stream Example - Reactor

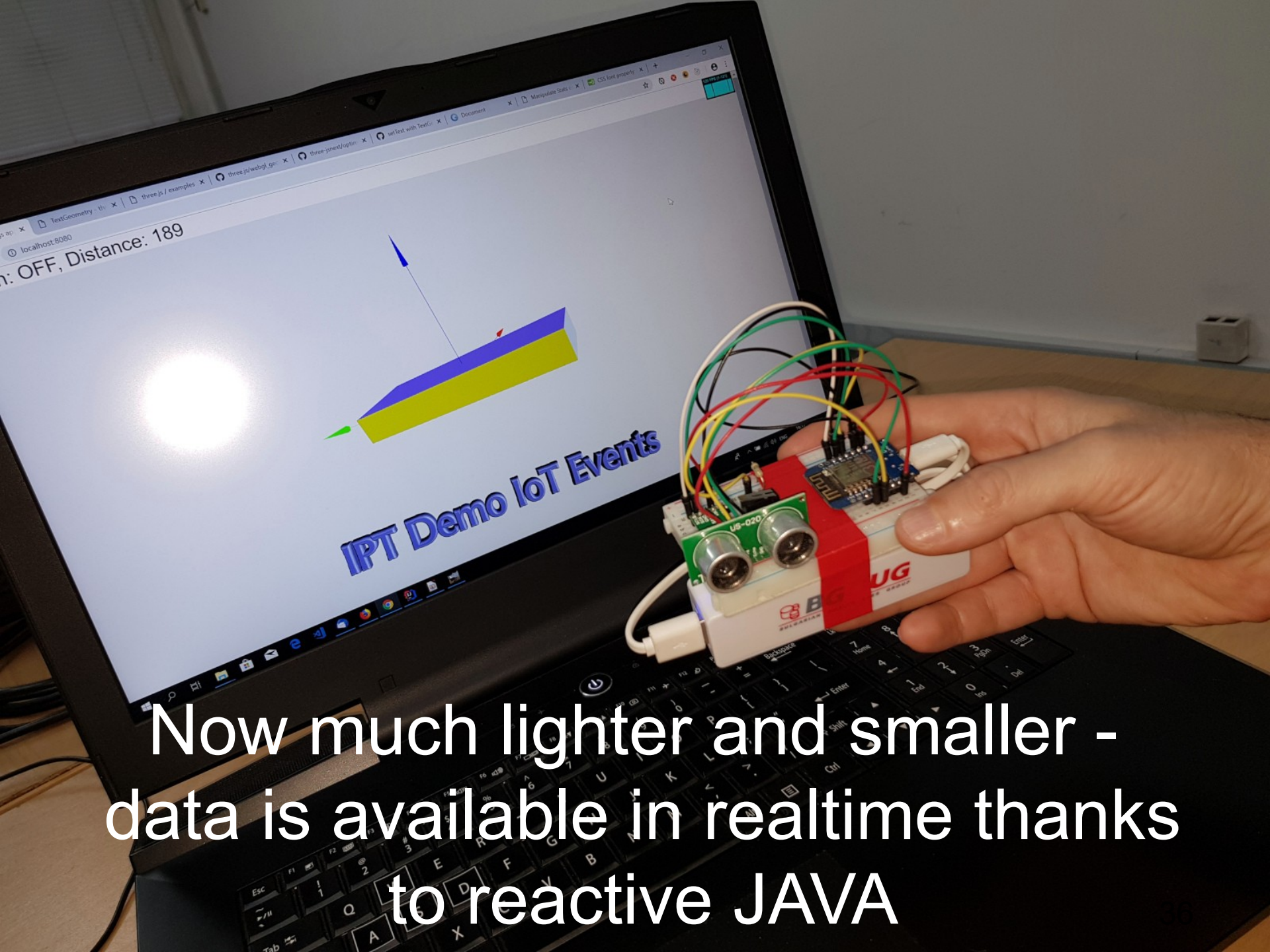
```
EmitterProcessor<String> emitter =
        EmitterProcessor.create();
FluxSink<String> sink = emitter.sink();
emitter.publishOn(Schedulers.single())
        .map(String::toUpperCase)
        .filter(s -> s.startsWith("HELLO"))
        .delayElements(Duration.of(1000, MILLIS))
        .subscribe(System.out::println);
sink.next("Hello World!"); // emit - non blocking
sink.next("Goodbye World!");
sink.next("Hello Trayan!");
Thread.sleep(3000);
```




https://en.wikipedia.org/wiki/File:Flight_dynamics_with_text.png

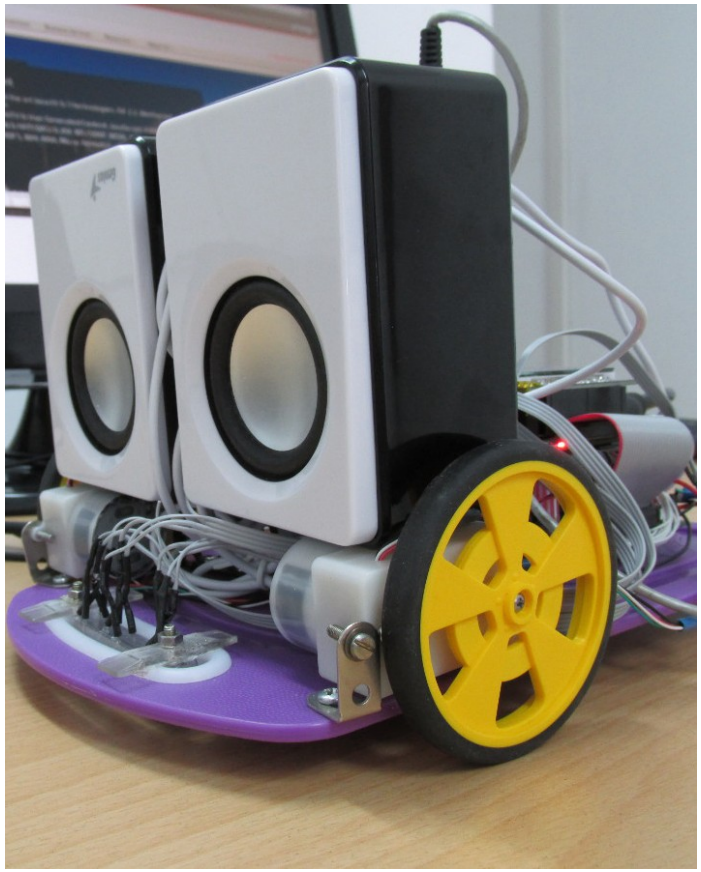


https://en.wikipedia.org/wiki/File:Centrale-intertielle_missile_S3_Musee_du_Bourget_P1010652.JPG



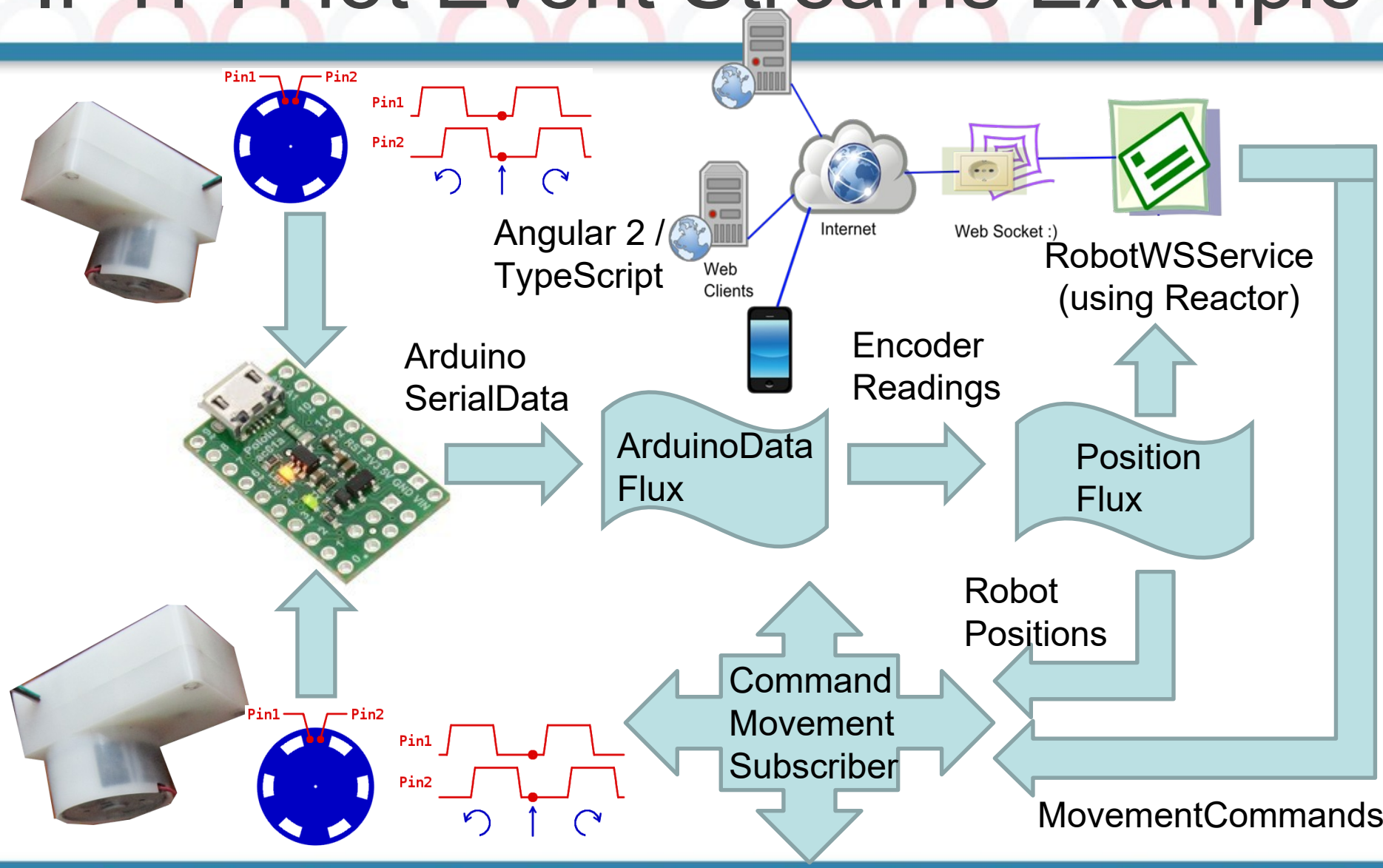

Now much lighter and smaller - data is available in realtime thanks to reactive JAVA

# 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



# Thank's for Your Attention!

**Trayan Iliev**

**http://robolearn.org/**

**https://github.com/iproduct**

**https://twitter.com/trayaniliev**

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