Introductuion to High Performance Reactive Architectures & Spring WebFlux

About me

Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies** http://www.iproduct.org

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

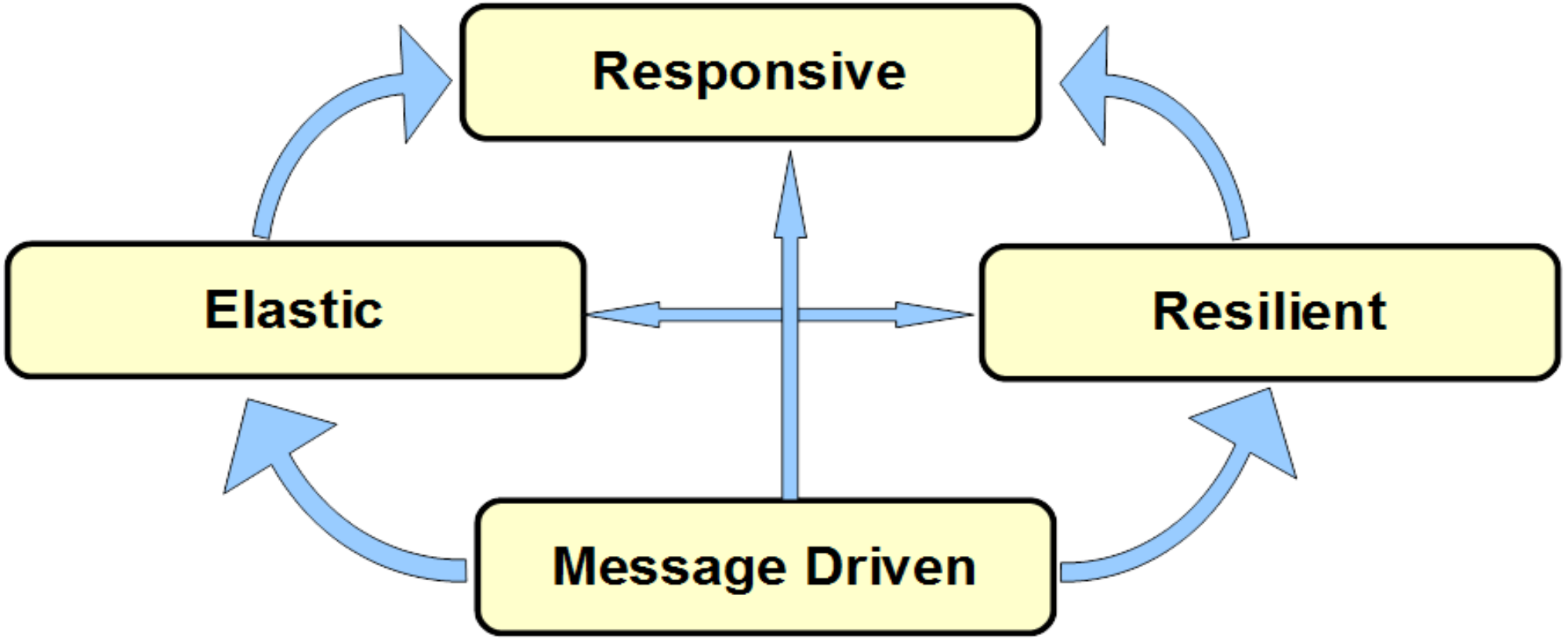
– Organizer RoboLearn hackathons and IoT enthusiast

Where to Find The Code and Materials?

https://github.com/iproduct/course-kotlin

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**

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]

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

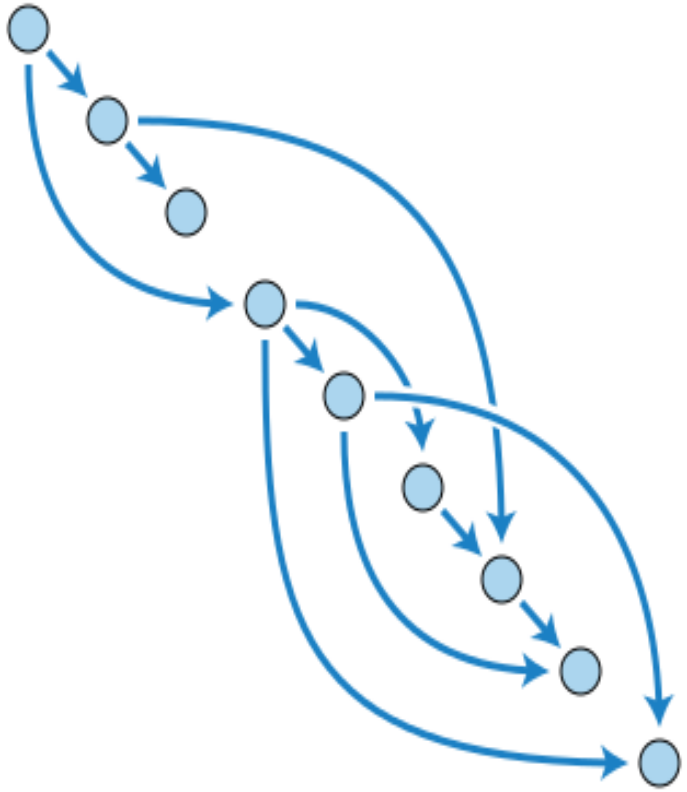
Data Stream Programming

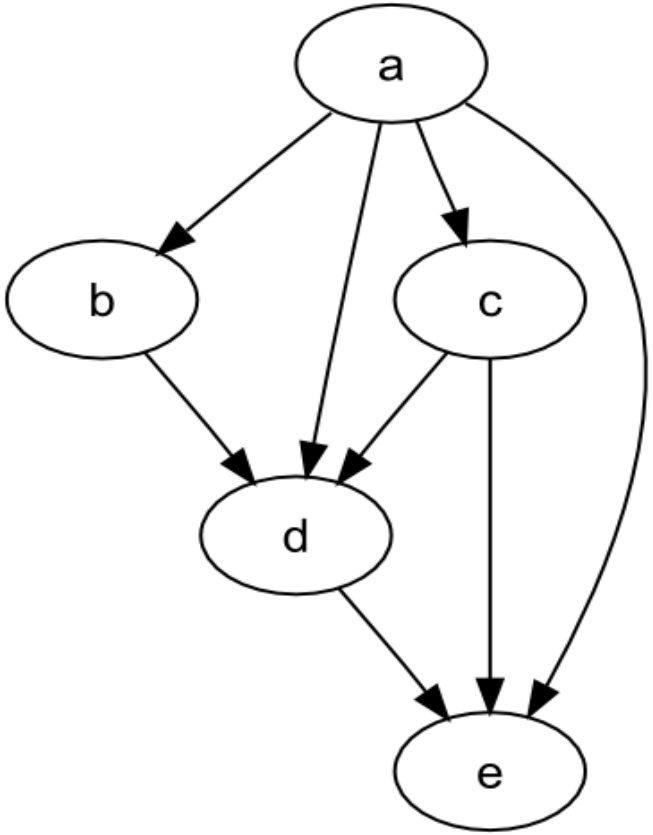
The idea of abstracting logic from execution is hardly new -- it was the dream of SOA. And the recent emergence of microservices and containers shows that the dream still lives on.

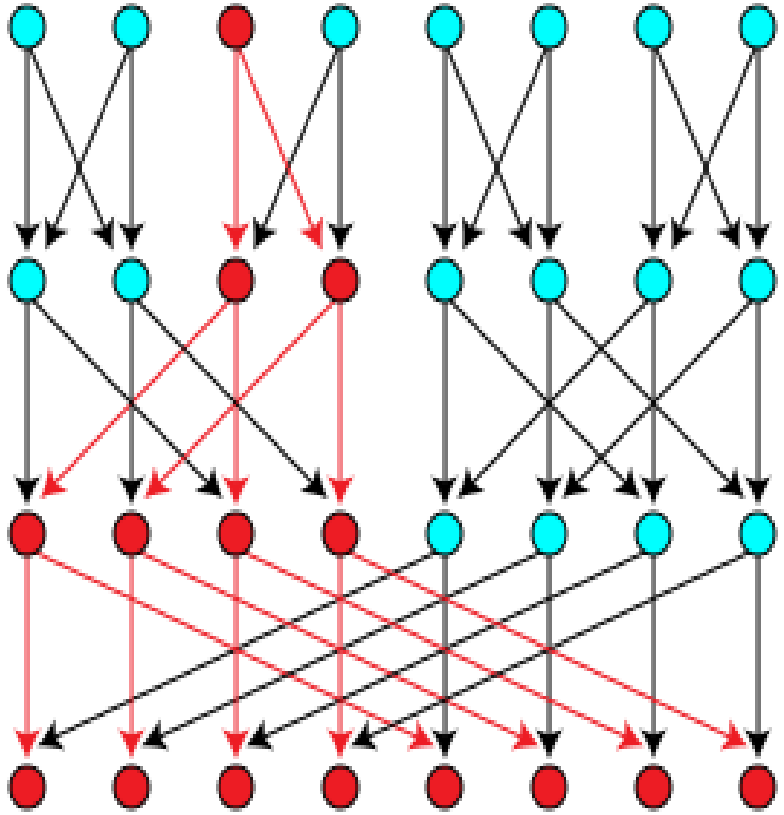
For developers, the question is whether they want to learn yet one more layer of abstraction to their coding. On one hand, there's the elusive promise of a common API to streaming engines that in theory should let you mix and match, or swap in and swap out.

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

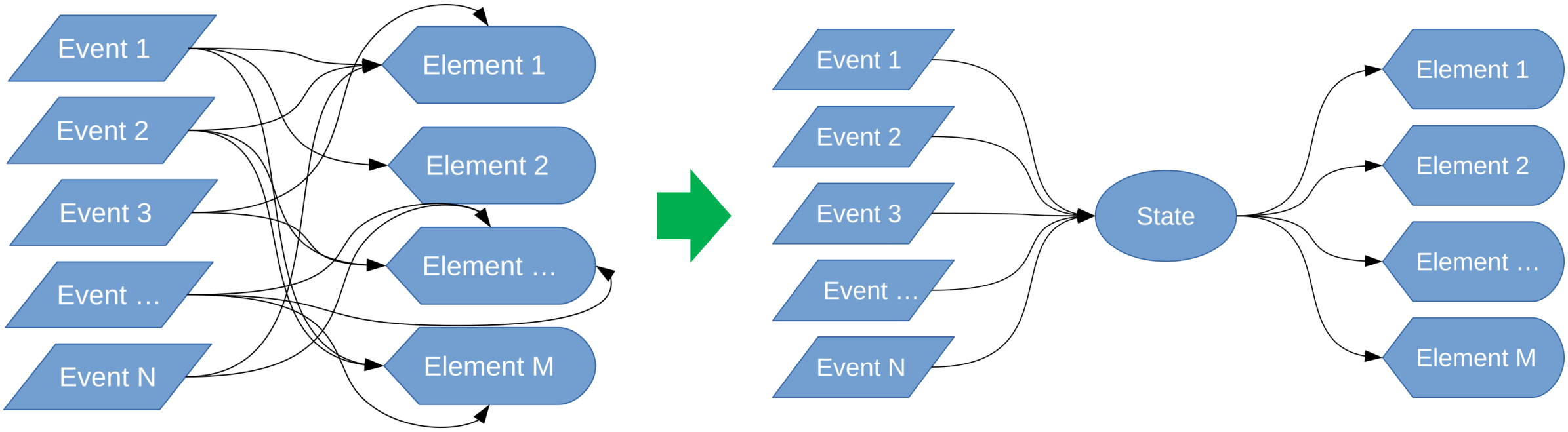
Direct Acyclic Graphs - DAG





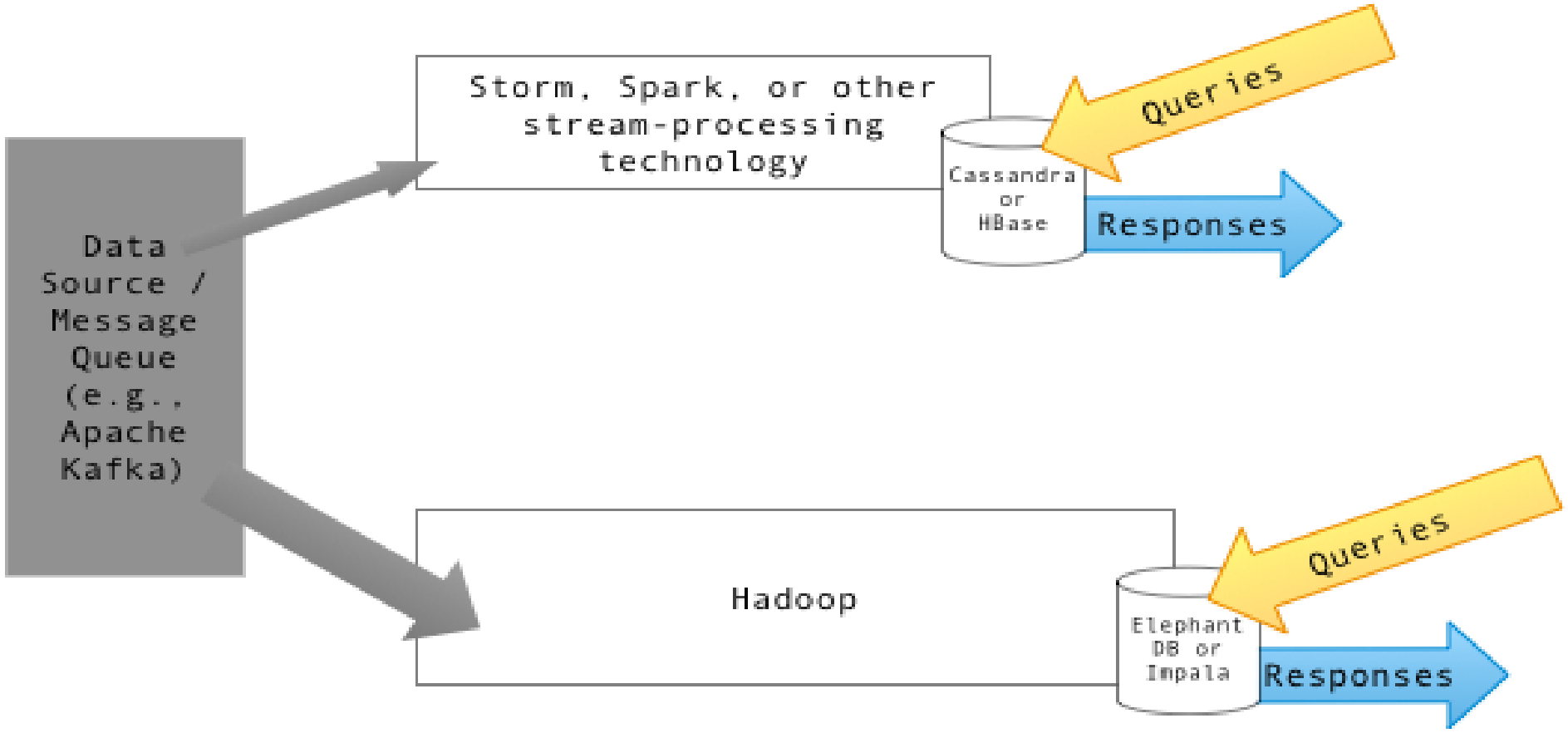


Event Sourcing – Events vs. Sate (Snapshots)



Lambda Architecture - I

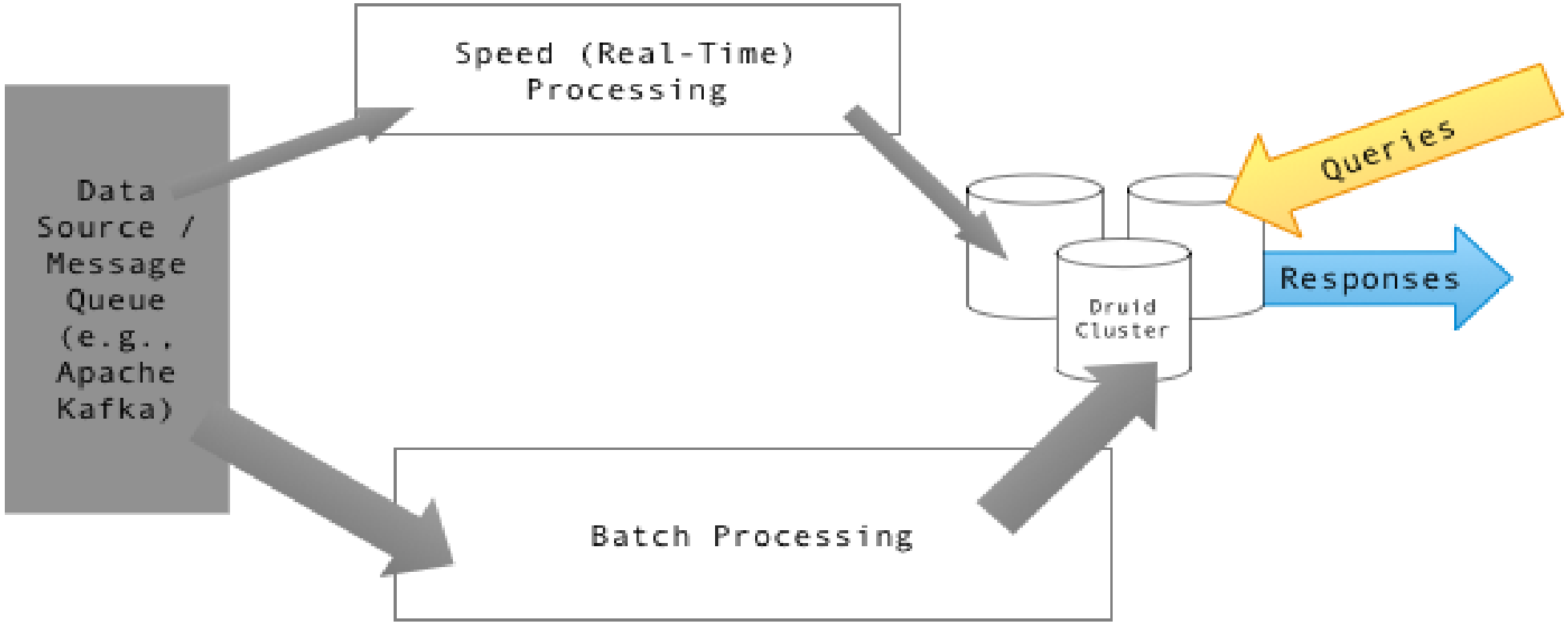
Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)



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

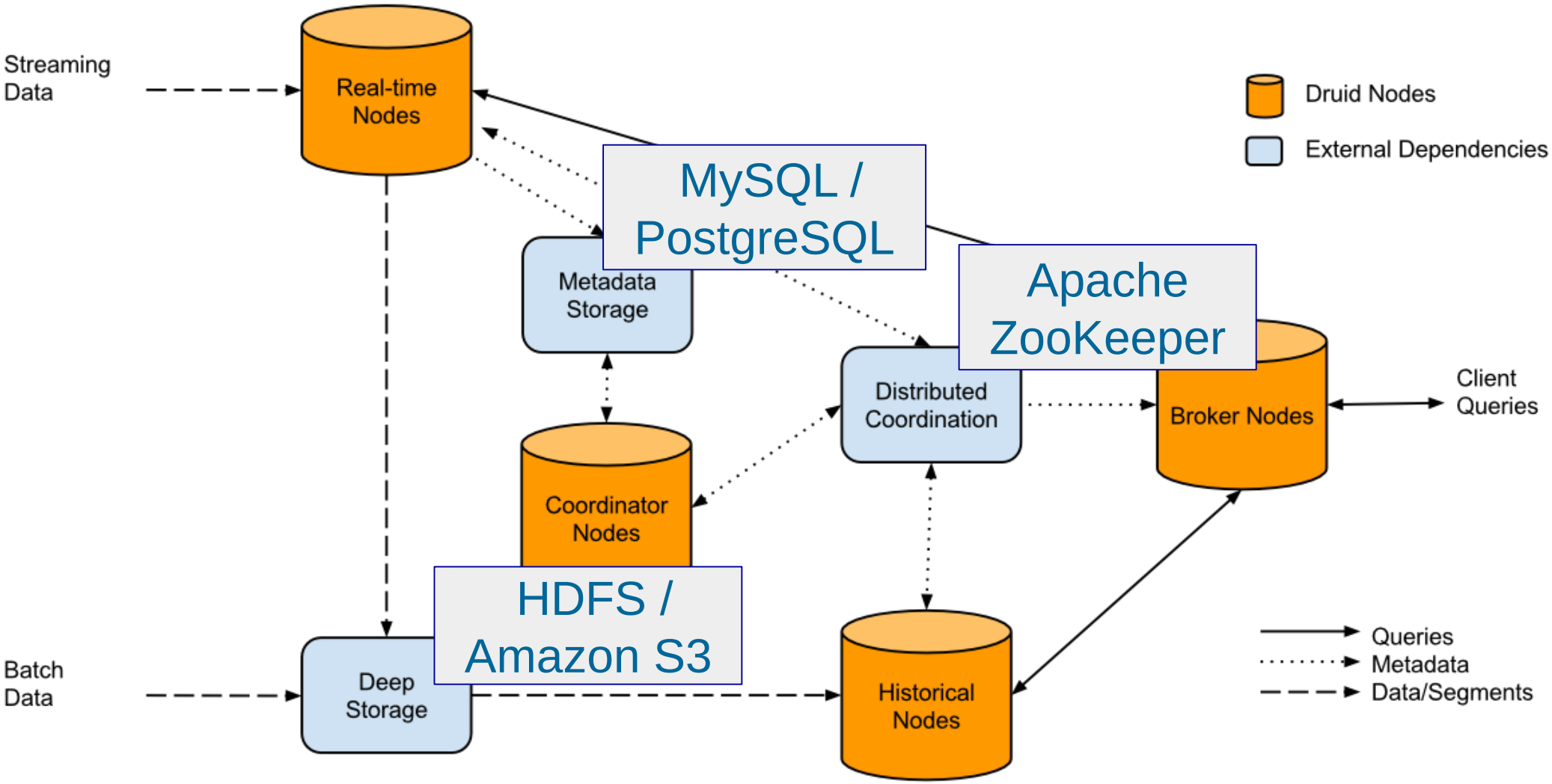
Lambda Architecture - II

Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)



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

Lambda Architecture - Druid Distributed Data Store



https://commons.wikimedia.org/w/index.php?curid=33899448 By Fangjin Yang, GFDL

Kappa Architecture

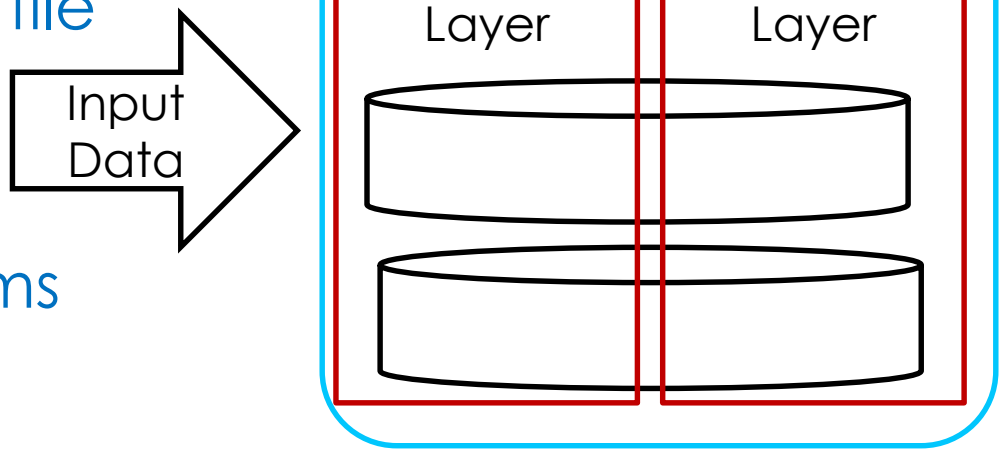
Query = K (New Data) = K (Live streaming data)

- Proposed by Jay Kreps in 2014
- Real-time processing of distinct events
- Drawbacks of Lambda architecture:
  - It can result in coding overhead due to comprehensive processing
  - Re-processes every batch cycle which may not be always beneficial
  - Lambda architecture modeled data can be difficult to migrate
- Canonical data store in a Kappa Architecture system is an append-only immutable log (like Kafka, Pulsar)

Kappa Architecture II

Query = K (New Data) = K (Live streaming data)

- Multiple data events or queries are logged in a queue to be catered against a distributed file system storage or history.
- The order of the events and queries is not predetermined. Stream processing platforms can interact with database at any time.
- It is resilient and highly available as handling terabytes of storage is required for each node of the system to support replication.
- Machine learning is done on the real time basis



Zeta Architecture

- Main characteristics of Zeta architecture:
  - file system (HDFS, S3, GoogleFS),
  - realtime data storage (HBase, Spanner, BigTable),
  - modular processing model and platform (MapReduce, Spark, Drill, BigQuery),
  - containerization and deployment (cgroups, Docker, Kubernetes),
  - Software solution architecture (serverless computing – e.g. Amazon Lambda)
- Recommender systems and machine learning
- Business applications and dynamic global resource management (Mesos + Myriad, YARN, Diego, Borg).

Distributed Stream Processing – Apache Projects:

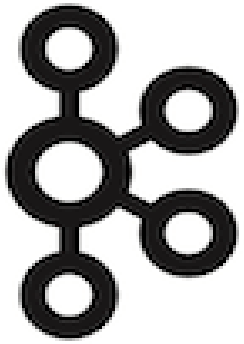
- Apache Spark is an open-source cluster-computing framework. Spark Streaming, Spark Mllib

- Apache Storm is a distributed stream processing – streams DAG

- Apache Samza is a distributed real-time stream processing framework.

Distributed Stream Processing – Apache 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



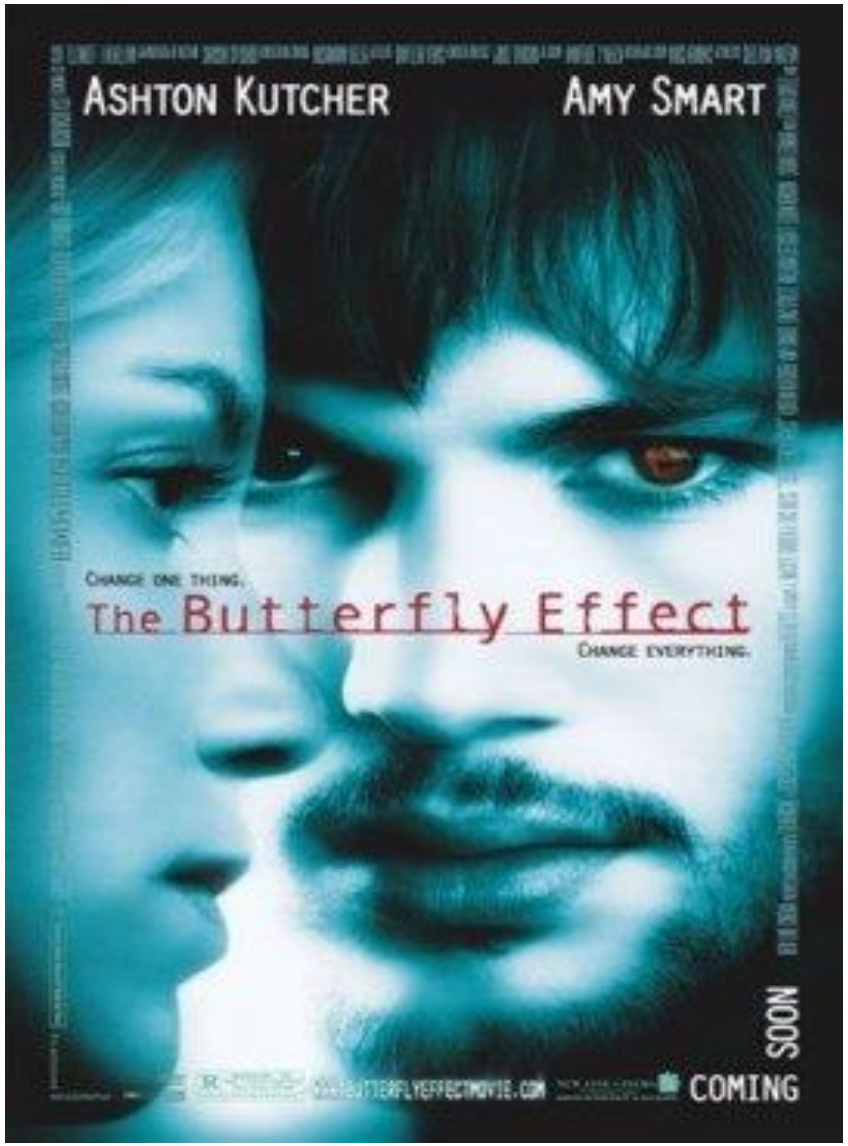
Functional Reactive Programming

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 vs. 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.

Ex. (RxGo):

```
observable := rxgo.Just("Hello", "Reactive", "World", "from", "RxGo")().
        Map(ToUpper). // map to upper case
        Filter(LengthGreaterThan4) // greaterThan4 func filters values > 4
for item := range observable.Observe() {
        fmt.Println(item.V)
}
```

Functional Reactive Programming (FRP)

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

  (a) Denotative

  (b) Temporally continuous

- FRP is asynchronous data-flow programming using the building blocks of functional programming (map, reduce, filter, etc.) and explicitly modeling time

Reactive Programming

- ReactiveX (Reactive Extensions) - open source polyglot (http://reactivex.io):

  Rx = Observables + Flow transformations + Schedulers

- Go: RxGo, Kotlin: RxKotlin, Java: RxJava, JavaScript: RxJS, Python: RxPY, C#: Rx.NET, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Ruby: Rx.rb, Python: RxPY, Groovy: RxGroovy, JRuby: RxJRuby, ...
- Reactive Streams Specification (http://www.reactive-streams.org/):
  - Publisher – provider of potentially unbounded number of sequenced elements, according to Subscriber(s) demand (backpressure): onNext* (onError | onComplete)
  - Subscriber, Subscription
  - Processor = Subscriber + Publisher

ReactiveX: Observable vs. Iterable

Example code showing how similar high-order functions can be applied to an Iterable and an Observable

`Iterable`

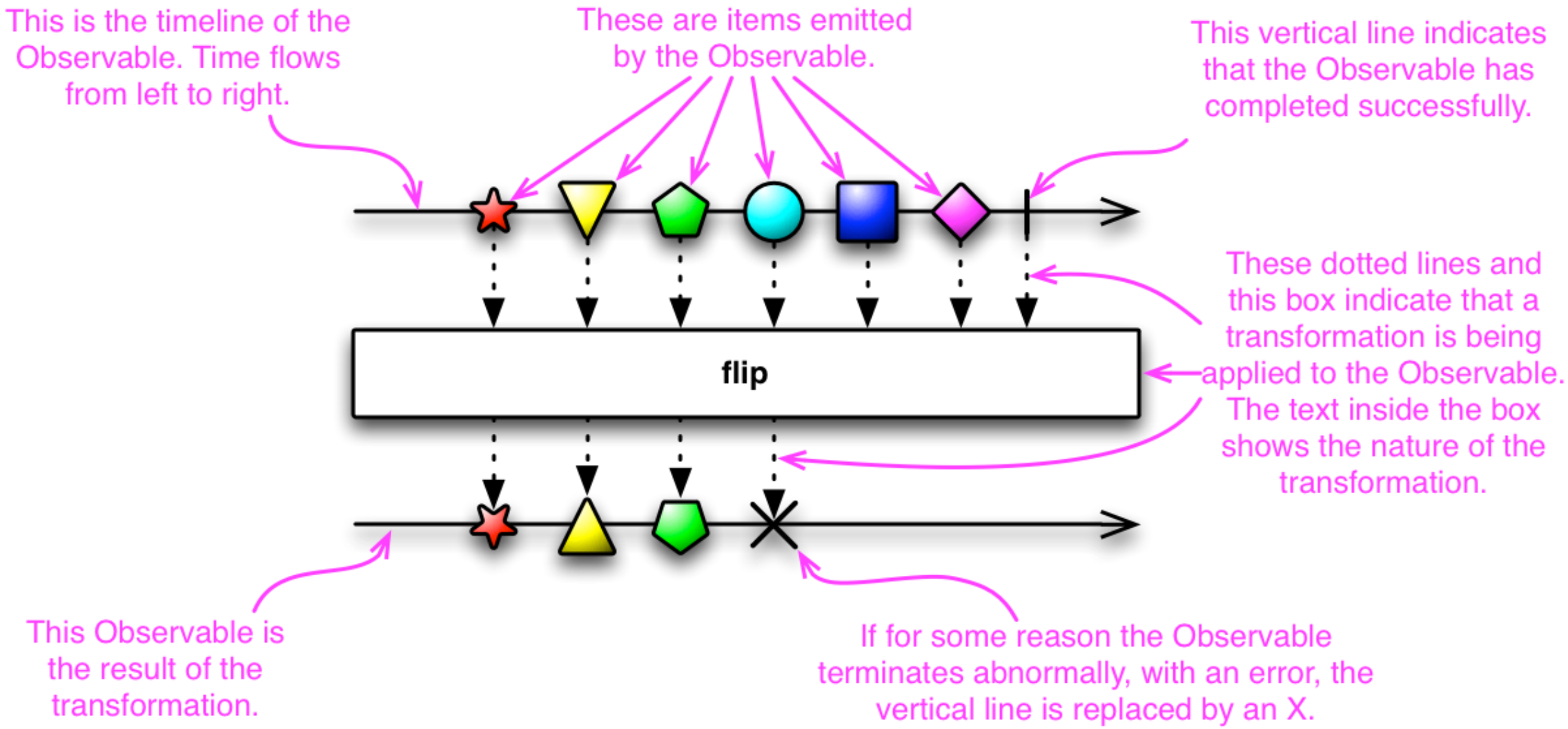
```
getDataFromLocalMemory()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transforme
d" })
  .forEach({ println "next => " + it
})
```

`Observable`

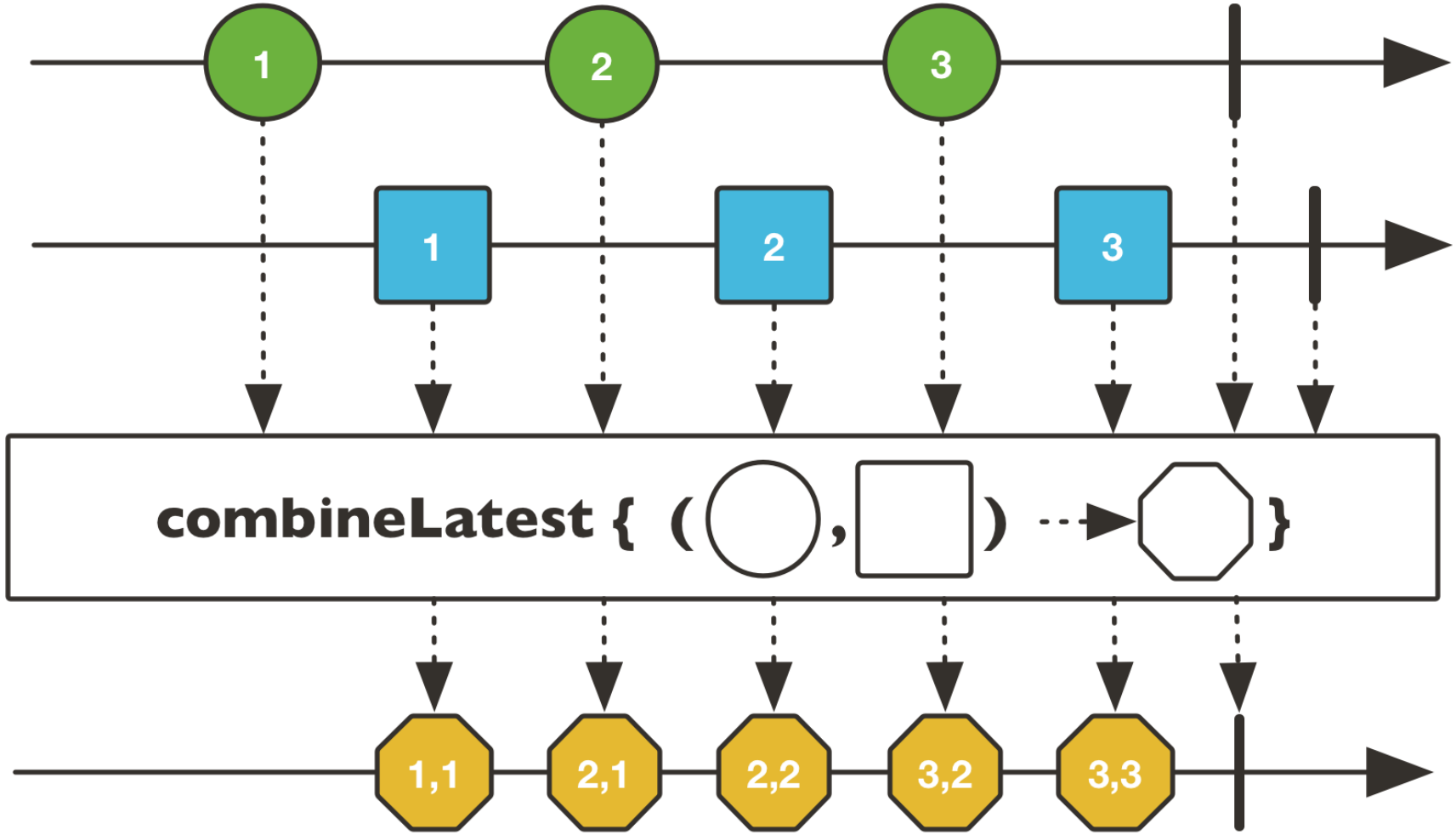
```
getDataFromNetwork()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed"
})
  .subscribe({ println "onNext => " + it
})
```

You can think of the Observable class as a **"push"** equivalent to Iterable, which is a **"pull."** With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive. By contrast, with an Observable the producer pushes values to the consumer whenever values are available. This approach is more flexible, because values can arrive synchronously or asynchronously.

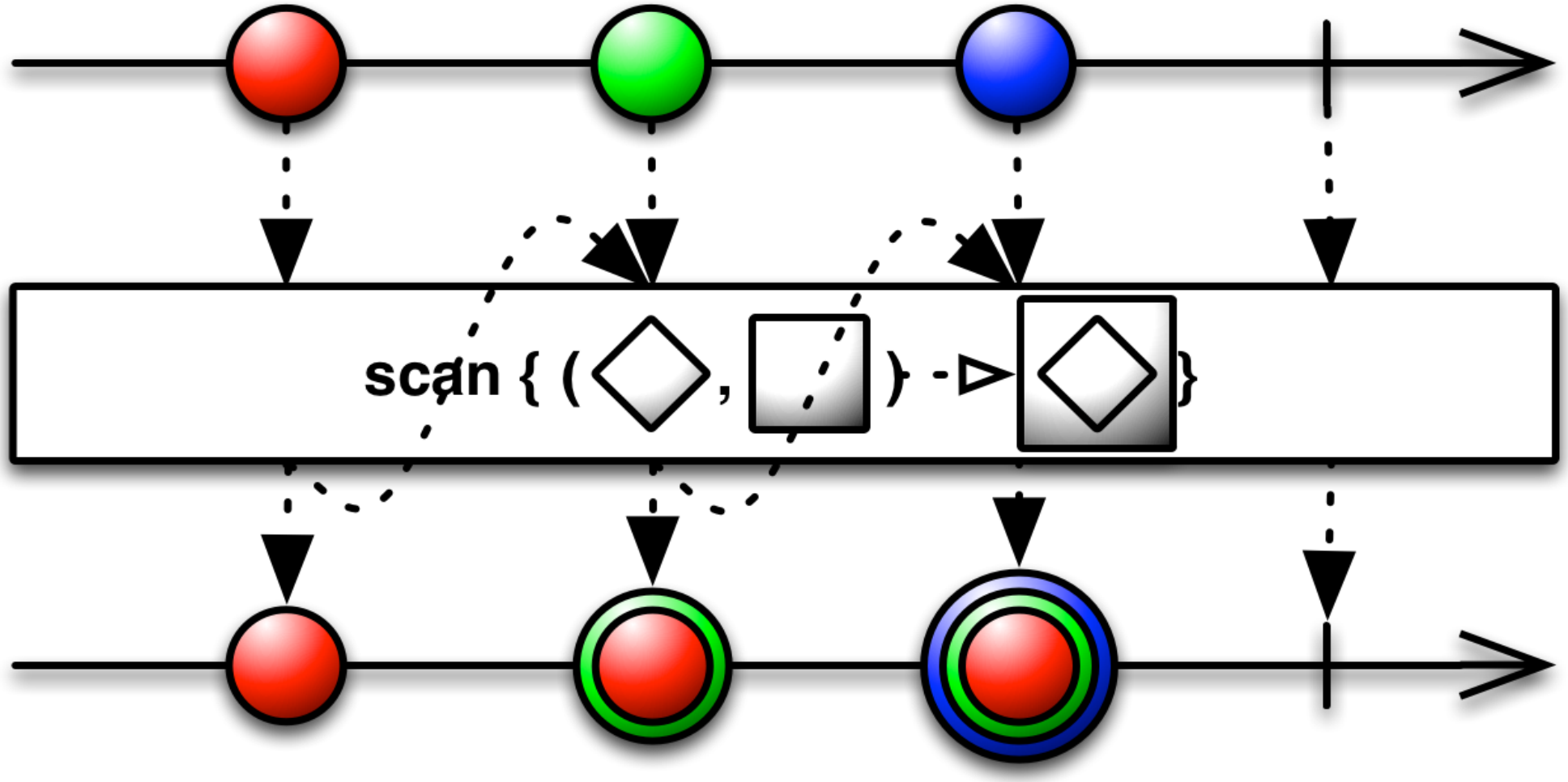
ReactiveX Observable – Marble Diagrams



Example: CombineLatest



Redux == Rx Scan Opearator



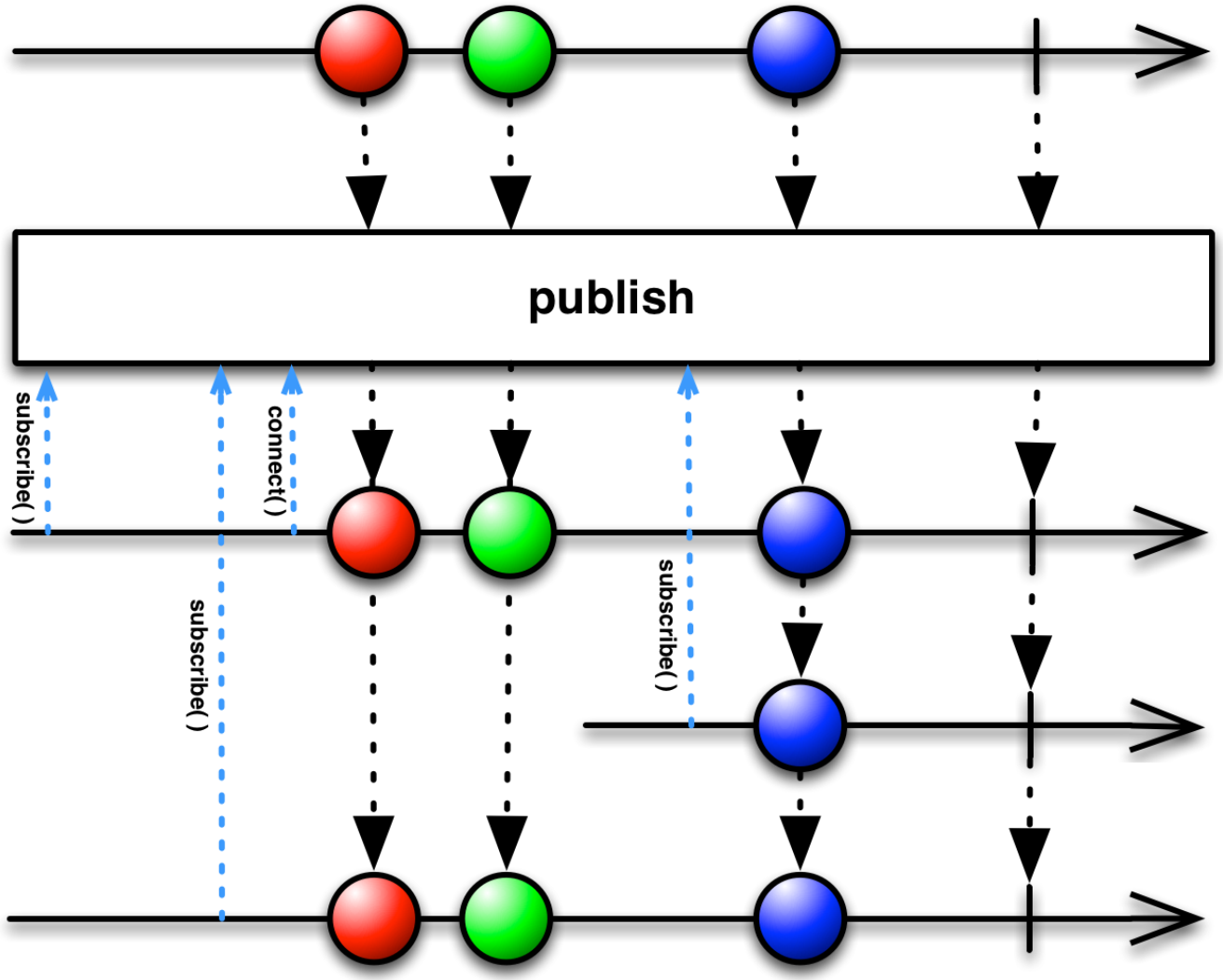
Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/

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

55

Converting Cold to Hot Stream



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

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));
```

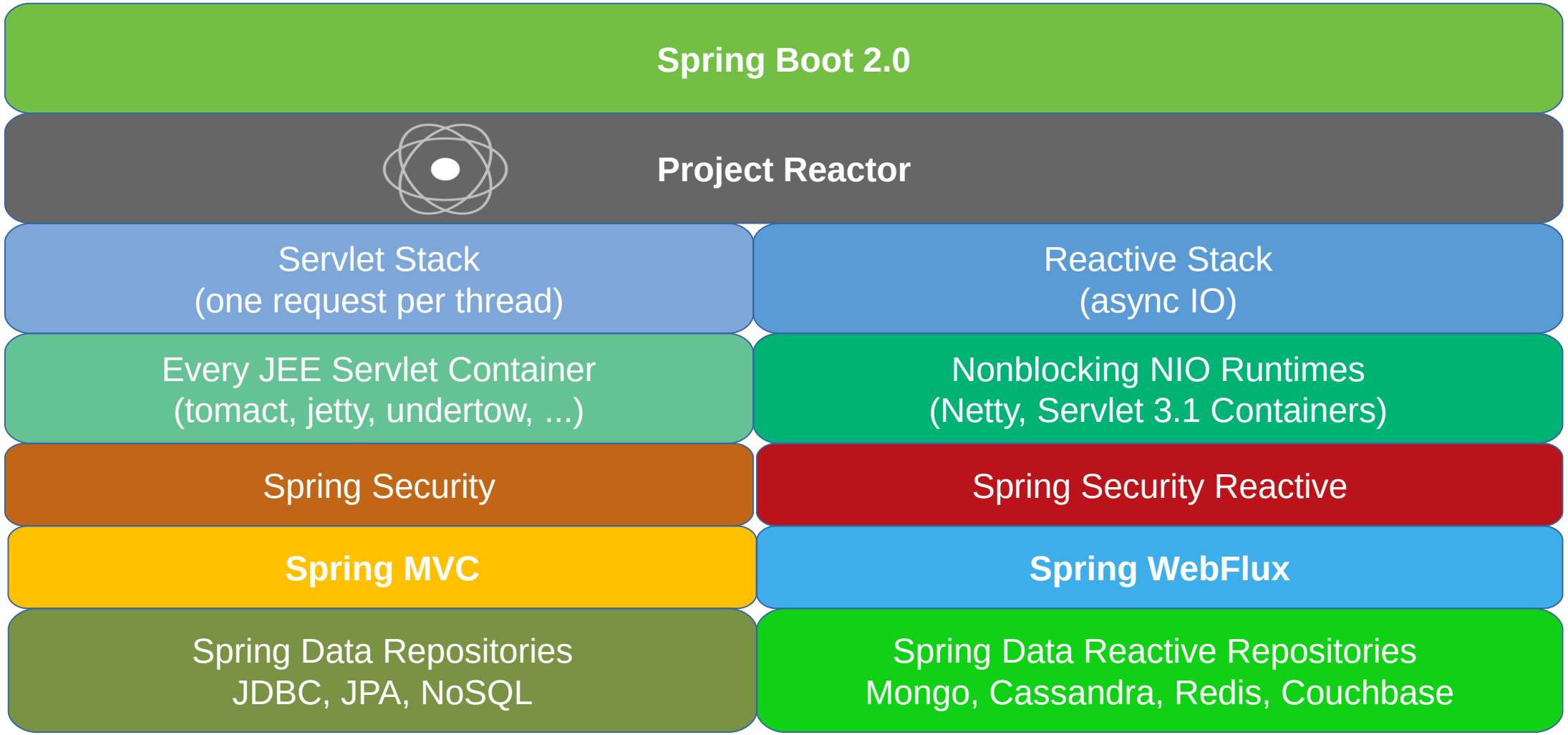
Result:

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

Top New Features in Spring 5

- Reactive Programming Model
- Spring Web Flux
- Reactive DB repositories & integrations + hot event streaming: MongoDB, CouchDB, Redis, Cassandra, Kafka
- JDK 8+ and Java EE 7+ baseline
- Testing improvements – WebTestClient (based on reactive WebFlux WebClient)
- Kotlin functional DSL

Spring 5 Main Building Blocks

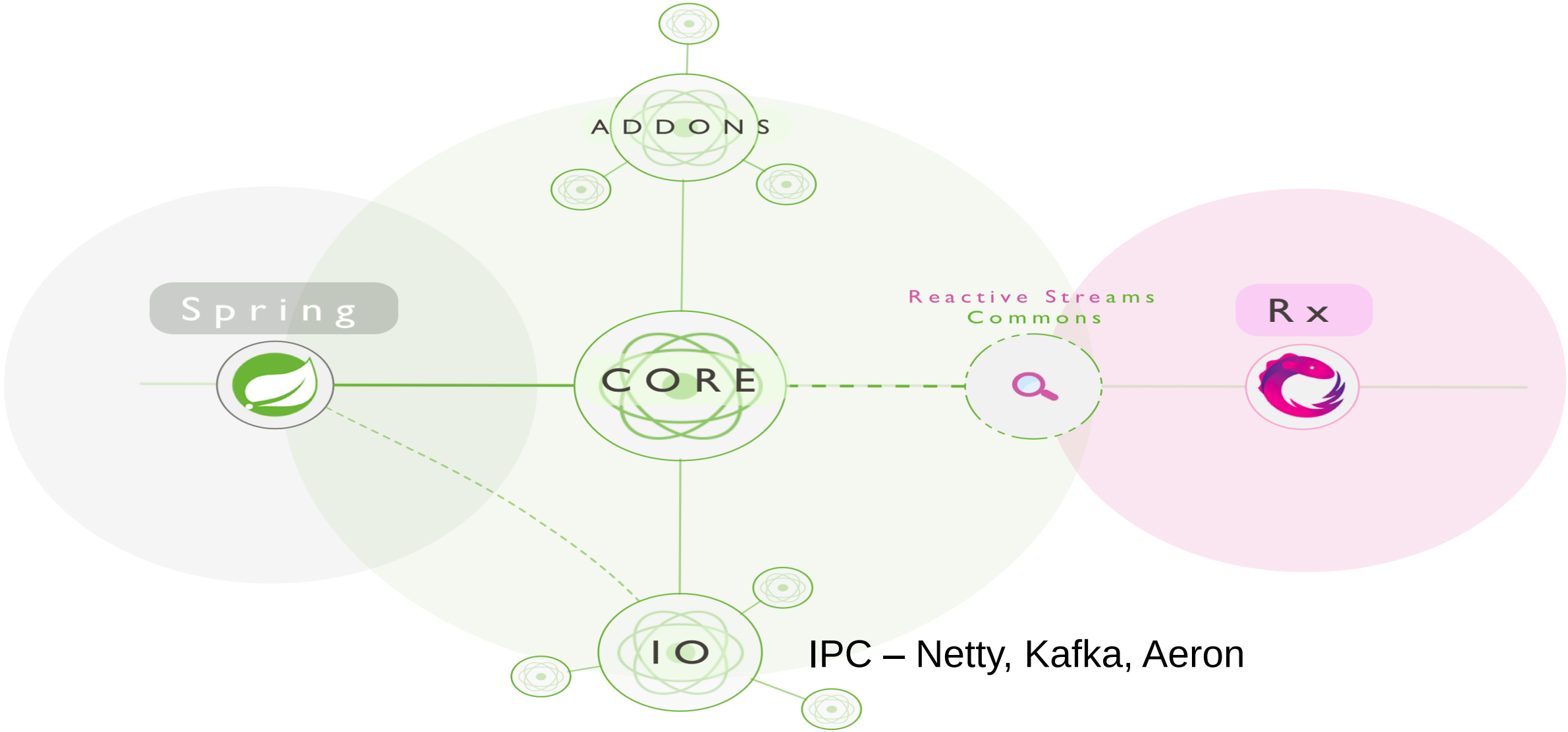


Source: https://spring.io

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 Sub Projects



IPC – Netty, Kafka, Aeron

Hot Stream Example - Reactor

```
val sink = Sinks.many().multicast().onBackpressureBuffer<String>()
val result = sink.asFlux()
//        .publishOn(Schedulers.single())
        .subscribeOn(Schedulers.single())
        .map { it.uppercase() }
        .delayElements(Duration.of(1000, ChronoUnit.MILLIS))
        .filter { s: String -> s.startsWith("HELLO") }

    result.map { data: String -> "Subscriber 1: $data" }.subscribe(System.out::println)
    result.map { data: String -> "Subscriber 2: $data" }.subscribe(System.out::println)

    sink.tryEmitNext("Hello World!") // emit - non blocking
    sink.tryEmitNext("Hello Kotlin!") // emit - non blocking
    sink.tryEmitNext("Hello Reactor!") // emit - non blocking
    sink.tryEmitNext("Goodbye World!")
    sink.tryEmitNext("Hello Trayan!")
```

Reactor Schedulers

Schedulers provides various Scheduler flavors usable by publishOn or subscribeOn:

- parallel(): Optimized for fast Runnable non-blocking executions
- single(): Optimized for low-latency Runnable one-off executions
- elastic(): Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) can grow indefinitely
- boundedElastic(): Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) is capped
- immediate(): to immediately run submitted Runnable instead of scheduling them (somewhat of a no-op or "null object" Scheduler)
- fromExecutorService(ExecutorService) to create new instances around Executors

Main Differences Between Kotlin Flow and Reactor Flux

Flow API is like Java 8 Stream or its Kotlin equivalent Sequence, but the difference is that it is suitable for asynchronous operations and manages backpressure.
So it is Flux equivalent in coroutines world, suitable for hot or cold stream, finite or infinite streams, with the following main differences:

- Flow is push-based while Flux is push-pull hybrid
- Backpressure is implemented via suspending functions
- Flow has only a single suspending collect method and operators are implemented as extensions
- Operators are easy to implement thanks to coroutines
- Extensions allow to add custom operators to Flow
- Collect operations are suspending functions
- map operator supports asynchronous operation (no need for flatMap) since it takes a suspending function parameter

Rsocket – over TCP, WebSockets and Aeron

Network communication is asynchronous. The RSocket protocol embraces this and models all communication as multiplexed streams of messages over a single network connection, and never synchronously blocks while waiting for a response. Some of the key reasons for using RSocket include:

- support for interaction models beyond request/response such as streaming responses and push
- application-level flow control semantics (async pull/push of bounded batch sizes) across network boundaries
- binary, multiplexed use of a single connection
- support resumption of long-lived subscriptions across transport connections
- need of an application protocol in order to use transport protocols such as WebSockets and Aeron

RSocket Interaction Models - I

- Fire-and-Forget - optimization of request/response that is useful when a response is not needed. It allows for significant performance optimizations, not just in saved network usage by skipping the response, but also in client and server processing time as no bookkeeping is needed to wait for and associate a response or cancellation request.
- Request/Response (single-response) – standard request/response semantics are still supported, and still expected to represent the majority of requests over a RSocket connection. These request/response interactions can be considered optimized "streams of only 1 response", and are asynchronous messages multiplexed over a single connection.

RSocket Interaction Models - II

- Request/Stream (multi-response, finite) – Extending from request/response is request/stream, which allows multiple messages to be streamed back. Think of this as a "collection" or "list" response, but instead of getting back all the data as a single response, each element is streamed back in order – e.g. fetching videos, products, file line-by-line
- Channel – a channel is bi-directional, with a stream of messages in both directions. For example:
  - client requests a stream of data that initially bursts the current view of the world
  - deltas/diffs are emitted from server to client as changes occur
  - client updates the subscription over time to add/remove criteria/topics/etc.
  - Without a bi-directional channel, the client would have to cancel the initial request, re-request, and receive all data from scratch, rather than just updating the subscription and efficiently getting just the difference.

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

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