Reactive Programming for a demanding world:

building event-driven and responsive applications with

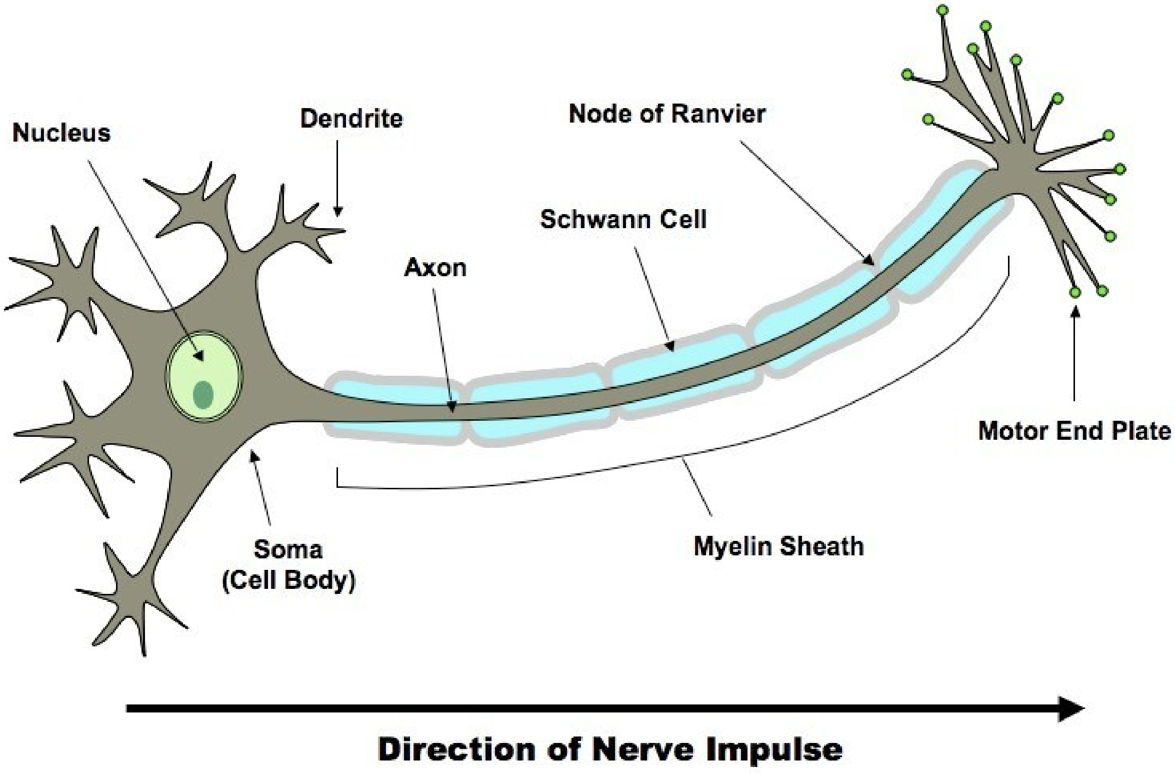
RxJava

by Mario Fusco
mario.fusco@gmail.com
@mariofusco

Reactive

“readily responsive to a stimulus”

Merriam-Webster dictionary



Why reactive? What changed?

A few years ago largest applications had **tens of servers** and **gigabytes of data**
Seconds of response time and **hours of offline** maintenance were **acceptable**

Today

- **Big Data**: usually measured in Petabytes and increasing with an extremely high frequency
- **Heterogeneous environment**: applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors
- **Usage patterns**: Users expect millisecond response times and 100% uptime

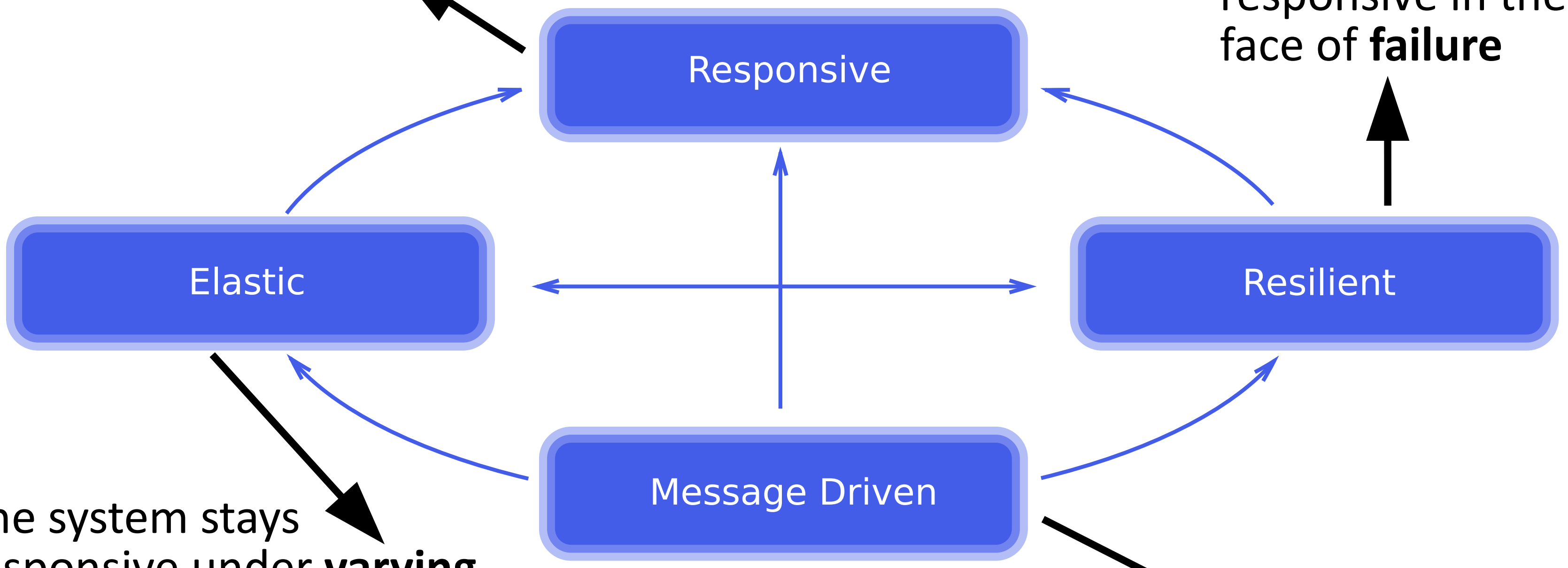
Today's demands are simply not met by yesterday’s software architectures!

The Reactive Manifesto

We Are Reactive

The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of **usability**

The system stays responsive in the face of **failure**



The system stays responsive under **varying workload**. It can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs

The system rely on **asynchronous message passing** to establish a boundary between components that ensures loose coupling, isolation and location transparency

The Reactive Streams Initiative

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure on the JVM

Problem

Handling streams of (live) data in an asynchronous and possibly non-blocking way

Scope

Finding a minimal API describing the operations available on Reactive Streams

Implementors

RxJava

Akka Streams

Ratpack

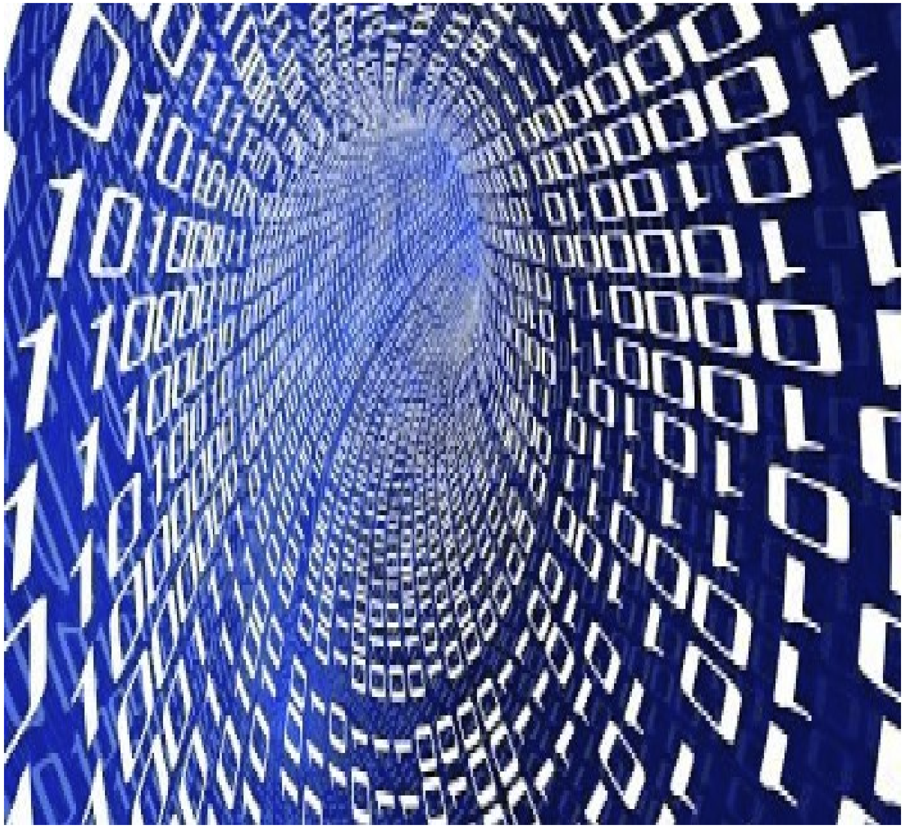
Reactor Composable

Rethinking programming the Reactive way

- Reactive programming is a programming paradigm about **data-flow**
- Think in terms of **discrete events** and **streams** of them
- React to events and define behaviors **combining** them
- The system **state** changes over time based on the **flow of events**
- Keep your data/events **immutable**

Never block!

Reactive programming is programming with asynchronous data streams



- A stream is a sequence of ongoing **events ordered in time**
- Events are processed **asynchronously**, by defining a function that will be executed when an event arrives

See Events Streams Everywhere

stock prices

weather

mouse position

shop's orders

flights/trains arrivals

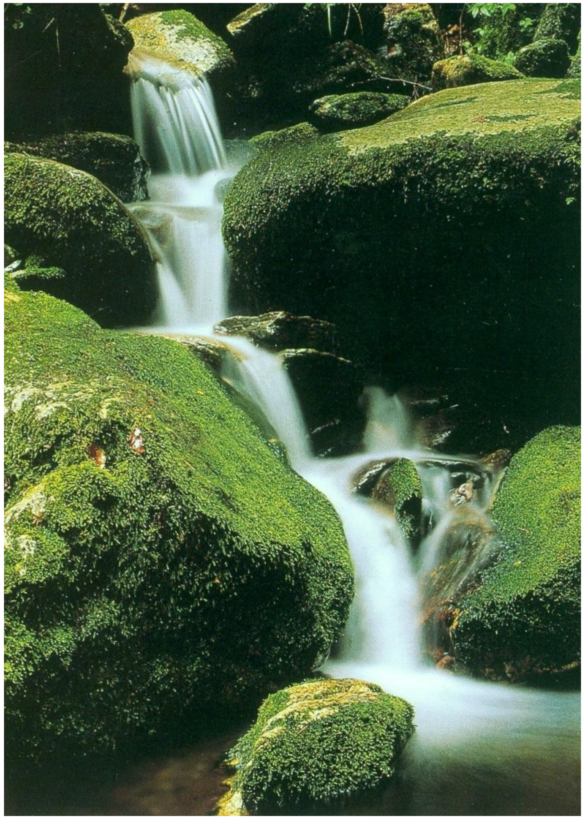
time

Reactive Programming Mantra

Streams are not collections

Streams are

- potentially unbounded in length
- focused on transformation of data
- time-dependent
- ephemeral
- traversable only once



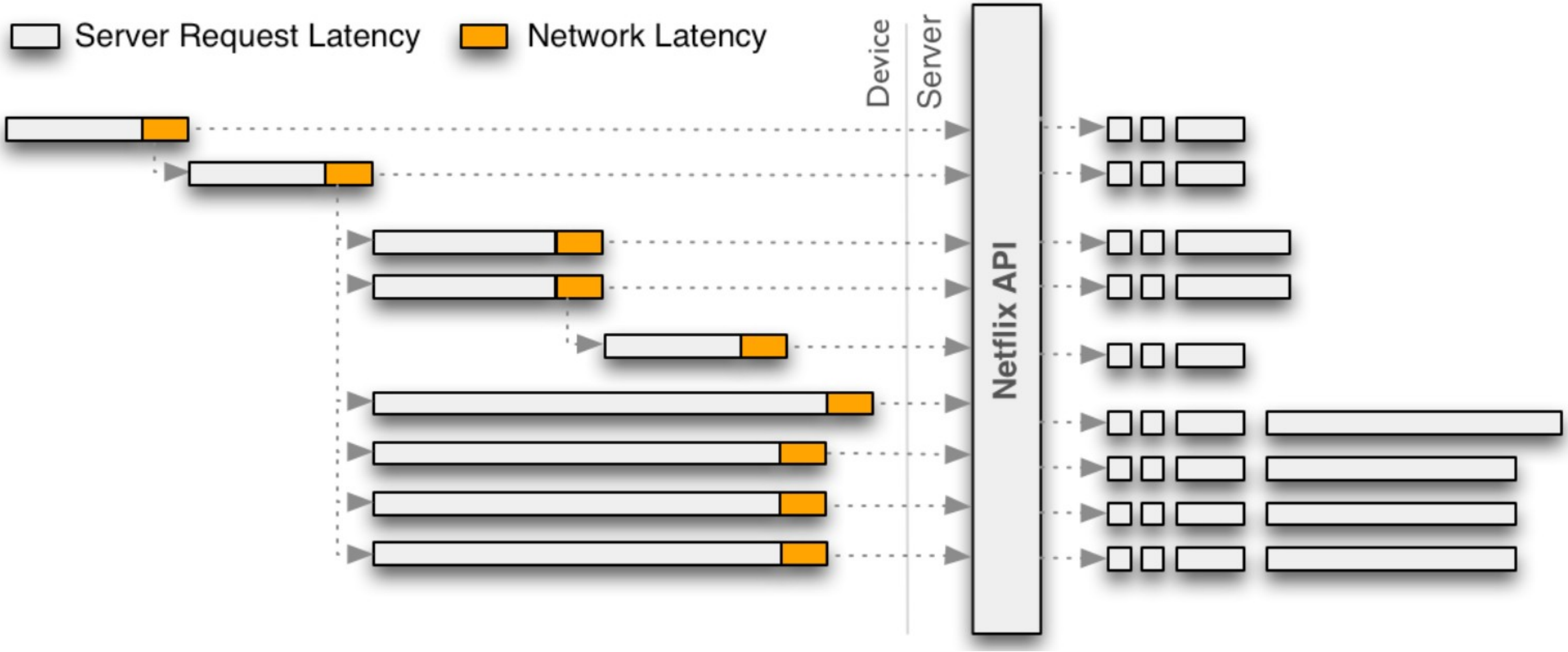
«You cannot step twice into the same stream. For as you are stepping in, other waters are ever flowing on to you.»
— Heraclitus

RxJava
Reactive Extension for async programming

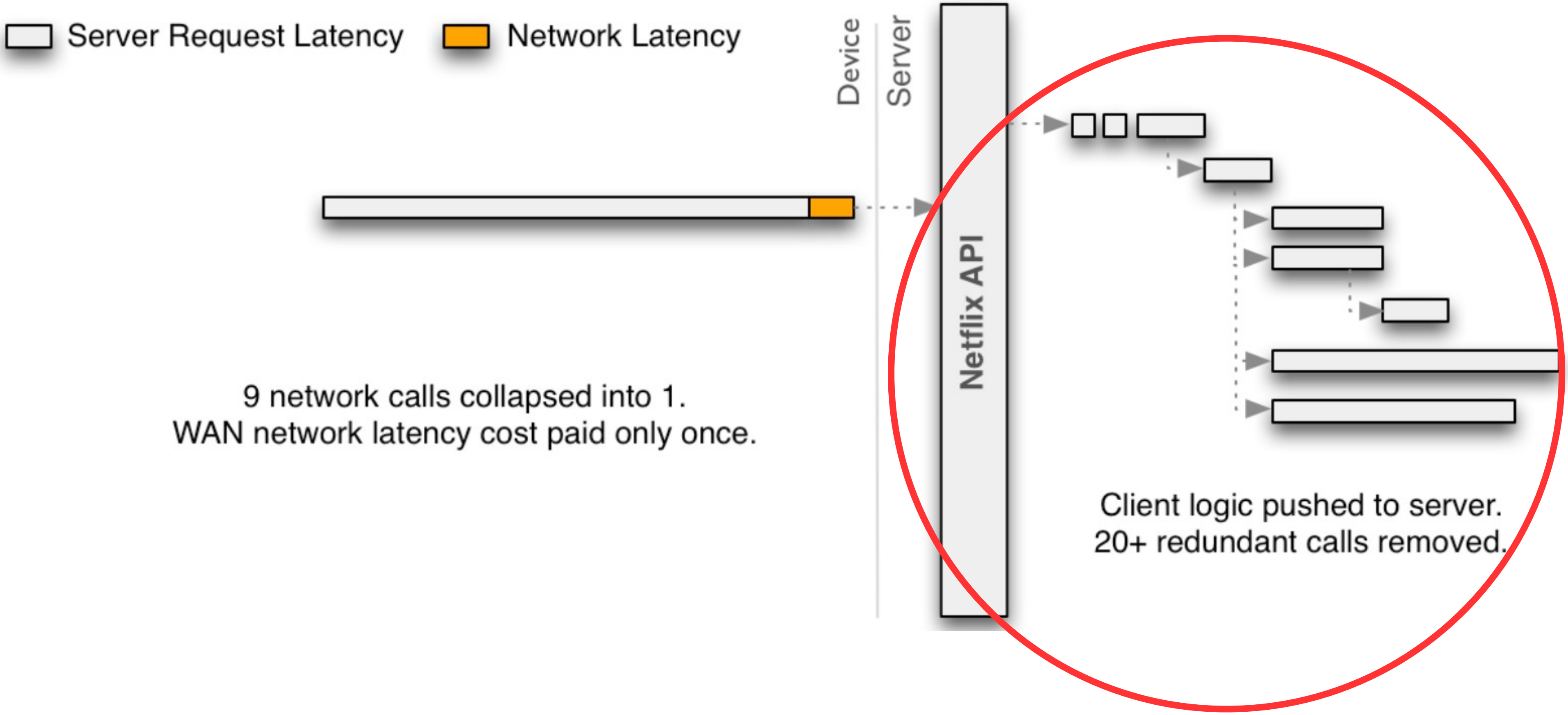
- A library for **composing asynchronous and event-based** programs using observable sequences for the Java VM
- Supports Java 6 or higher and JVM-based languages such as Groovy, Clojure, JRuby, Kotlin and Scala
- Includes a **DSL** providing extensive operations for streams transformation, filtering and recombination
- Implements pure “**push**” model
- **Decouple** events production from consumption
- Allows blocking only for **back pressure**
- First class support for **error handling**, **scheduling** & **flow control**
- Used by **Netflix** to make the entire service layer asynchronous

http://reactivex.io
http://github.com/ReactiveX/RxJava

How Netflix uses RxJava
From N network call ...



… to only 1
Pushing client logic to server



9 network calls collapsed into 1.
WAN network latency cost paid only once.

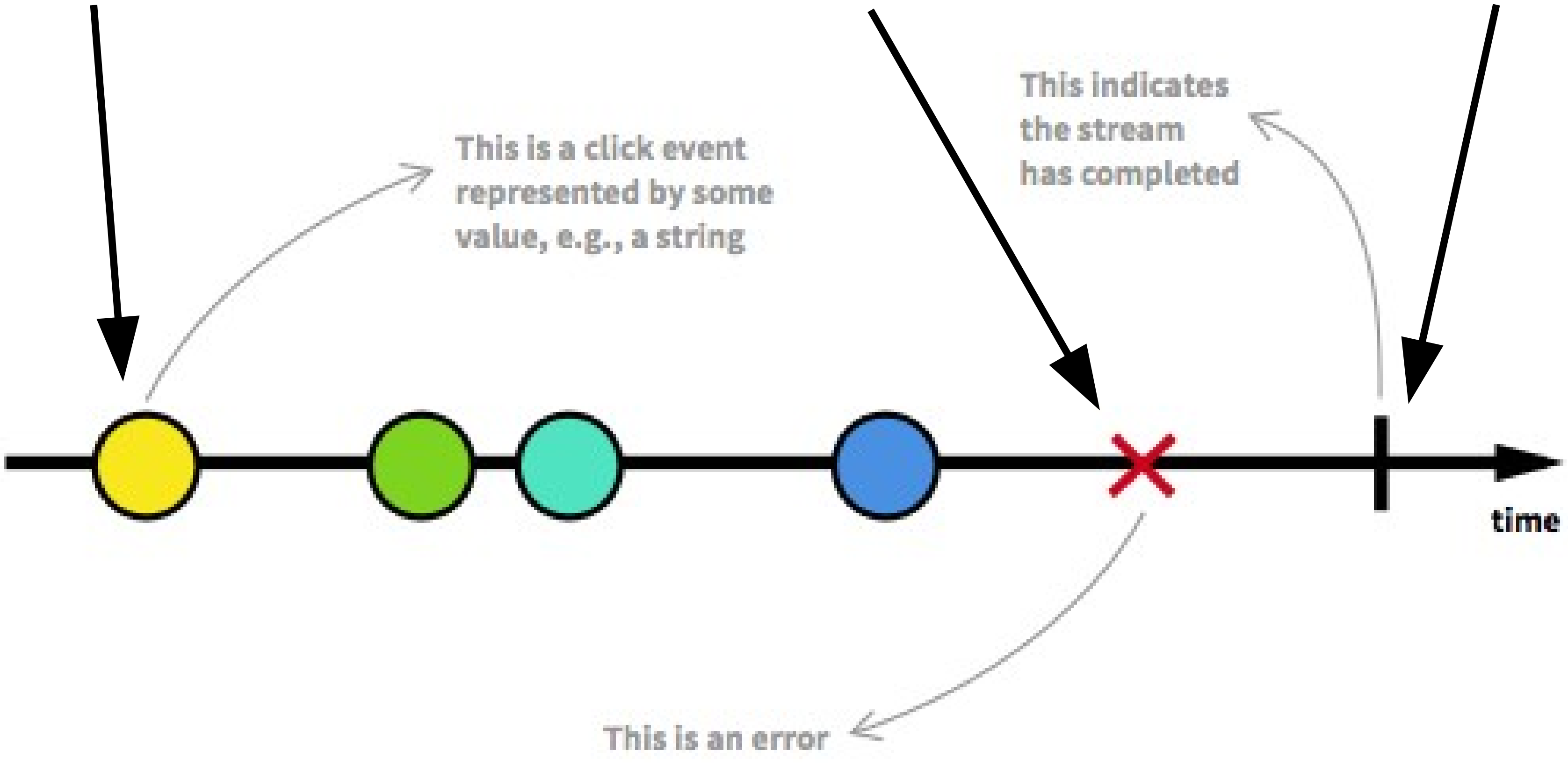
Client logic pushed to server.
20+ redundant calls removed.

Marble diagrams: Representing events' streams ...

A stream is **a sequence of ongoing events ordered in time**.

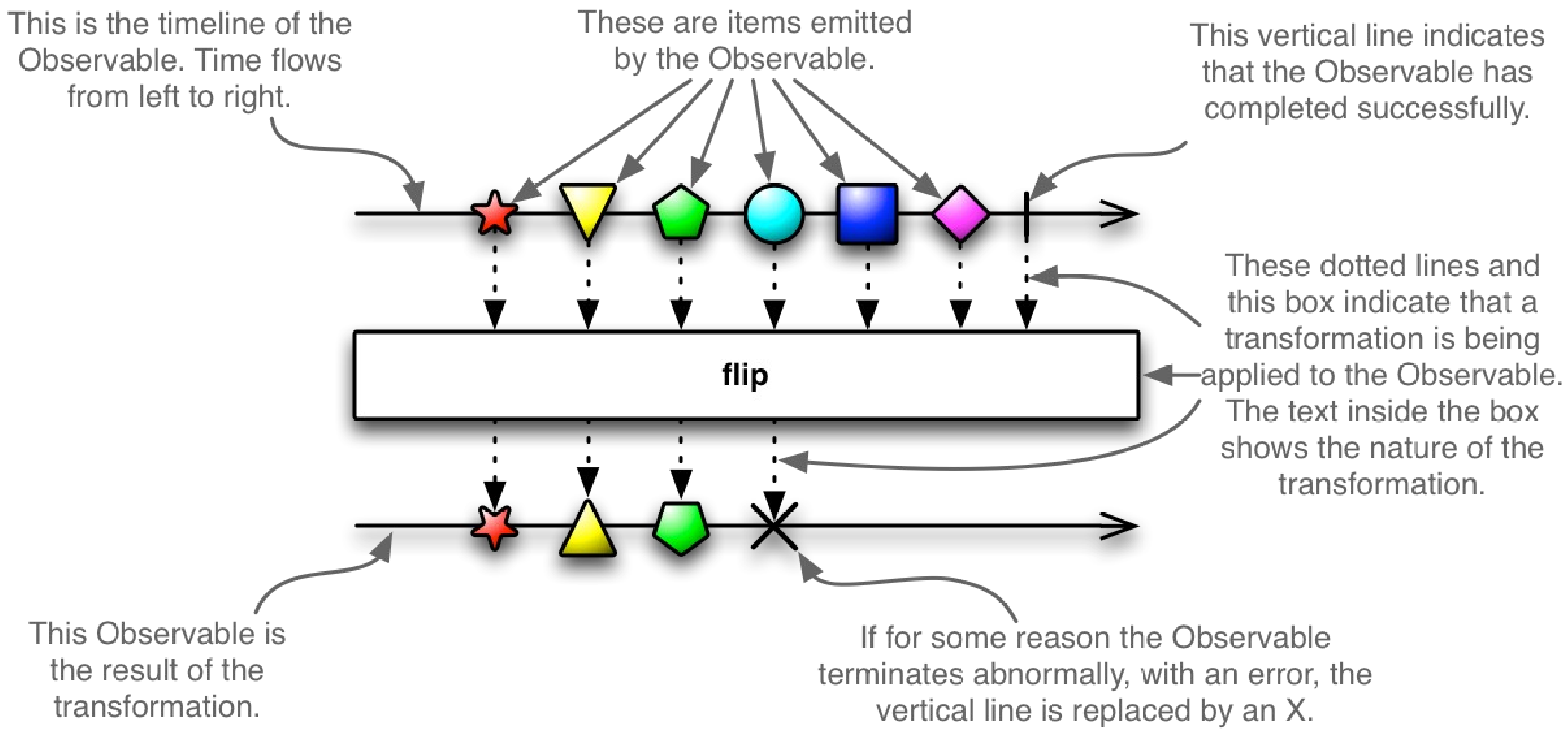
It can emit three different things:

1. a value (of some type)
2. an error
3. "completed" signal

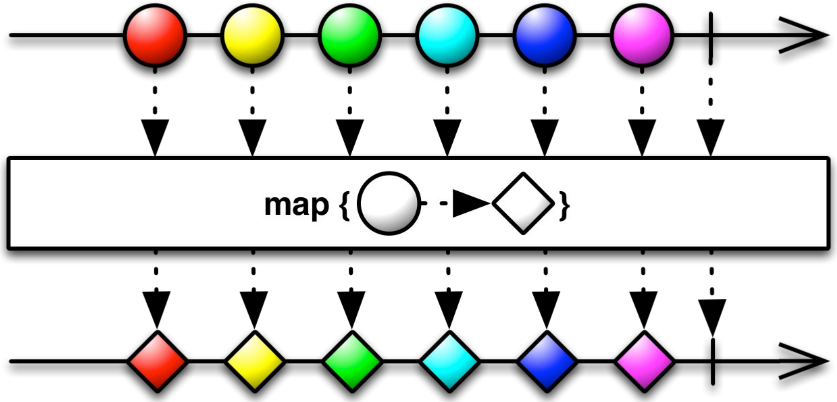


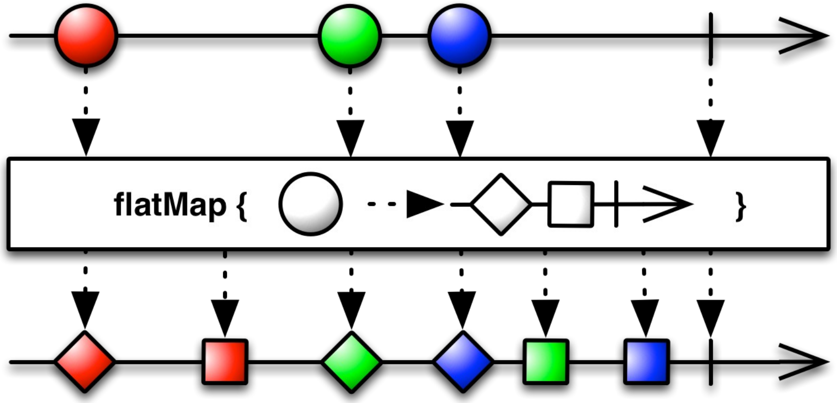
This is a click event represented by some value, e.g., a string
This indicates the stream has completed
time
This is an error

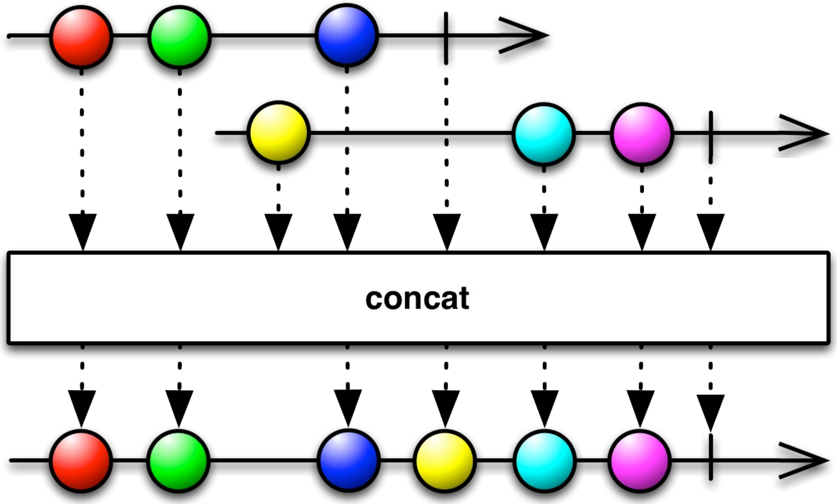
… and events' transformations

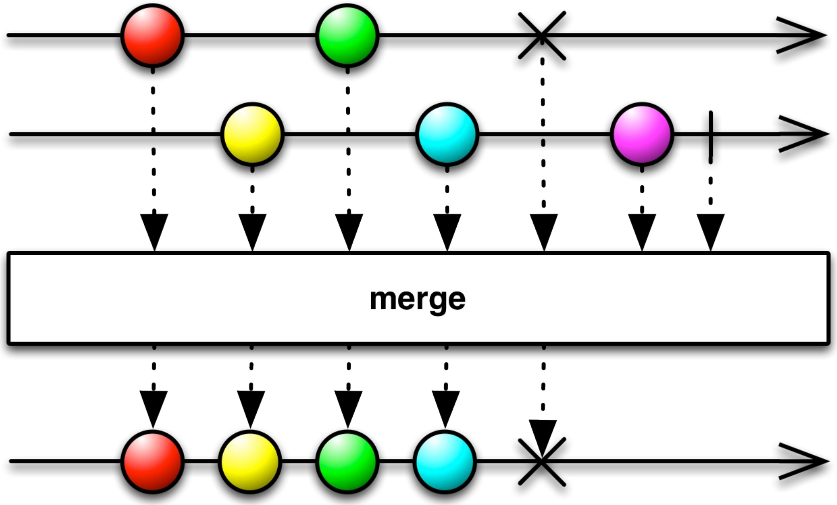

This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
flip
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

RxJava operations as marble diagrams









Observable

The Observable interface defines how to access
asynchronous sequences of multiple items

	single value	multiple values
synchronous	`T getData()`	`Iterable<T> getData()`
asynchronous	`Future<T> getData()`	`Observable<T> getData()`

An Observable is the asynchronous/push “dual” to the synchronous/pull Iterable

	Iterable (pull)	Obesrvable (push)
retrieve data	`T next()`	`onNext(T)`
signal error	`throws Exception`	`onError(Exception)`
completion	`!hasNext()`	`onCompleted()`

Observable as async Stream

```
// Stream<Stock> containing 100 Stocks
getDataFromLocalMemory()
    .skip(10)
    .filter(s -> s.getValue > 100)
    .map(s -> s.getName() + “: ” + s.getValue())
    .forEach(System.out::println);
```

```
// Observable<Stock> emitting 100 Stocks
getDataFromNetwork()
    .skip(10)
    .filter(s -> s.getValue > 100)
    .map(s -> s.getName() + “: ” + s.getValue())
    .forEach(System.out::println);
```

Observable and Concurrency

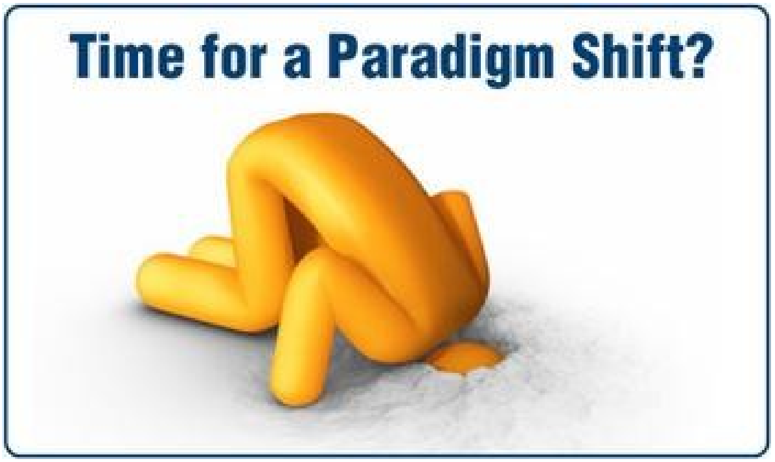
An Observable is **sequential** → No concurrent emissions

Scheduling and combining Observables enables **concurrency while retaining sequential emission**

Reactive Programming requires a mental shift

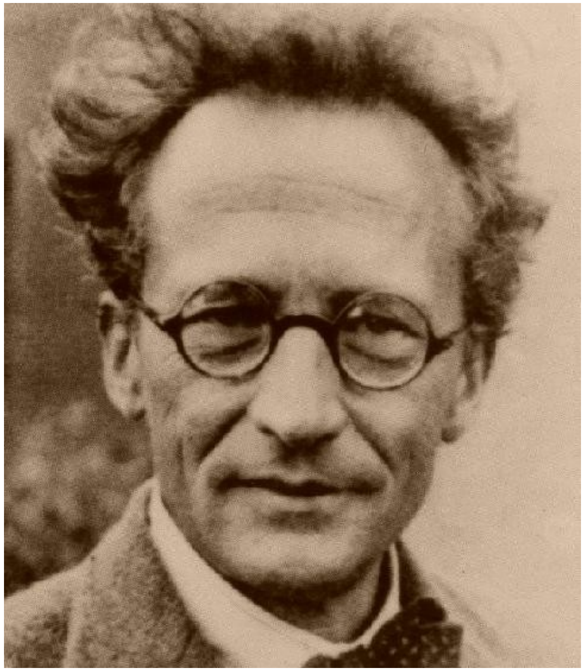
from **imperative** to **functional**

from **sync** to **async**

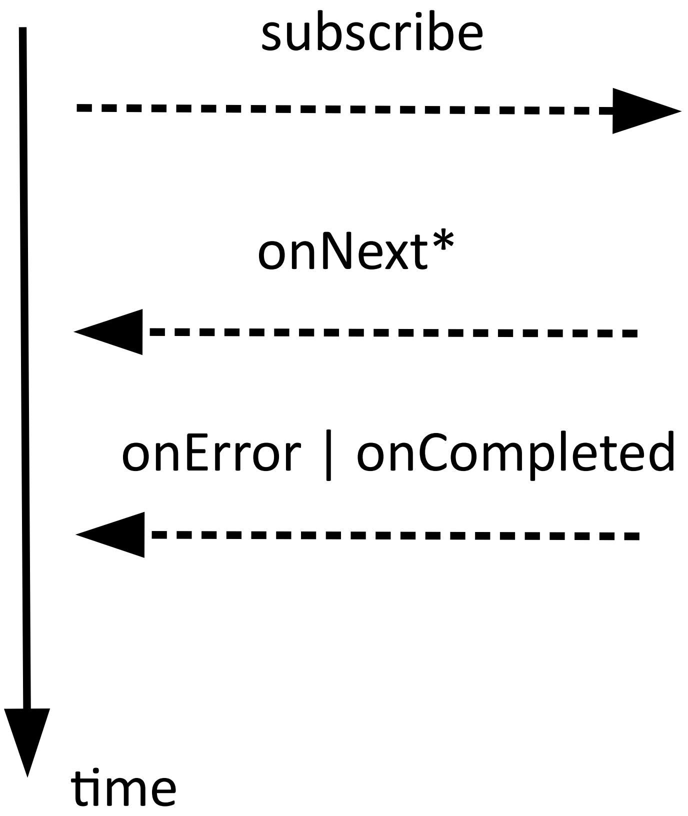


from **pull** to **push**

Observing an Observable



Observer



subscribe

onNext*

onError | onCompleted

time

Observable

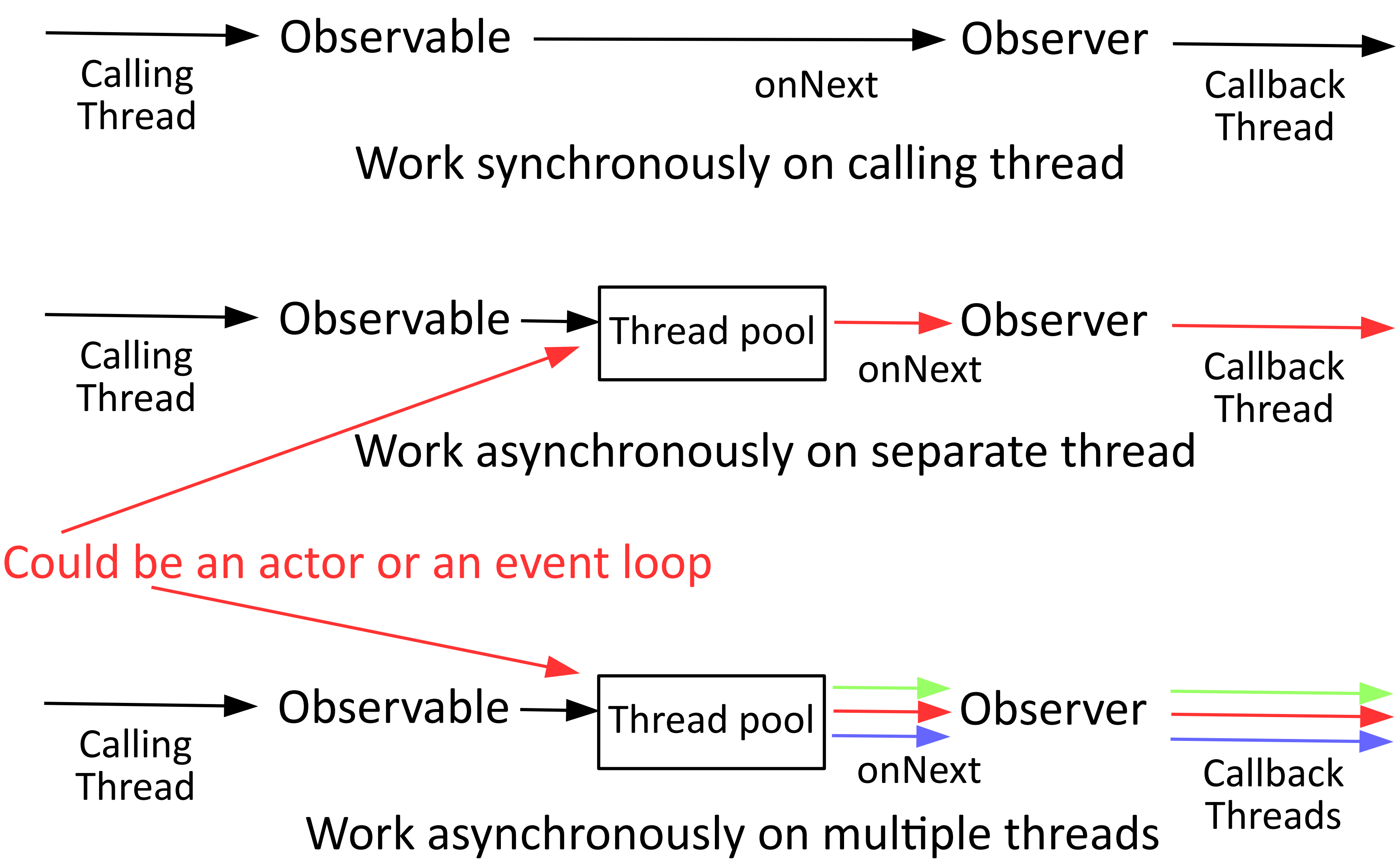
How is the Observable implemented?

- Maybe it executes its logic on subscriber thread?
- Maybe it delegates part of the work to other threads?
- Does it use NIO?
- Maybe it is an actor?
- Does it return cached data?

Observer does not care!

```
public interface Observer<T> {
    void onCompleted();

    void onError(Throwable var1);

    void onNext(T var1);
}
```

Non-Opinionated Concurrency



Calling Thread

Observable

onNext

Observer

Callback Thread

Work synchronously on calling thread

Calling Thread

Observable

Thread pool

onNext

Observer

Callback Thread

Work asynchronously on separate thread

Could be an actor or an event loop

Calling Thread

Observable

Thread pool

onNext

Observer

Callback Threads

Work asynchronously on multiple threads

Enough speaking

Show me the code!

Fetching town's temperature

```
public class TempInfo {

    public static final Random random = new Random();

    public final String town;
    public final int temp;

    public TempInfo(String town, int temp) {
        this.town = town;
        this.temp = temp;
    }

    public static TempInfo fetch(String temp) {
        return new TempInfo(temp, random.nextInt(70) - 20);
    }

    @Override
    public String toString() {
        return String.format(town + " : " + temp);
    }
}
```

Creating Observables ...

- ... with just a single value

```
public static Observable<TempInfo> getTemp(String town) {
    return Observable.just(TempInfo.fetch(town));
}
```

- ... from an Iterable

```
public static Observable<TempInfo> getTemps(String... towns) {
    return Observable.from(Stream.of(towns)
                     .map(town -> TempInfo.fetch(town))
                     .collect(toList()));
}
```

- ... from another Observable

```
public static Observable<TempInfo> getFeed(String town) {
    return Observable.create(subscriber -> {
        while (true) {
            subscriber.onNext(TempInfo.fetch(town));
            Thread.sleep(1000);
        }
    });
}
```

Combining Observables

- Subscribing one Observable to another

```
public static Observable<TempInfo> getFeed(String town) {
    return Observable.create(
        subscriber -> Observable.interval(1, TimeUnit.SECONDS)
                .subscribe(i -> subscriber
                                    .onNext(TempInfo.fetch(town))));
}
```

- Merging more Observables

```
public static Observable<TempInfo> getFeeds(String... towns) {
    return Observable.merge(Arrays.stream(towns)
                                  .map(town -> getFeed(town))
                                  .collect(toList()));
}
```

Managing errors and completion

```
public static Observable<TempInfo> getFeed(String town) {
    return Observable.create(subscriber ->
        Observable.interval(1, TimeUnit.SECONDS)
                  .subscribe(i -> {
                       if (i > 5) subscriber.onCompleted();
                       try {
                           subscriber.onNext(TempInfo.fetch(town));
                       } catch (Exception e) {
                           subscriber.onError(e);
                       }
    }));
}
```

```
      Observable<TempInfo> feed = getFeeds("Milano", "Roma", "Napoli");
      feed.subscribe(new Observer<TempInfo>() {
          public void onCompleted() { System.out.println("Done!"); }

          public void onError(Throwable t) {
              System.out.println("Got problem: " + t);
          }

          public void onNext(TempInfo t) { System.out.println(t); }
      });
```

Hot & Cold Observables

HOT

emits immediately whether its Observer is ready or not

examples
mouse & keyboard events
system events
stock prices
time

COLD

emits at controlled rate when requested by its Observers

examples
in-memory Iterable
database query
web service request
reading file

Dealing with a slow consumer

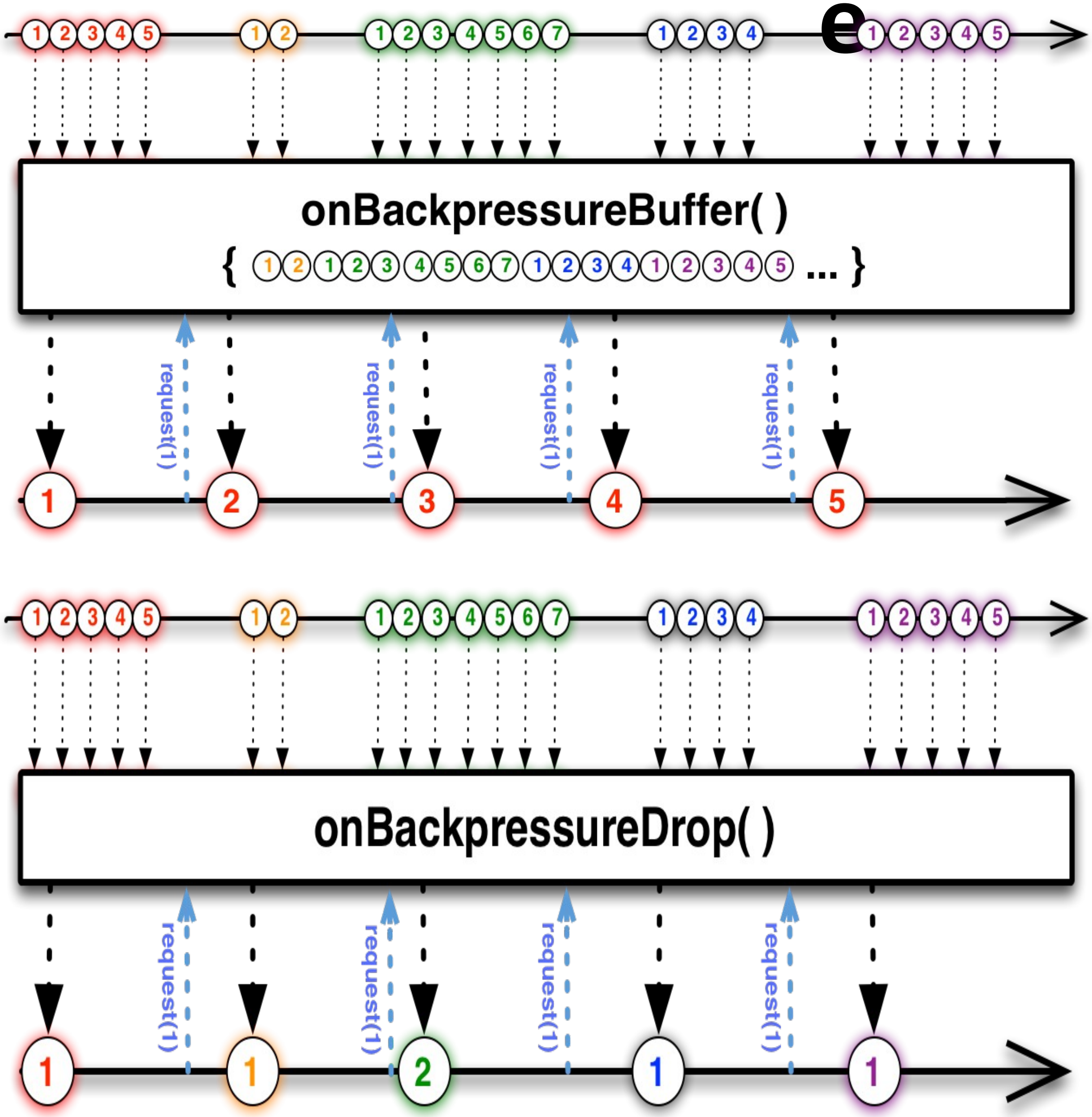
Push (reactive) when consumer keeps up with producer

Switch to Pull (interactive) when consumer is slow

When you subscribe to an Observable, you can request reactive pull backpressure

```
observable.subscribe(new Subscriber<T>() {

    @Override public void onStart() { request(1); }

    @Override
    public void onCompleted() { /* handle sequence-complete */ }

    @Override
    public void onError(Throwable e) { /* handle error */ }

    @Override public void onNext(T n) {
      // do something with the emitted item
      request(1); // request another item
    }
});
```

Backpressure



Reactive pull backpressure isn't magic

Backpressure doesn't make the problem of an overproducing Observable or an underconsuming Subscriber go away. It just moves the problem up the chain of operators to a point where it can be handled better.

Thanks … Questions?

Q

A

Mario Fusco
Red Hat – Senior Software Engineer

mario.fusco@gmail.com
twitter: @mariofusco