Reactive Programming

...

Silviu Marcu
https://www.linkedin.com/in/smarcuxp

- Why Reactive?
- The Reactive Manifesto
- Is Reactive just a trend?
- Spring reactor demo - Basics
- Spring webflux demo - Reactive Web App

2000

Multithreading
CGI vs.
Servlets

2000

Multithreading
CGI vs.
Servlets

2005

Java 5

2000

Multithreading
CGI vs.
Servlets

2005

Java 5

2010

Cloud
Microservices

2000

Multithreading
CGI vs.
Servlets

2005

Java 5

2010

Cloud
Microservices

2015

Reactive

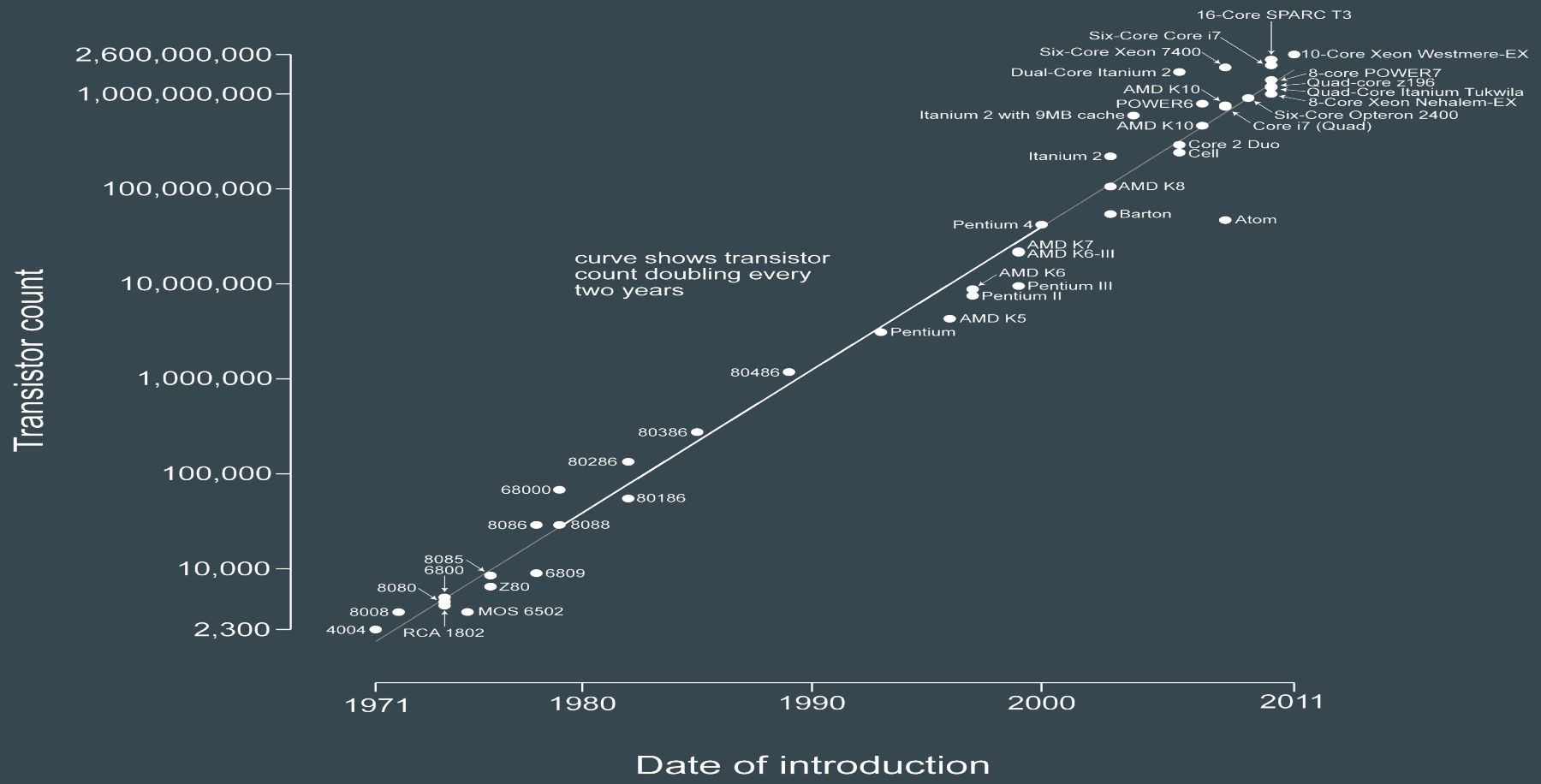
IoT
ML / AI

"For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law. Writing code that effectively exploits multiple processors can be very challenging. "

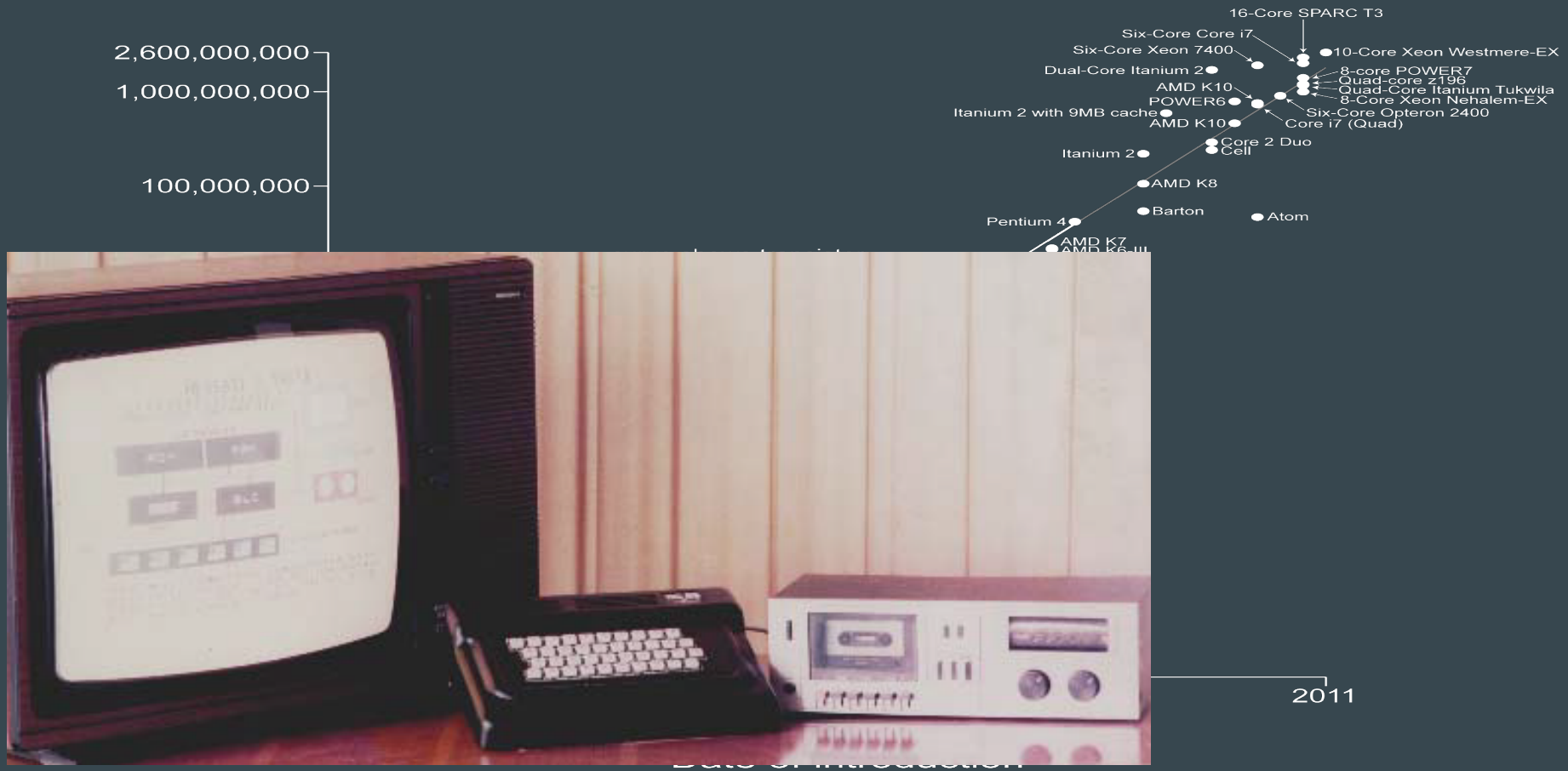
--Doron Rajwan

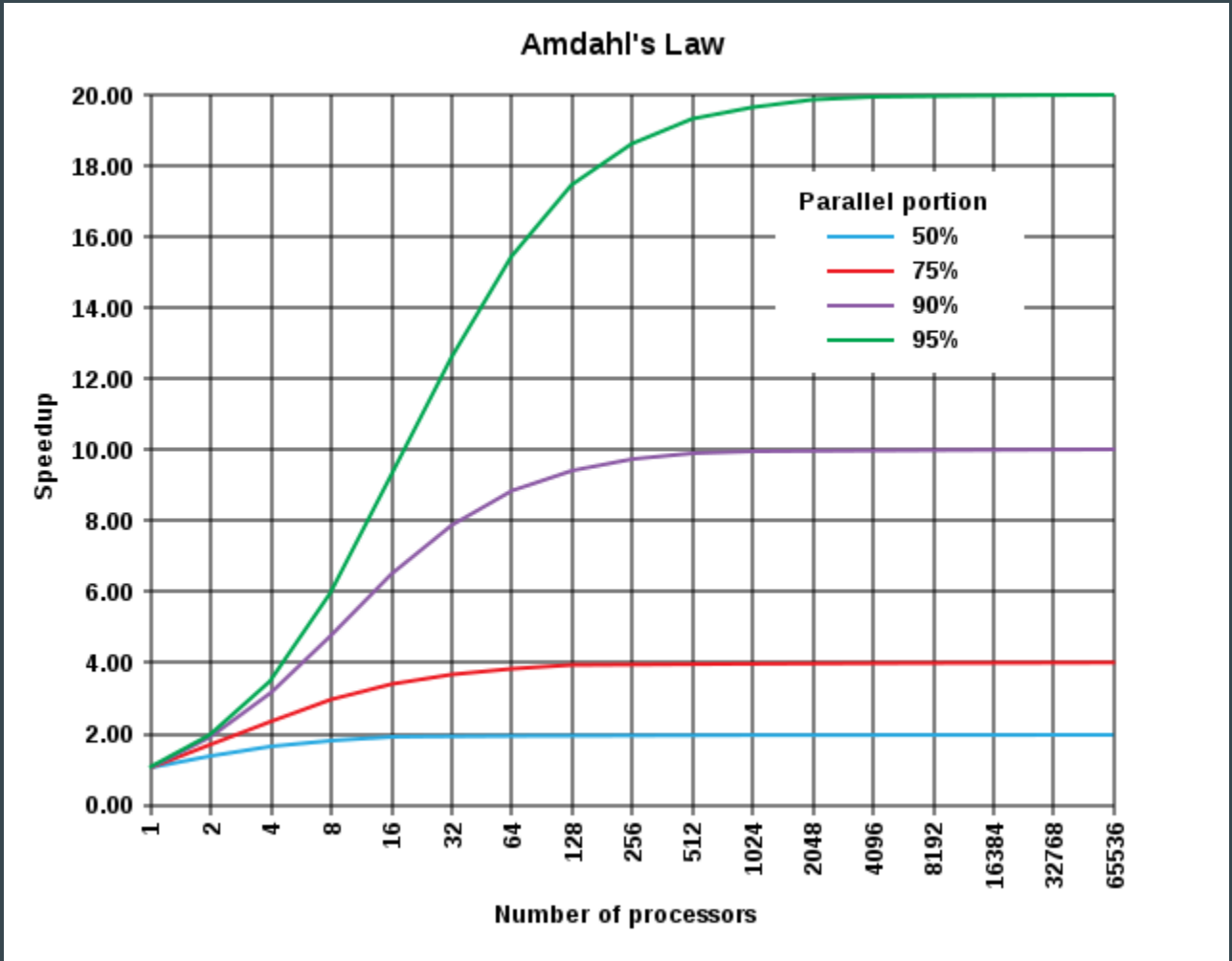
Research Scientist, Intel Corp

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Microprocessor Transistor Counts 1971-2011 & Moore's Law




Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of processors

New Ecosystem

Cloud

Decoupled Services

Latency

Existing Applications

- Already has to deal with LATENCY
- Scalability issues

Reactive Applications

- Non blocking and Event Driven architecture
- Flow Control (pushback pressure)

ASYNCHRONICITY / NON BLOCKING

Asynchronicity … callback … callback … callback … callback

```
function doSomething(params){
  $.get(url, function(result){
    setTimeout(function(){
      startAsyncProcess(function(){
        $.post(url, function(response){
          if(response.good){
            setStateasGoodResponse(function(){
              console.log('Hooray!')
            });
          }
        });
      });
    });
  });
}
```

Blocking vs. Non-Blocking

The Reactive Manifesto

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.

The Reactive Manifesto

Responsive:

The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

The Reactive Manifesto

Resilient:

The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

The Reactive Manifesto

Elastic:

The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

The Reactive Manifesto

Message Driven:

Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency. This boundary also provides the means to delegate failures as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

Is Reactive just a trend?

Spring 5 gets REACTIVE

https://spring.io/blog/2016/09/22/new-in-spring-5-functional-web-framework

Java 9 - reactive programming (flow api)

https://community.oracle.com/docs/DOC-1006738

Project Reactor - https://projectreactor.io/docs

Reactive X - https://github.com/ReactiveX

RxJava - https://github.com/ReactiveX/RxJava

Spring Reactor Demo 1 - Basics

git clone https://github.com/smarcu/spring-reactor-demo1

git checkout tags/STEP-1	**Flux, log, map, subscribe**
git checkout tags/STEP-2	**Subscriber**
git checkout tags/STEP-3	**Subscriber with batch**
git checkout tags/STEP-4	**Threads (single background)**
git checkout tags/STEP-5	**Threads (multiple background)**
git checkout tags/STEP-6	**Threads (publisher)**

Publisher - Subscriber

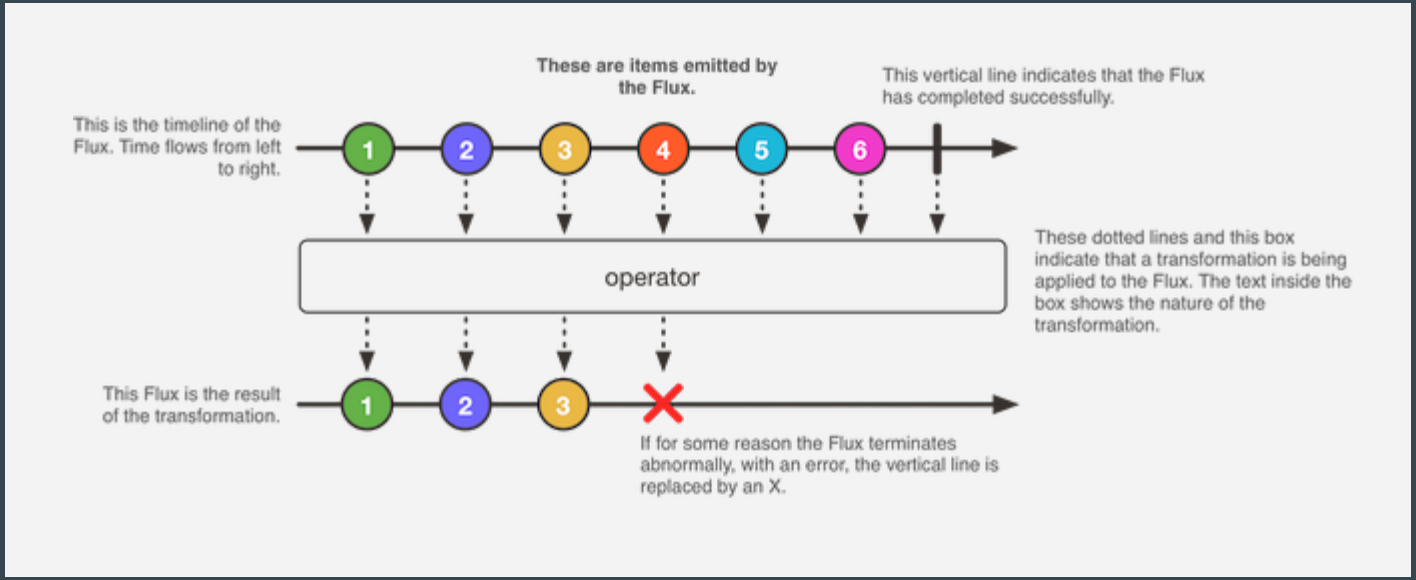
```
public interface Publisher<T> {
   void subscribe(Subscriber<? super T> var);
}

public interface Subscriber<T> {
   void onSubscribe(Subscription var);
   void onNext(T var);
   void onError(Throwable var);
   void onComplete();
}
```

The Publisher notifies the Subscriber of newly available values *as they come*, and this push aspect is key to being reactive.

Operations applied to pushed values <u>are expressed declaratively</u> rather than imperatively.

Flux <T>

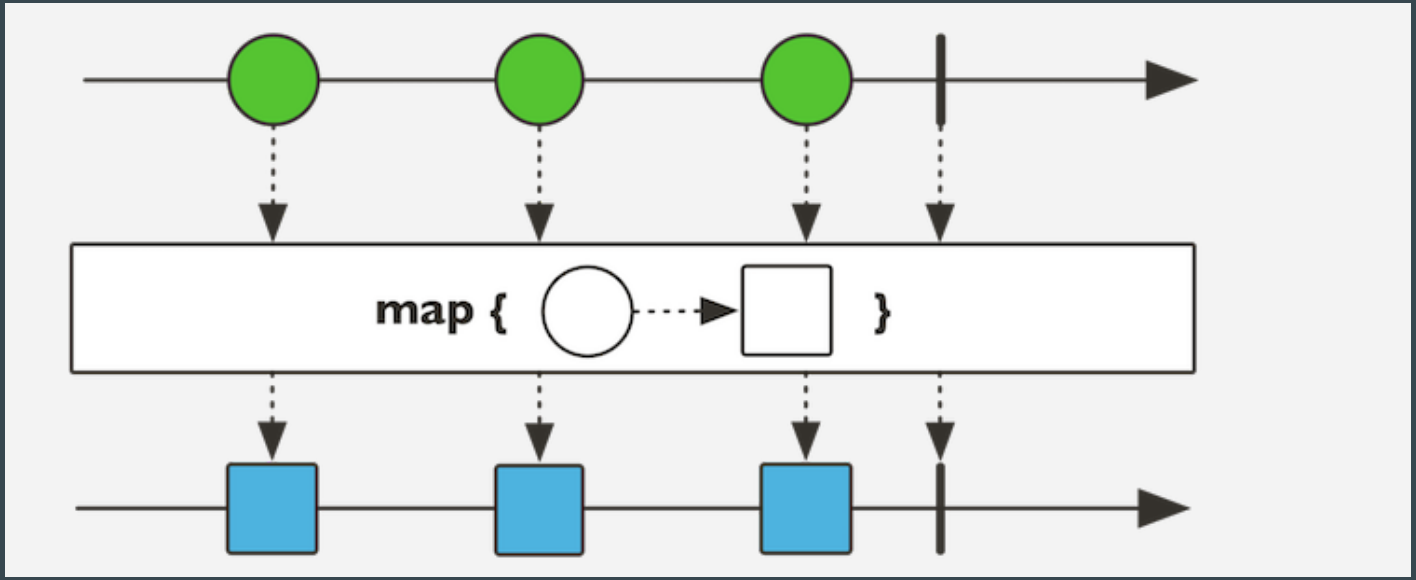
A Reactive Streams Publisher with rx operators that emits 0 to N elements, and then completes (successfully or with an error).



https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html
https://projectreactor.io/docs/core/release/reference/docs/index.html#intro-reactive

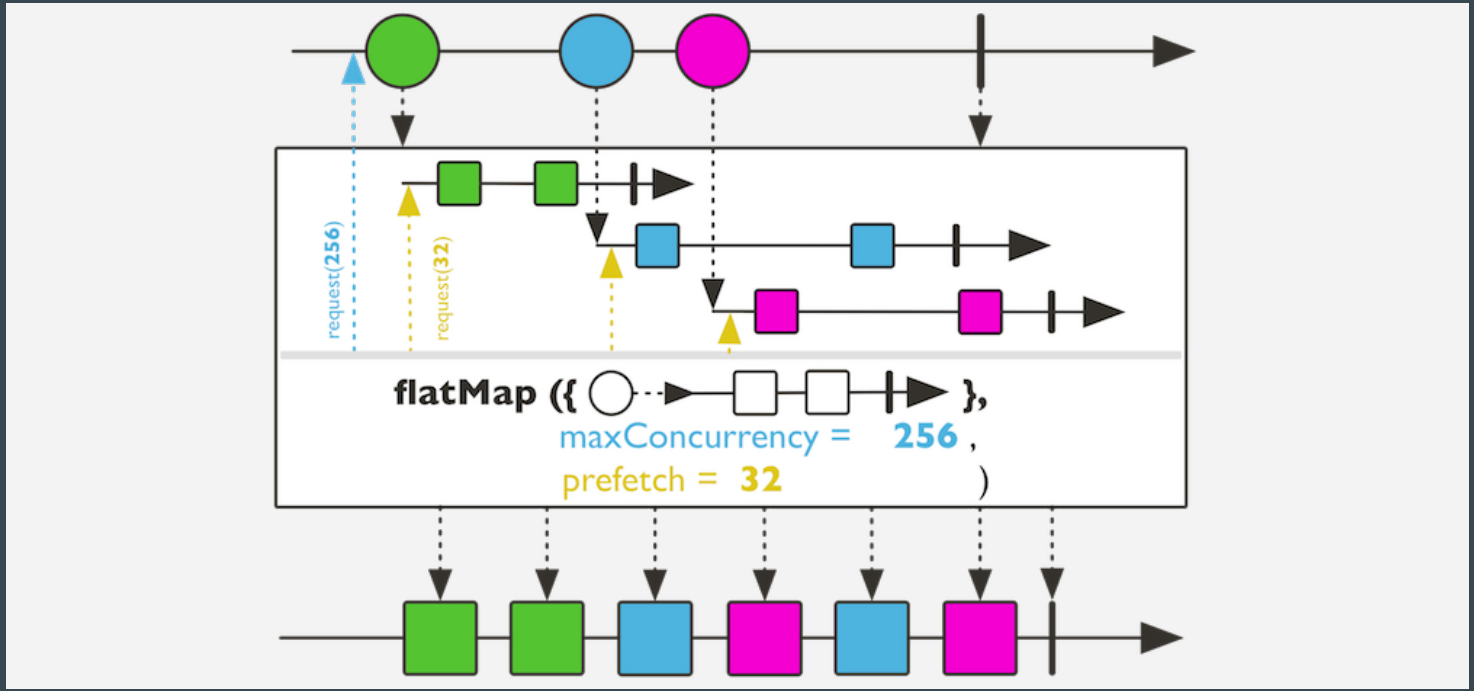
Flux - map()

Transform the items emitted by this Flux by applying a function to each item.



Flux - flatMap()

Transform the items emitted by this Flux into Publishers, then flatten the emissions from those by merging them into a single Flux, so that they may interleave. The concurrency argument allows to control how many merged Publisher can happen in parallel.



Spring Reactor Demo 2 - Web App

git clone https://github.com/smarcu/spring-webflux-demo

git checkout tags/STEP-1	**Rest server, javascript sse**
git checkout tags/STEP-2	**Add Spring Integration**
git checkout tags/STEP-3	**Implement js gyroscope**
git checkout tags/STEP-4	**inputCh - flux - outputCh**

Reference

https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape

https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code

From Imperative to Reactive Apps: https://www.infoq.com/presentations/imperative-reactive-web-apps

Reactive Streams: http://www.reactive-streams.org/

Html javascript server side events: https://www.html5rocks.com/en/tutorials/eventsource/basics/