Reactive Streams.

4 interfaces.

Et après ?

Julien Ponge

Red Hat

Photo by Arno Senoner on Unsplash

"Reactive Streams is an initiative to provide a **standard** for **asynchronous stream processing** with **non-blocking back pressure.**"

– https://www.reactive-streams.org/

MUTINY!

akka

User-facing APIs

< your own library / client / driver / ... >

Protocol

Back-pressured, asynchronous, non-blocking
(see Reactive Streams TCK)

APIs

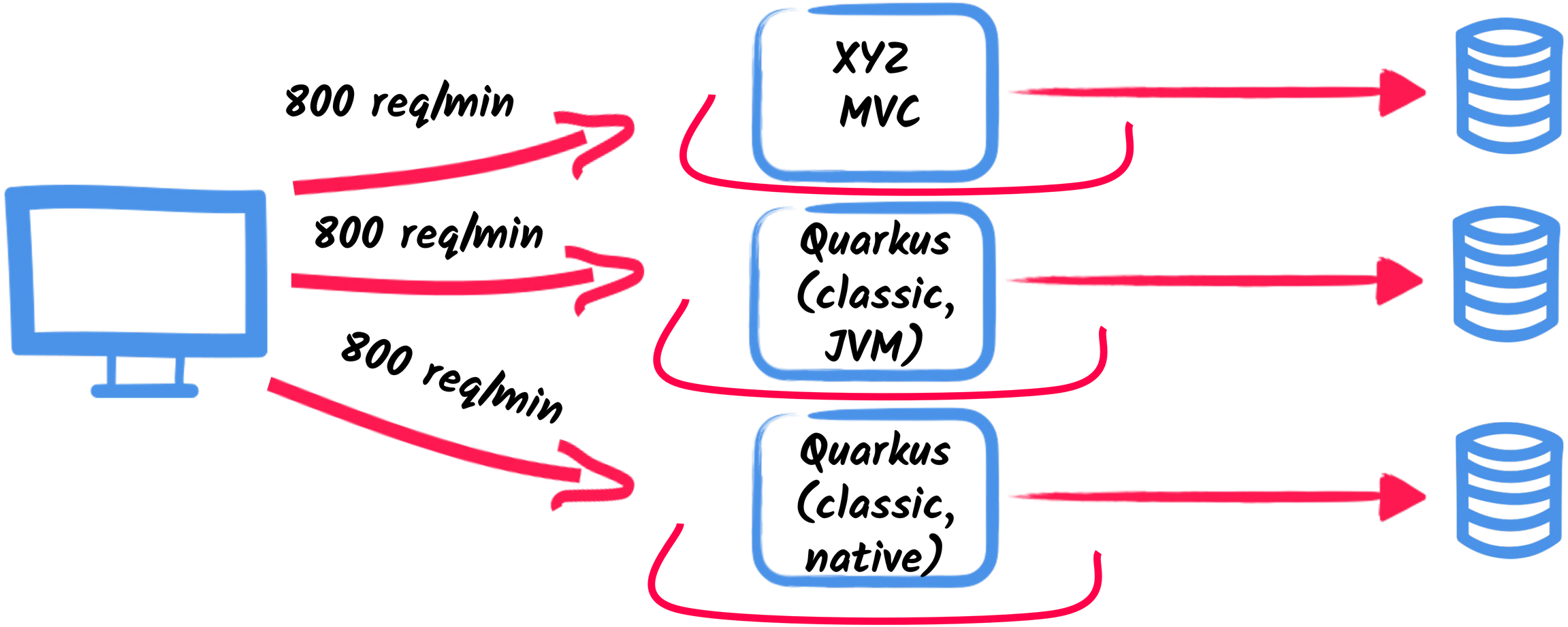
org.reactivestreams.*
(Java 6 legacy)

java.util.concurrent.Flow.*
(since JDK 9)

Show me what's reactive code like?
why reactive matters?

in a containers world

The 20* days test



$$$

Heap Size: 256Mb

Application	Instance type	Price per month (Instance + ESB)
XYZ MVC	T2.medium (2 vCPU, 4 Gb)	26.08 $
XYZ MVC	T2.small (2 vCPU, 1 Gb)	14.70 $ (1.86% of timeout)
Quarkus	T2.small (2 vCPU, 1 Gb)	14.70 $
Quarkus	T2.micro (1 vCPU, 1Gb)	9.15 $
Quarkus Native	T2.micro (1 vCPU, 1Gb)	9.15 $
Quarkus Native	T2.nano (1 vCPU, 0.5 Gb)	6.25 $ (3.64% of timeout)

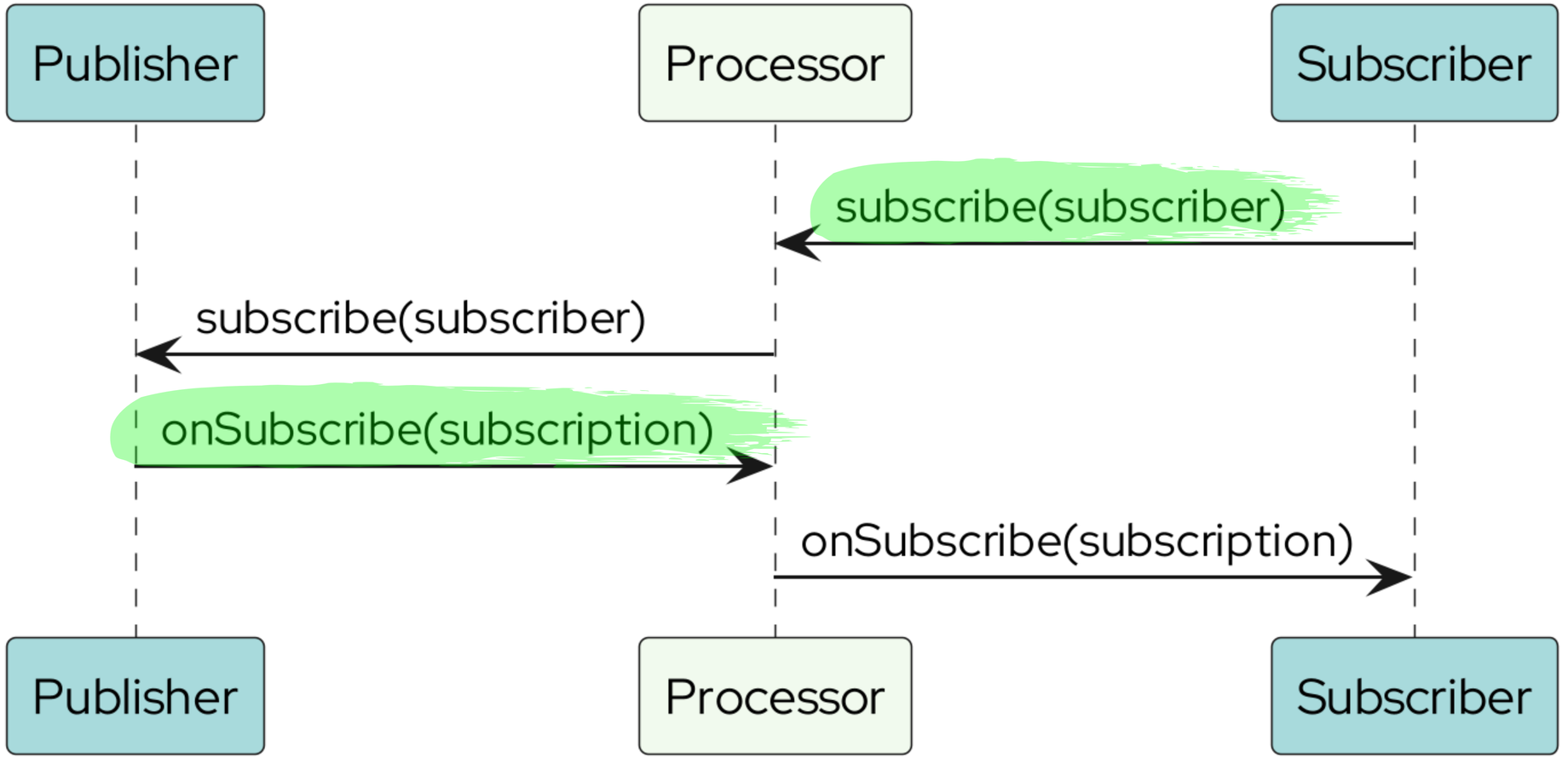
Ok, 4 interfaces. Must be easy, right?

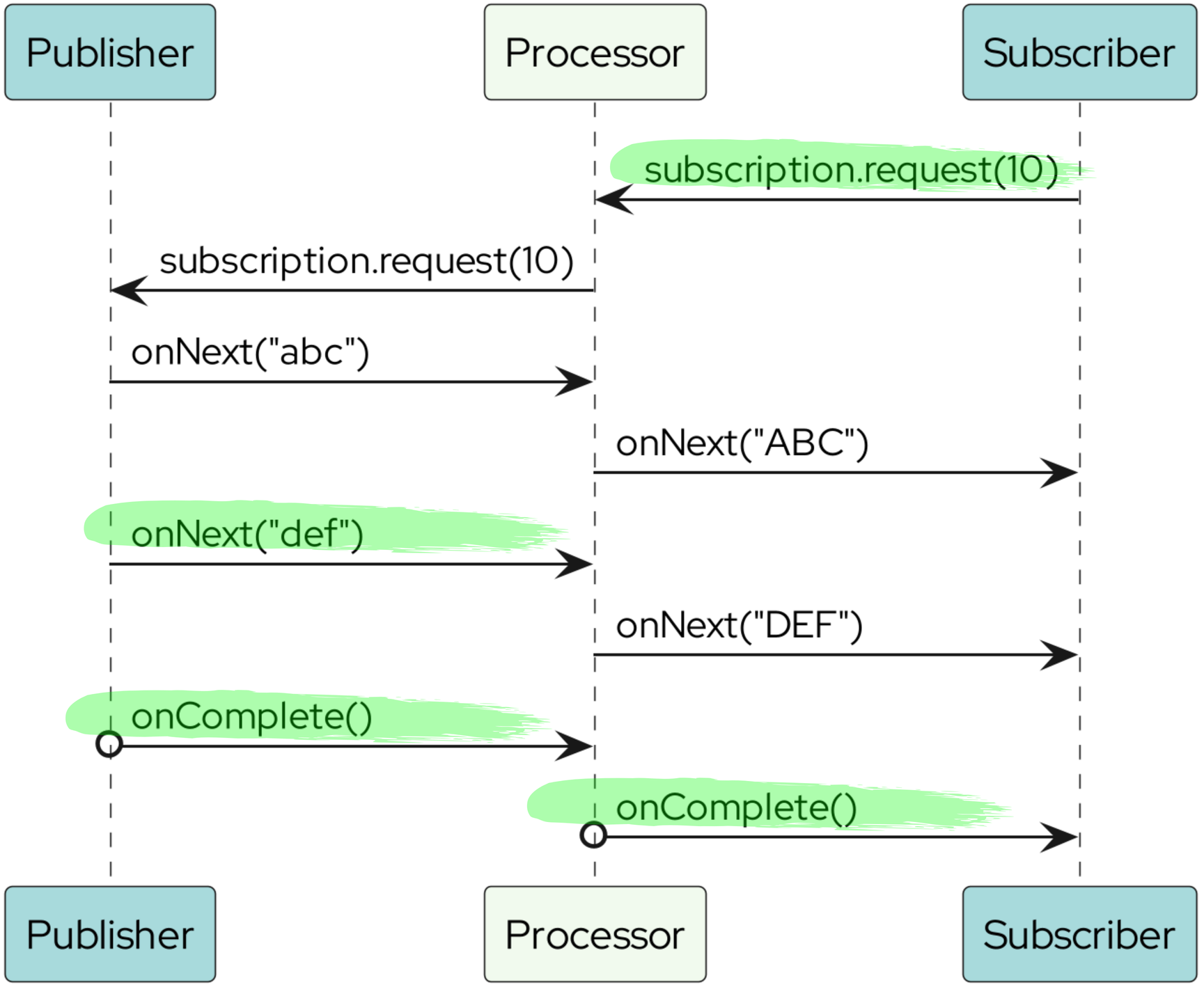
`Publisher<T>`

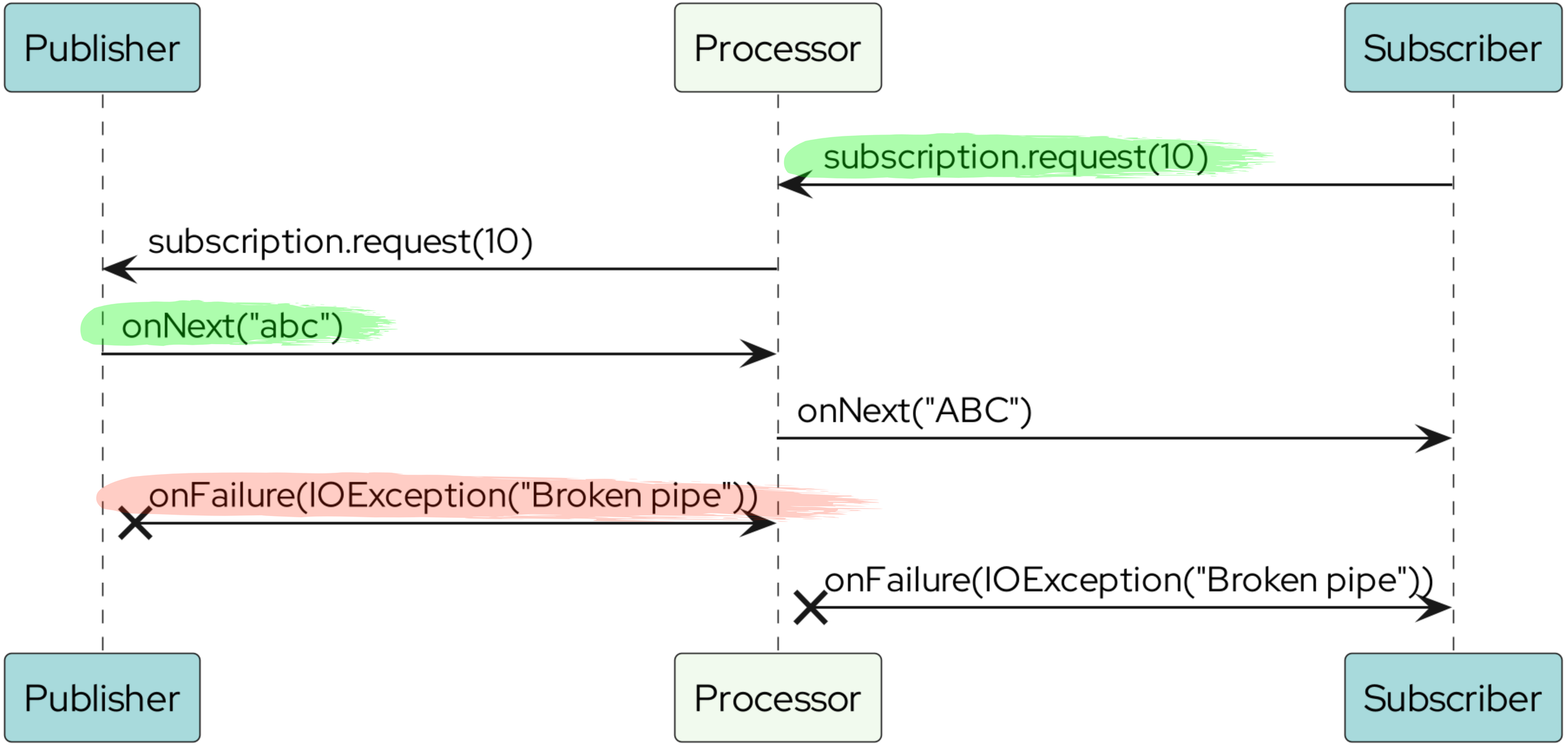
`Subscriber<T>`

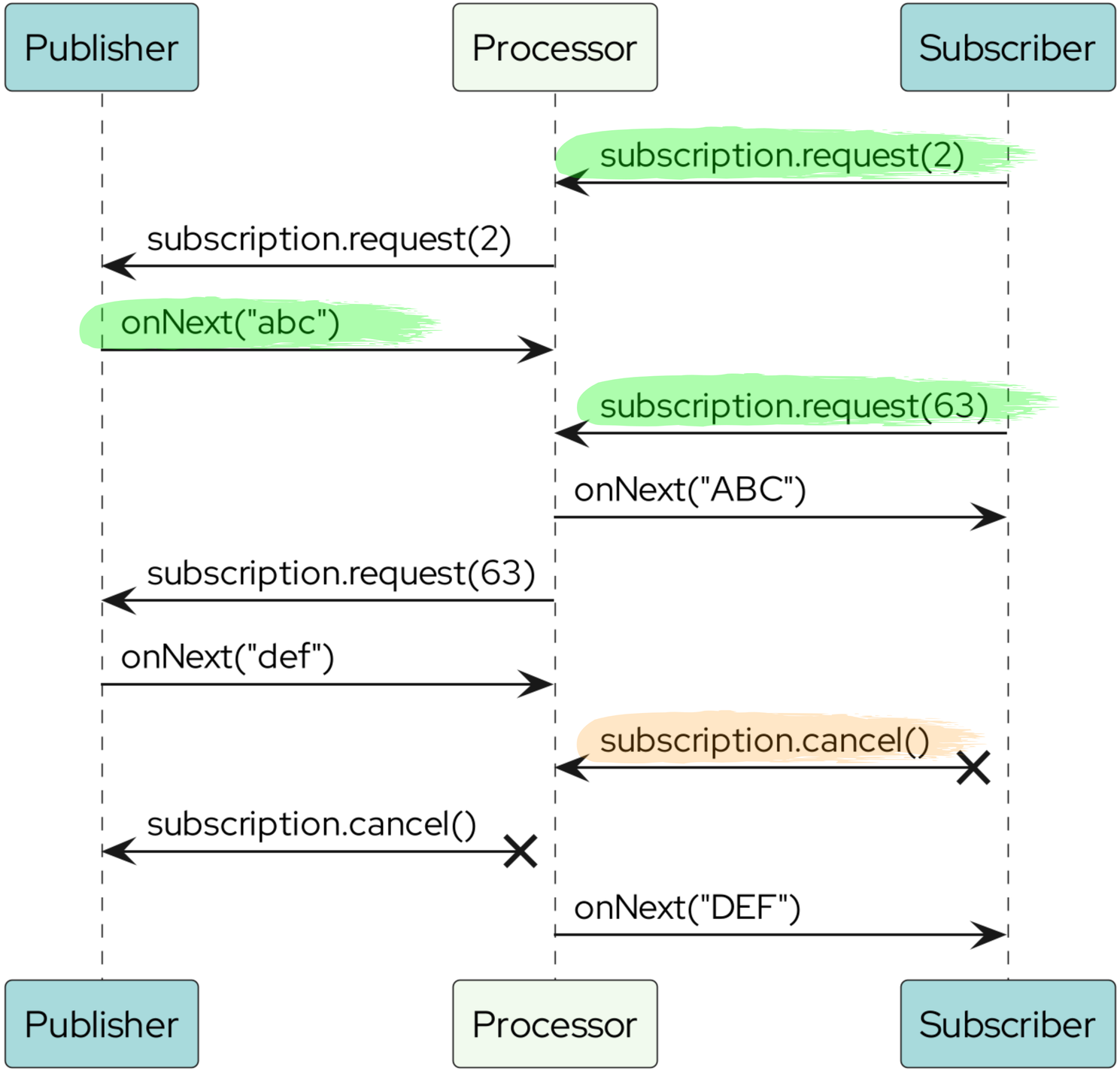
`Processor<T,U>`

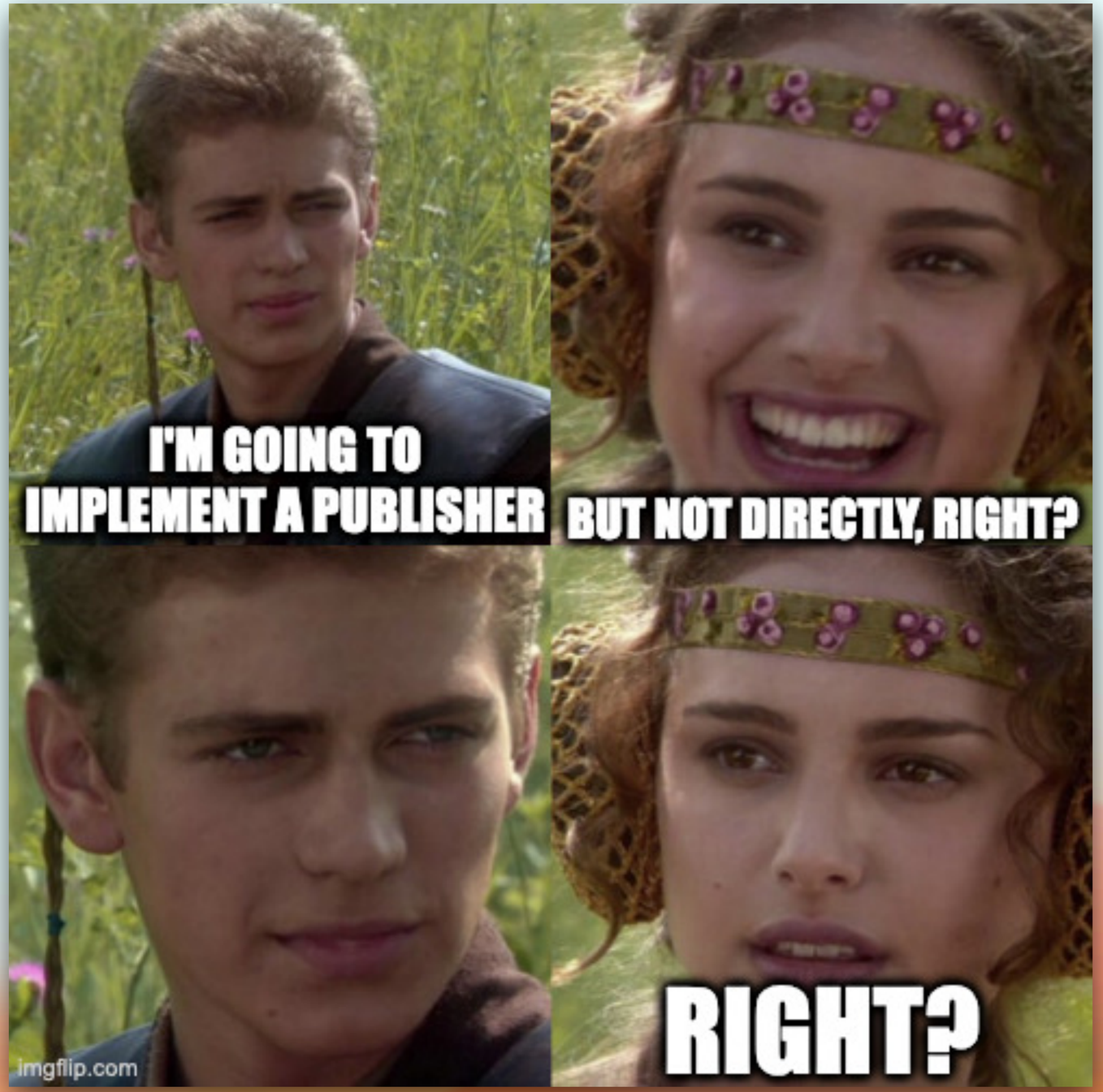
`Subscription`


Publisher
Processor
Subscriber
subscribe(subscriber)
subscribe(subscriber)
onSubscribe(subscription)
onSubscribe(subscription)
Publisher
Processor
Subscriber


Publisher
Processor
Subscriber
subscription.request(10)
subscription.request(10)
onNext("abc")
onNext("ABC")
onNext("def")
onNext("DEF")
onComplete()
onComplete()
Publisher
Processor
Subscriber


Publisher
Processor
Subscriber
subscription.request(10)
subscription.request(10)
onNext("abc")
onNext("ABC")
onFailure(IOException("Broken pipe"))
onFailure(IOException("Broken pipe"))
Publisher
Processor
Subscriber


Publisher
Processor
Subscriber
subscription.request(2)
subscription.request(2)
onNext("abc")
subscription.request(63)
onNext("ABC")
subscription.request(63)
onNext("def")
subscription.cancel()
subscription.cancel()
onNext("DEF")
Publisher
Processor
Subscriber


I'M GOING TO IMPLEMENT A PUBLISHER
BUT NOT DIRECTLY, RIGHT?
RIGHT?
imgflip.com

Mutiny Zero

– minimal helper libraries to make publishers

(44 Kilobytes JAR)

Mutiny Zero

RS <-> Flow
Mutiny Zero Adapters

Vert.x -based
stream publishers

Past Java 6 legacy – Transitioning to JDK Flow

Mutiny 2

Quarkus 3

Hibernate Reactive 2

Reactive Messaging

Analysing the Performance and Costs of Reactive Programming Libraries in Java

Julien Ponge
jponge@redhat.com
Red Hat
Lyon, France

Arthur Navarro
arnavarr@redhat.com
Red Hat
Villeurbanne, France

Clément Escoffier
cescoffi@redhat.com
Red Hat
Valence, France

Frédéric Le Mouël
frederic.le-mouel@insa-lyon.fr
Univ Lyon, INSA Lyon, Inria, CITI, EA3720
Villeurbanne, France

Abstract

Modern services running in cloud and edge environments need to be resource-efficient to increase deployment density and reduce operating costs. Asynchronous I/O combined with asynchronous programming provides a solid technical foundation to reach these goals. Reactive programming and reactive streams are gaining traction in the Java ecosystem. However, reactive streams implementations tend to be complex to work with and maintain. This paper discusses the performance of the three major reactive streams compliant libraries used in Java applications: RxJava, Project Reactor, and SmallRye Mutiny. As we will show, advanced optimization techniques such as operator fusion do not yield better performance on realistic I/O-bound workloads, and they significantly increase development and maintenance costs.

CCS Concepts: • **Software and its engineering**;

Keywords: reactive programming, reactive streams, java, benchmarking

ACM Reference Format:
Julien Ponge, Arthur Navarro, Clément Escoffier, and Frédéric Le Mouël. 2021. Analysing the Performance and Costs of Reactive Programming Libraries in Java. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '21), October 18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486605.3486788

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
REBLS '21, October 18, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9108-5/21/10…$15.00
https://doi.org/10.1145/3486605.3486788

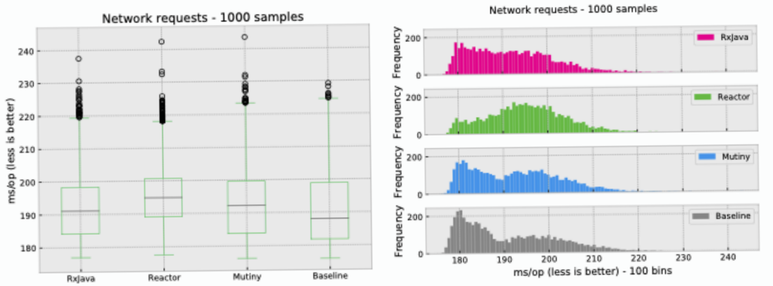
1 Introduction

Modern applications are made by composing distributed services that are developed in-house or taken from off-the-shelve third-party vendors. Services are being increasingly deployed and operated in *Kubernetes* clusters in *cloud* and *edge* environments[4]. Micro-services recently became a popular architecture style where each service has a tight functional scope, has data ownership and has its own release life-cycle. Such services can be scaled up and down in a fine-grained fashion to respond to fluctuating workloads. For instance, a service may have 12 instances running at peak time during the day and 0 at night when there is no traffic. It is increasingly important to maximize *deployment density* in such environments where costs are driven by resource usage[21], hence deploy *resource-efficient* services[5].

One of the key ingredients for resource efficiency is to move away from traditional software stacks where each network connection is associated with a thread, and where I/O operations are blocking[8]. By moving to asynchronous I/O, one can multiplex multiple concurrent connection processing on a limited number of threads[7, 13], but this requires abandoning familiar imperative programming constructs.

There is a great interest in the Java ecosystem for embracing asynchronous I/O and asynchronous programming, with *reactive streams*[16] playing a pivotal role as a foundation for higher-level programming models and middleware[9, 17, 19, 20]. Still, reactive streams implementations such as *RxJava*[19], *Reactor*[20] and *Mutiny*[17] are complex. The maintenance of such libraries is expensive due to the complexity of the reactive streams protocol. As reactive is often associated with hopes for better performance, most libraries introduced complex optimizations. By contrast, the Mutiny library took a different approach by not including any complex optimization, resulting in more straightforward, easier to maintain code. A natural question arises: how does Mutiny perform in comparison to the other optimized libraries?

This paper discusses the performance of the 3 major reactive programming libraries used in Java software stacks:



(a) Box plot **(b)** Histograms

Figure 8. Network requests processing performance.

The *Reactive Manifesto* focuses on the design of reactive *systems*[5], that is, systems that remain responsive as workloads increase and/or as failures arise[7, 9, 13]. Reactive systems are based on asynchronous message passing for elasticity and resilience purposes, hence reactive streams are key to their design and implementation. Reactive systems, reactive streams and reactive programming are different facets of the modern distributed framework stacks as applications need to face scalability[8] and resiliency challenges[2, 7, 13, 18, 22].

We designed our own benchmark suite because there doesn't exist any for either reactive programming libraries in Java or reactive streams implementations. The wide diversity and discrepancy of operator combinations across the libraries make the creation of a standard benchmark suite difficult. We do not claim that the benchmark suite that we designed is perfect: we selected a set of representative operators based on our experience of reactive programming usage in the field.

The *Renaissance* benchmark suite addresses parallel applications[14], and it contains a benchmark called *rx-scrabble* that uses an old version of RxJava 1 and file system operations. The scope of this benchmark suite is too distant for being useful in the context of this paper, and the file processing benchmark above already performs a fair comparison of the performance of RxJava, Reactor and Mutiny in presence of I/O operations.

5 Conclusion

This paper compared the performance of 3 reactive programming libraries for Java that comply with the reactive streams specification: RxJava, Reactor and Mutiny. To that purpose, we developed a suite of custom micro-benchmarks.

We first measured the performance of single operations and stream reactive types in purely CPU-bound micro-benchmarks. We benchmarked commonly used individual operators: transforming values, selecting values, chaining with individual and stream-producing operations. We also benchmarked pipelines with multiple operators, as this is more representative of how reactive libraries are being used in applications.

RxJava performs the best for single operation reactive types, but it does not try to replicate the reactive streams protocol as Mutiny and Reactor do. We found a bug causing out-of-memory exhaustion with RxJava `flatMapSingle` operator. Reactor ended up slowest in benchmarks, with Mutiny being half-way to the performance of RxJava on multiple operators pipelines. Single operation reactive types are important as they model frequently-used asynchronous operations such as database insert queries or acknowledging messages. In fact, a typical HTTP-exposing micro-service uses single asynchronous operations more often than it has to deal with streams.

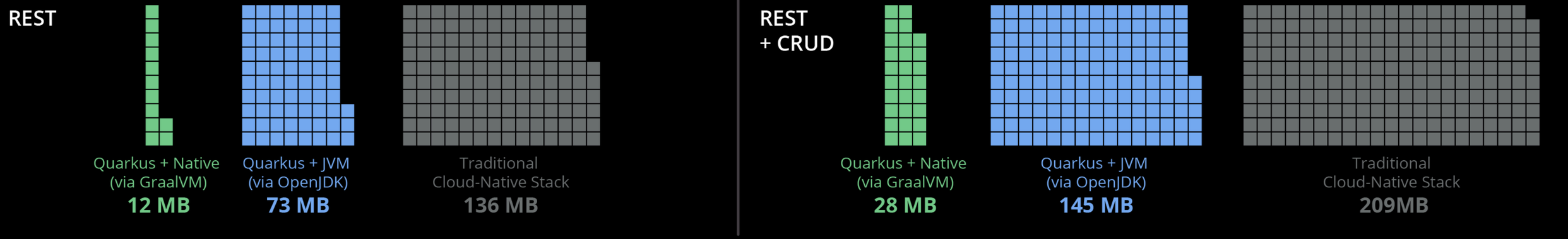
Both Reactor and RxJava have similar performance on stream operations, and outperform Mutiny as soon as chaining re-subscription is involved (e.g., `transformToUni`). In the multiple operations reactive pipeline benchmark, Mutiny ended up 13% slower than RxJava and Reactor. Individual operation benchmarks show that the performance of Mutiny is comparable to that of the other libraries for direct data transformation and selection. RxJava and Reactor share lots of history and operators internals. Their performance in CPU-bound cases can be explained by operator fusing and pre-fetching techniques, but at the cost of more complex code bases, as Table 1 shows. We could not assess if the *"relaxation"* of the reactive streams protocol between RxJava

QUARKUS

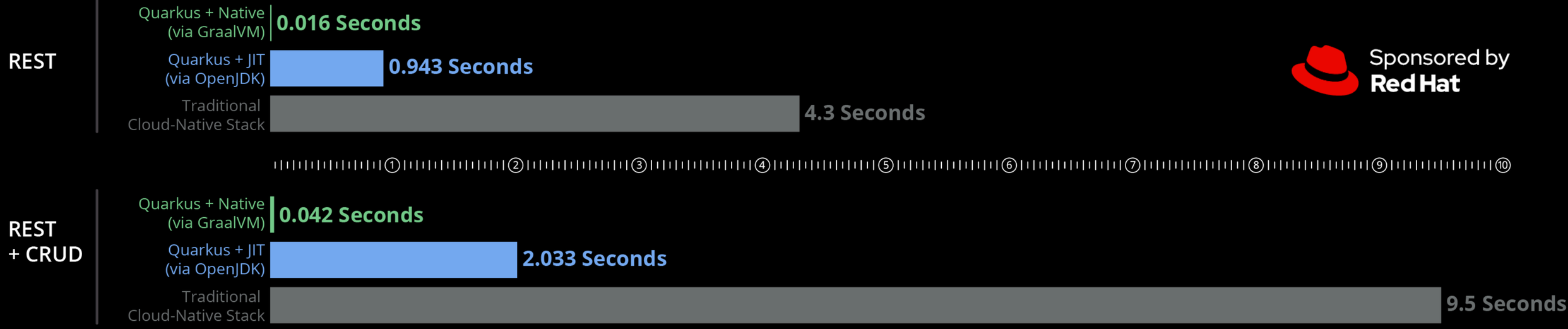
Netty HIBERNATE + many more!
VERT.X MUTINY! Infinispan

Memory (RSS) in Megabytes*

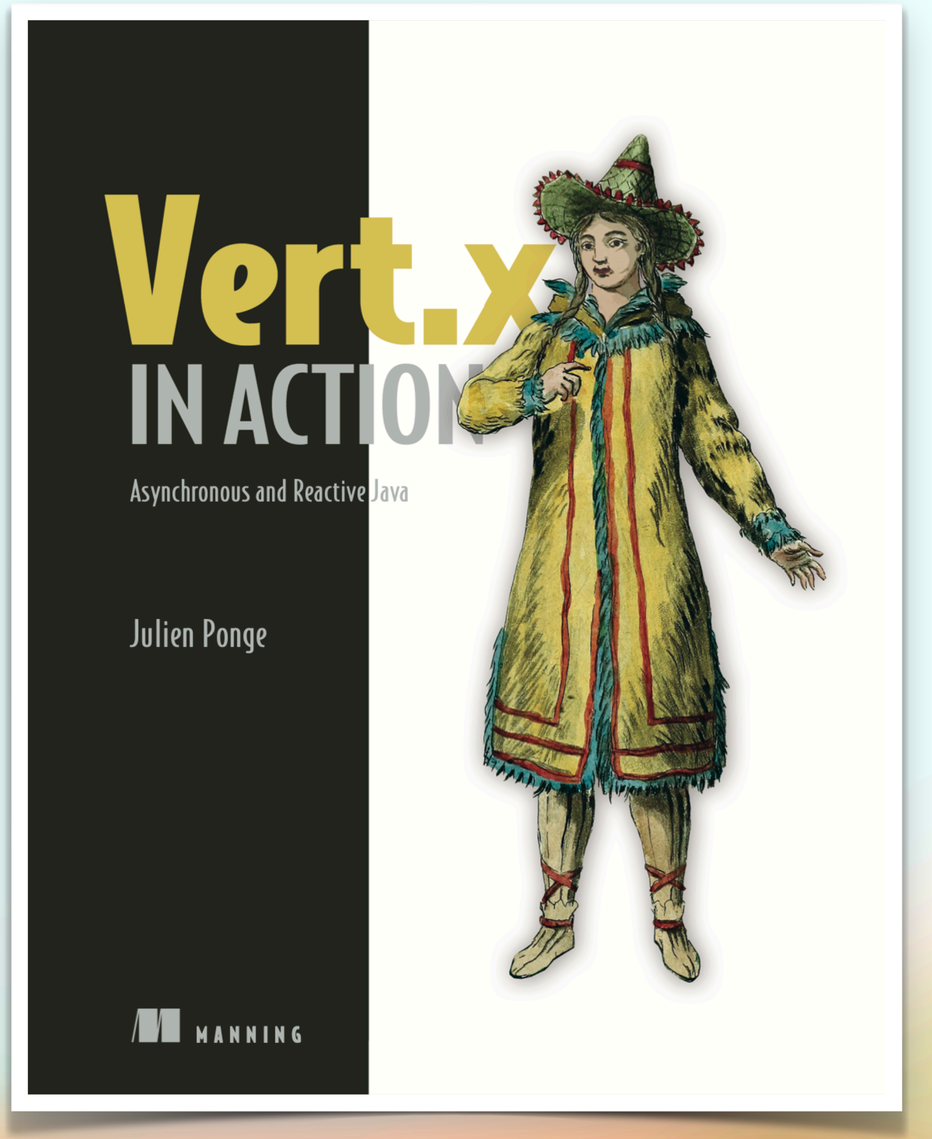
*Tested on a single-core machine



BOOT + First Response Time



Sponsored by Red Hat



Q&A

Volcamp
L'éruption des Techs

Julien Ponge
Red Hat