Reactive

Performance

About Me

@OlehDokuka

- Software Engineer focused on distributed systems development, adopting Reactive Manifesto and Reactive Programming techniques
- Open source geek, active contributor of Project Reactor / RSocket
- Author of the book "Reactive Programming in Spring 5”
- Achieved 4-times better performance by tuning Reactor for RSocket Project

Statements

About Reactive And Non-Blocking

Statements 1

Reactive Non-Blocking is way faster than imperative blocking

Responses per second at 20 queries per request, Dell R440 Xeon Gold + 10 GbE (5 tests)

Rnk	Framework	Performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
1	spring-webflux-pgclient	31,522	100.0% (67.1%)	0	Ful	Jav	Nty	Non	Lin	Pg	Lin	Mcr	Rea
2	spring-webflux-jdbc	21,889	69.4% (46.6%)	0	Ful	Jav	Nty	Non	Lin	Pg	Lin	Mcr	Rea
3	spring	20,313	64.4% (43.2%)	0	Ful	Jav	tom	Non	Lin	Pg	Lin	Mcr	Rea

Source: TechEmpower: Web Frameworks Benchmark - Round 18: Multiple queries

Statements 2

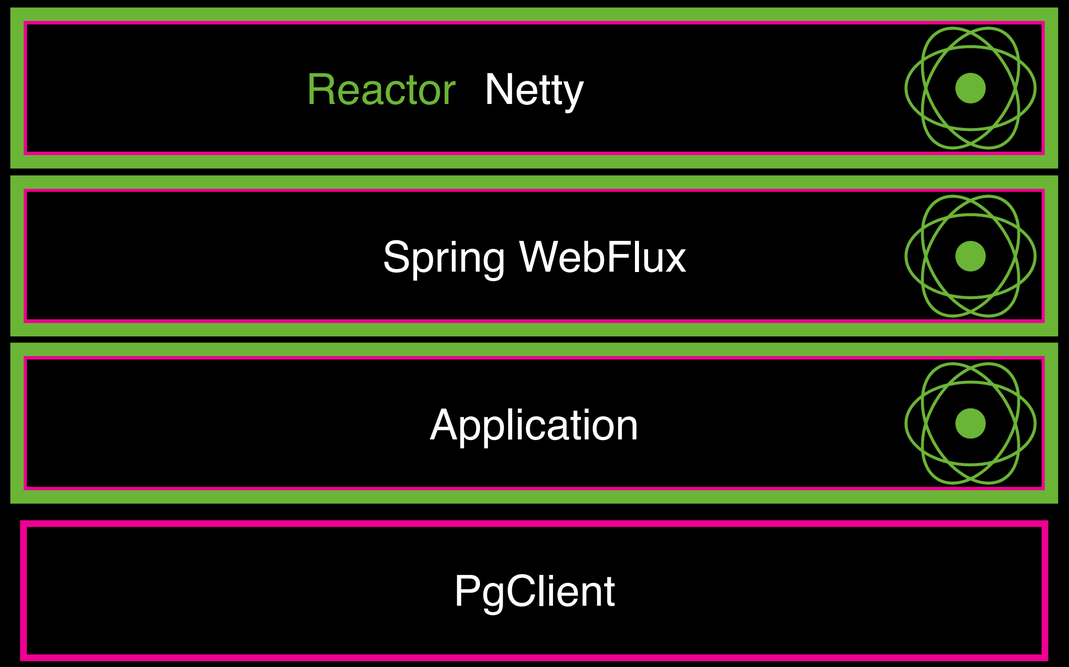
However: Async Non-Blocking is faster than Reactive

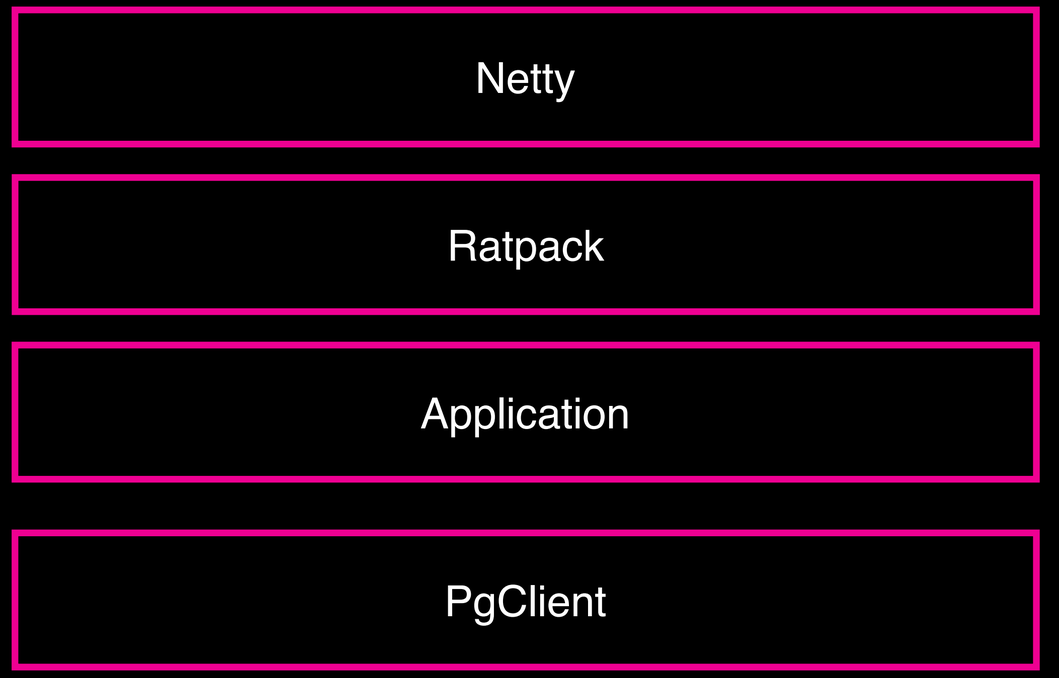
Responses per second at 20 queries per request, Dell R440 Xeon Gold + 10 GbE (11 tests)

Rnk	Framework	Performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
1	ratpack-pgclient	42,940	100.0% (91.4%)	0	Mcr	Jav	Nty	Non	Lin	Pg	Lin	Raw	Rea
2	micronaut	33,173	77.3% (70.6%)	0	Mcr	Jav	Nty	Non	Lin	Pg	Lin	Raw	Rea
3	spring-webflux-pgclient	31,522	73.4% (67.1%)	0	Ful	Jav	Nty	Non	Lin	Pg	Lin	Mcr	Rea

Source: TechEmpower: Web Frameworks Benchmark - Round 18: Multiple queries

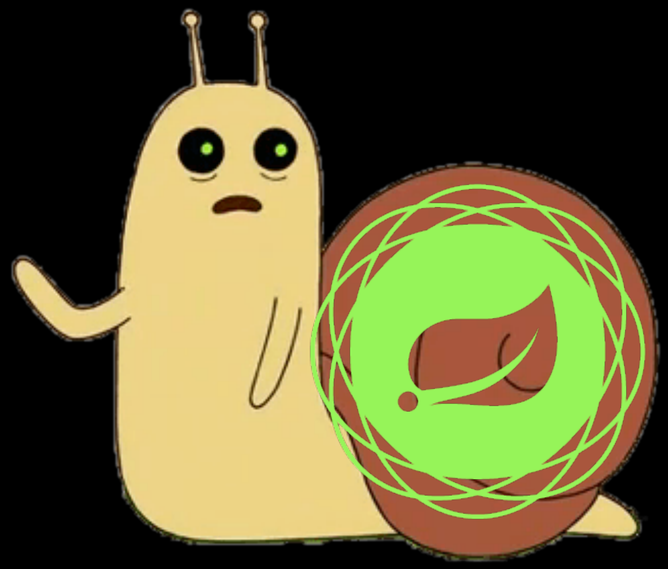
Statements 2





Statements 3

Reactive-Streams are NOT that fast



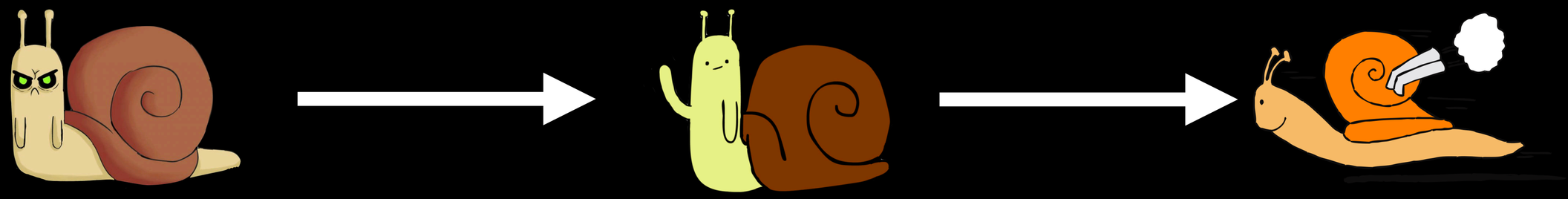
Statements 4

However: Reactive states for simplicity

Operators offered by Reactive Libraries often reduce what was once an elaborate challenge into a few lines of code

Source: ReactiveX.io

Today’s Outline



Problematic

How To

Performance

A few words on

Performance Measurement

Performance Measurement

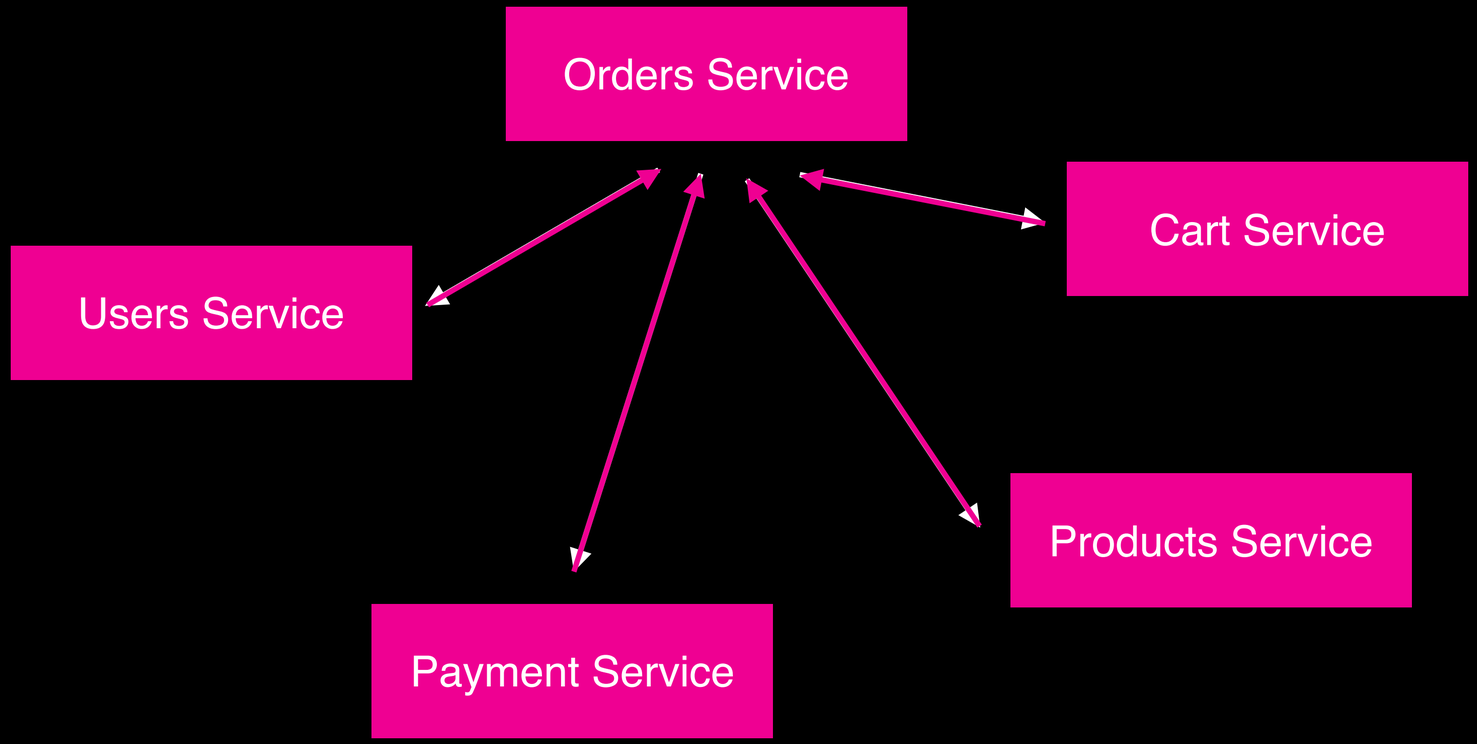
- We will NOT measure network
- We will measure Reactive Code performance using JMH
- We will look at things like Throughput / Latency

The Store Project

For Measurements

The Store Project: Execution Flow

- Check User Auth
- Resolve Current Cart
- Resolve Products Info
- Make a Payment



Code Session

Where the overhead comes from

Reactive-Streams
Lifecycle

Phase 1: Assemble

```
Flux.range(0, 100)
    .map(Object::toString)
    .filter(s -> s.length() > 2)
    .take(5)
```

```
var fR = new FluxRange(0, 100);
var fM = new FluxMap<>(fR, …);
var fF = new FluxFilter<>(fM, …);
var fT = new FluxTake<>(fF, 5);
```

We decorate **`Publisher`** into **`Publisher`** into **`Publisher`** …

Phase 2: Subscription

```
Flux.range(0, 100)
    .map(Object::toString)
    .filter(s -> s.length() > 2)
    .take(5)
    .subscribe(subscriber)
```

```
final class FluxRange<T> ... {

  void subscribe(Subscriber actual) {
    actaul.onSubscribe(
      new RangeSubscription<>(actual,
    )
  }
}
```

Stacktrace

```
subscribe:53, FluxRange (reactor.core.publisher)
subscribe:59, FluxMap (reactor.core.publisher)
subscribe:53, FluxFilter (reactor.core.publisher)
subscribe:56, FluxTake (reactor.core.publisher)
main:14,
```

Phase 2: Subscription

```
Flux.range(0, 100)
    .map(Object::toString)
    .filter(s -> s.length() > 2)
    .take(5)
    .subscribe(subscriber)
```

```
Subscriber#onSubscribe(Subscription)
```

Phase 3: Runtime

```
Flux.range(0, 100)
    .map(Object::toString)
    .filter(s -> s.length() > 2)
    .take(5)
    .subscribe(subscriber)
```

```
Subscriber#onComplete()
```

```
#request(5)Subscription
```

Phases: Summary

- At least 2 Objects Allocation per Operator

Phases: Summary

- At least 2 Objects Allocation per Operator - Means more GC Pauses

Phases: Summary

- At least 2 Objects Allocation per Operator - Means more GC Pauses
- Deeeeeeeep Stacktrace

Phases: Summary

Stacktrace

Runtime

```
onNext:158, BaseSubscriber (reactor.core.publisher)
onNext:122, FluxTake$TakeSubscriber (reactor.core.publisher)
onNext:107, FluxFilter$FilterSubscriber (reactor.core.publisher)
onNext:213, FluxMap$MapConditionalSubscriber (reactor.core.publisher)
request:107, FluxRange$RangeSubscription (reactor.core.publisher)
request:281, FluxMap$MapConditionalSubscriber (reactor.core.publisher)
request:179, FluxFilter$FilterSubscriber (reactor.core.publisher)
request:154, FluxTake$TakeSubscriber (reactor.core.publisher)
```

Subscription

```
onSubscribe:146, BaseSubscriber (reactor.core.publisher)
onSubscribe:99, FluxTake$TakeSubscriber (reactor.core.publisher)
onSubscribe:79, FluxFilter$FilterSubscriber (reactor.core.publisher)
onSubscribe:185, FluxMap$MapConditionalSubscriber (reactor.core.publisher)
subscribe:68, FluxRange (reactor.core.publisher)
subscribe:59, FluxMap (reactor.core.publisher)
subscribe:53, FluxFilter (reactor.core.publisher)
subscribe:56, FluxTake (reactor.core.publisher)
```

Assembly

```
main:14,
```

Phases: Summary

Compilation Limit Options

Table 15-4 summarizes the compilation limit options, which determine how much code is compiled).

Table 15-4 Compilation Limit Options

Option	Default	Description
`MaxInlineLevel`	9	Maximum number of nested calls that are inlined
`MaxInlineSize`	35	Maximum bytecode size of a method to be inlined
`MinInliningThreshold`	250	Minimum invocation count a method needs to have to be inlined
`InlineSynchronizedMethods`	true	Inline synchronized methods

https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/codecache.htm#BABGGHJE

Phases: Summary

- At least 2 Objects Allocation per Operator - Means more GC Pauses
- Deeeeeeeep Stacktrace - Means less efficient C2 and less Inlinings
- Every `request(N)` invocation leads to an additional volatile write

Phases: Summary

- At least 2 Objects Allocation per Operator - Means more GC Pauses
- Deeeeeeeep Stacktrace - Means less efficient C2 and less Inlinings
- Every `request(N)` invocation leads to an additional volatile write - Means more expensive CPU instructions

Rule 1

Do LESS operators

Or Make Imperative Great Again

Code Session

Rule 1: Summary

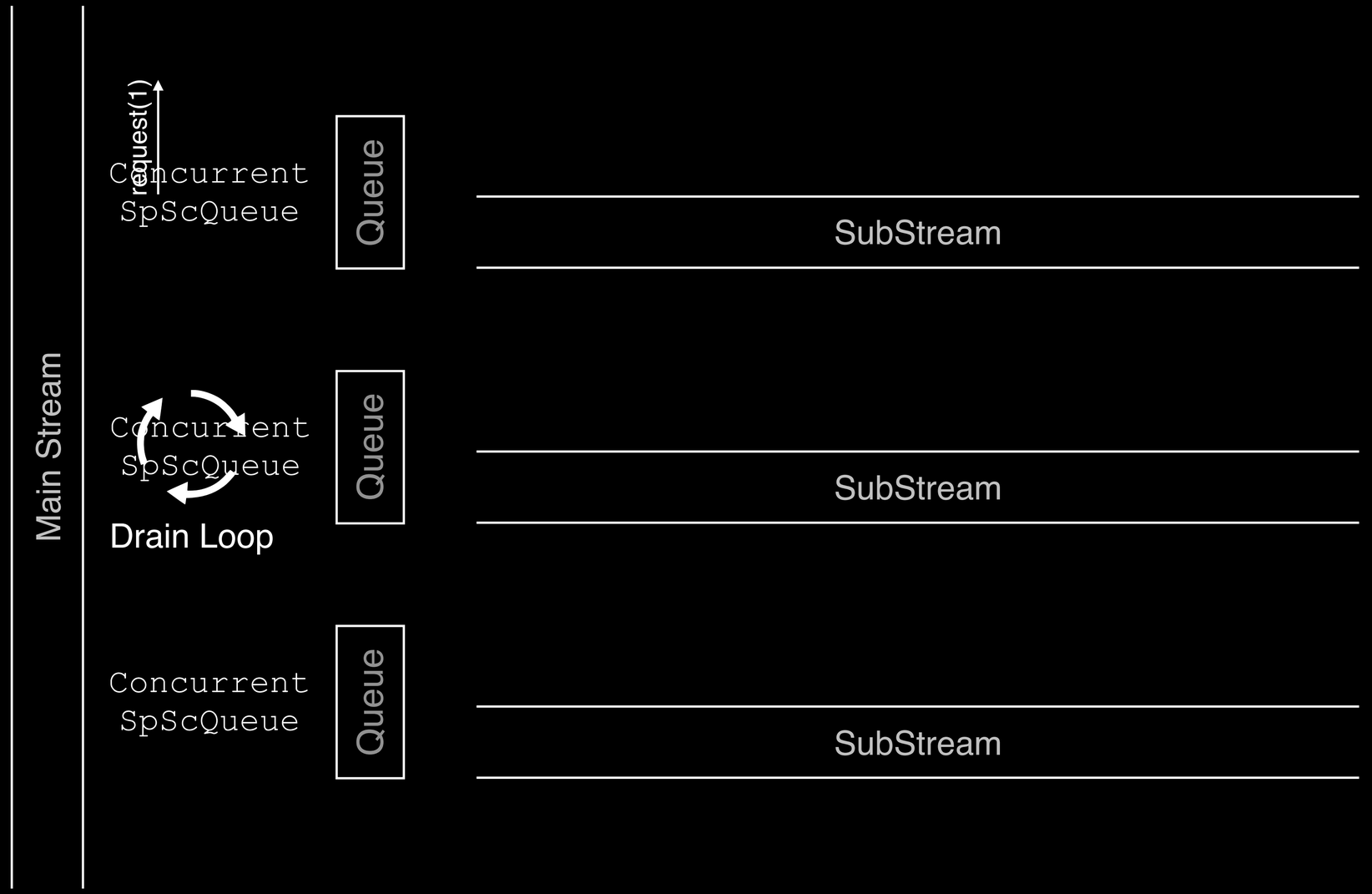
- Reduce number of plain operators like map / filter / doOnXXX
- Replace trailing map + filter -> handle

Reactive-Streams
Heavy Operators

Reactive Streams: Heavy Operators

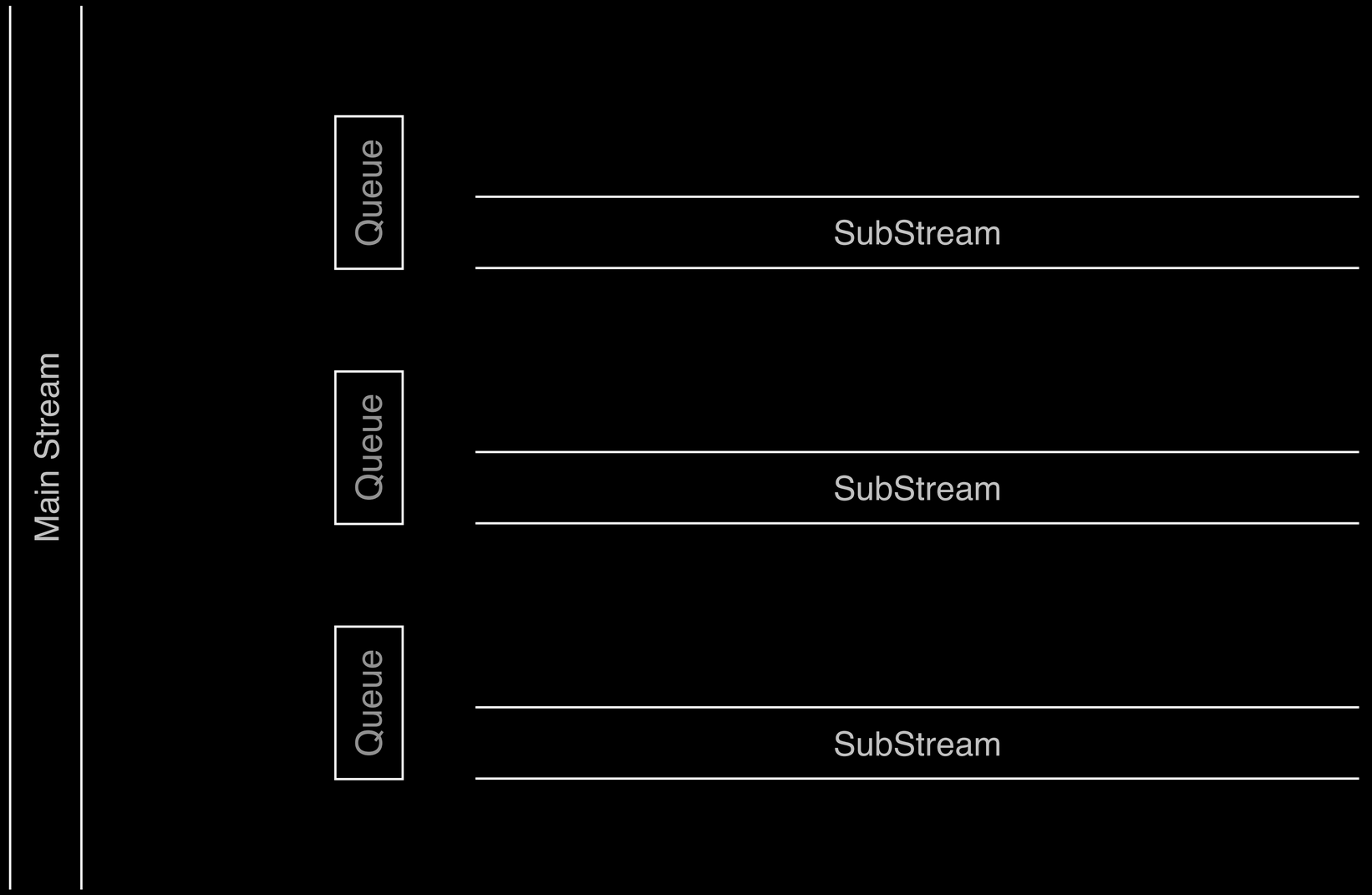
Reactive Streams: Heavy Operators

- FluxFlatMap



Reactive Streams: Heavy Operators

- FluxFlatMap
- FluxGroupBy



Reactive Streams: Heavy Operators

- FluxFlatMap
- FluxGroupBy

- FluxPublishOn

Reactive Streams: Heavy Operators

- FluxFlatMap
- FluxGroupBy

- FluxPublishOn

Concurrent SpScQueue

- FluxConcatMap

Heavy Operators: Summary

- FluxFlatMap - N Queues per Streams
- FluxGroupBy

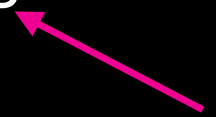
(Max N = Max Concurrency)

- FluxPublishOn

- FluxConcatMap

Heavy Operators: Summary

- FluxFlatMap - N Queues per Streams + volatile read/write per Element
- FluxGroupBy - N Queues per Streams



(Max N = Unlimited)

- FluxPublishOn

- FluxConcatMap

Heavy Operators: Summary

- FluxFlatMap - N Queues per Streams + volatile read/write per Element
- FluxGroupBy - N Queues per Streams

- FluxPublishOn - volatile read/write per Element

- FluxConcatMap

Heavy Operators: Summary

- FluxFlatMap - N Queues per Streams + volatile read/write per Element
- FluxGroupBy - N Queues per Streams

- FluxPublishOn - volatile read/write per Element

- FluxConcatMap - one extra Queue Object

Rule 2

Avoid flatMap

Whenever It Possible

Code Session

Rule 2: Summary

- Handle Errors with Handle (flatMap + try/catch -> handle)
- Replace FlatMap with ConcatMap for synchronous sub streams

Rule 3

Know Your Tool

Whenever It Possible

Code Session

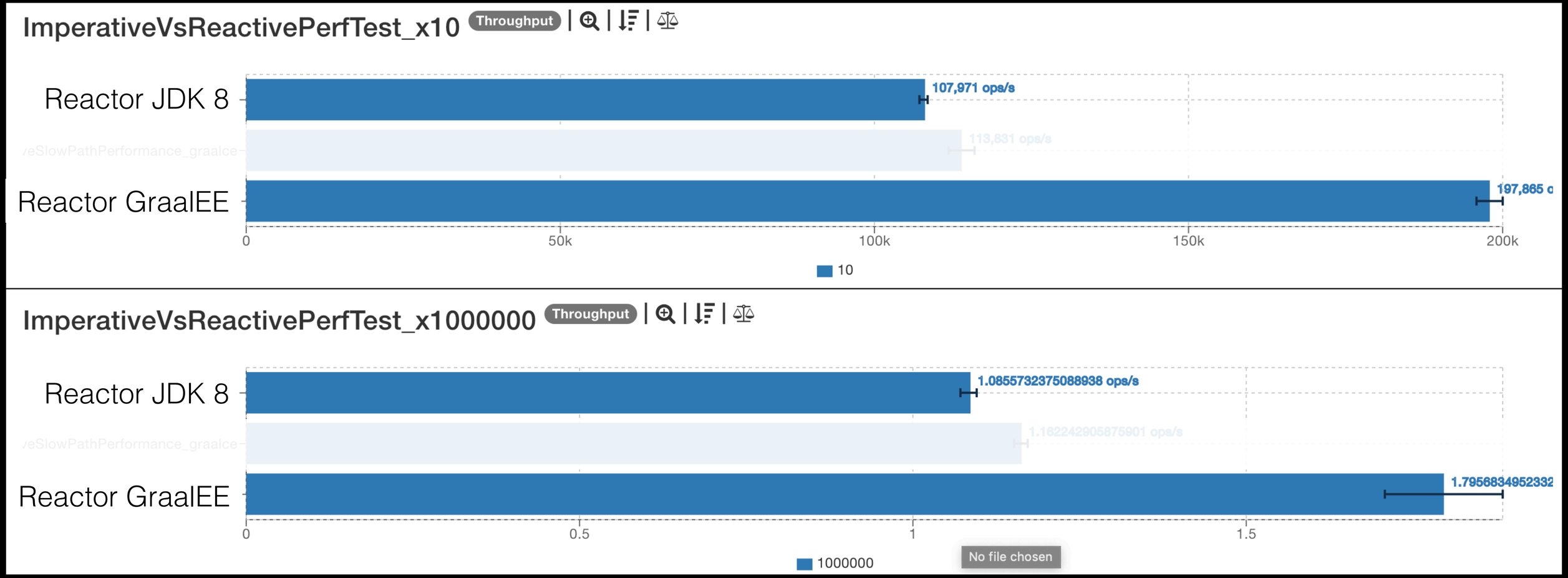
Rule 3: Summary

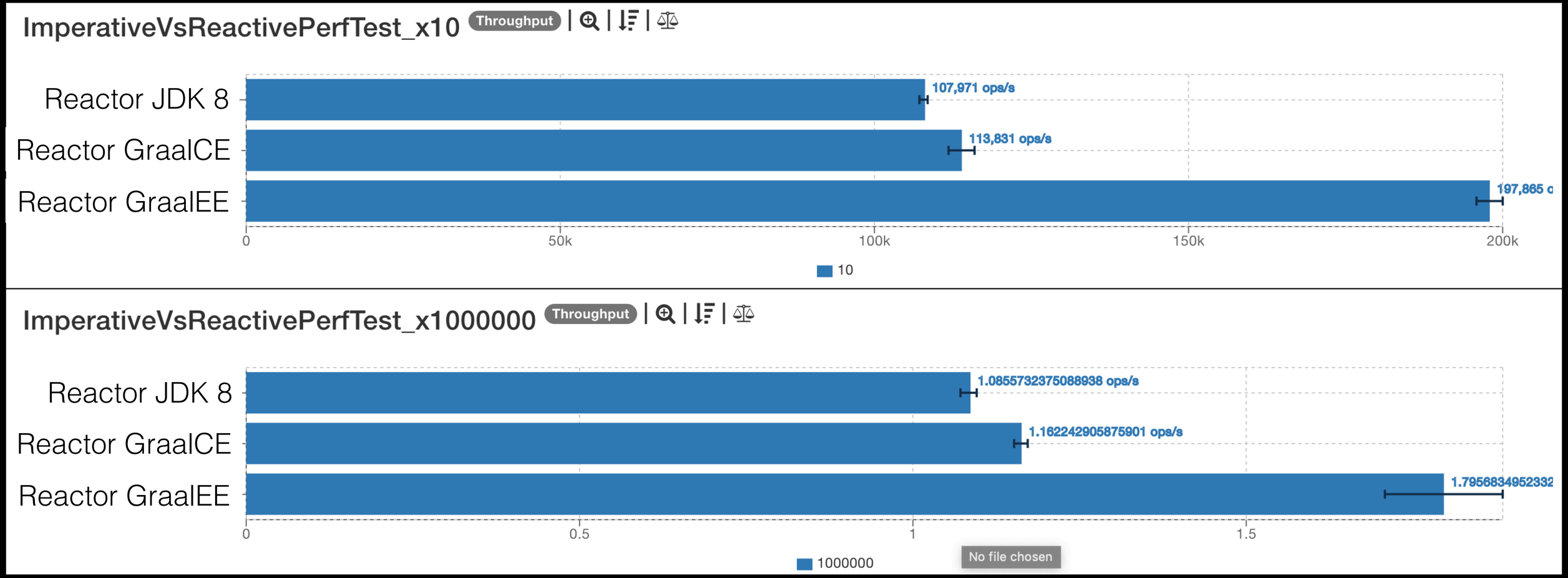
- FlatMap / ConcatMap / PublishOn /
- Queues can give different Queue impl with different performance characteristics
- Reduce number of request(N) by tuning prefetch params

Rule 4

Use Shiny Graal

Whenever It Possible


ImperativeVsReactivePerfTest_x10
Throughput
Reactor JDK 8
107,971 ops/s
eSlowPathPerformance_graalce
113,831 ops/s
Reactor GraalEE
197,865 o
0
50k
100k
150k
200k
10
ImperativeVsReactivePerfTest_x1000000
Throughput
Reactor JDK 8
1.0855732375088938 ops/s
eSlowPathPerformance_graalce
1.1622429058759O1 ops/s
Reactor GraalEE
1.7956834952332
0
0.5
1
1.5
1000000
No file chosen


ImperativeVsReactivePerfTest_x10
Throughput
Reactor JDK 8
Reactor GraalCE
Reactor GraalEE
107,971 ops/s
113,831 ops/s
197,865 o
0
50k
100k
150k
200k
10
ImperativeVsReactivePerfTest_x1000000
Throughput
Reactor JDK 8
Reactor GraalCE
Reactor GraalEE
1.0855732375088938 ops/s
1.162242905875901 ops/s
1.7956834952332
0
0.5
1
1.5
1000000
No file chosen

Takeaways

@OlehDokuka

- Avoid Operators redundancy
- Use Imperative
- Use flatMap correctly
- Tune your Reactor
- Get better inlining with Graal