Analysing the performance and costs of reactive programming libraries in Java

Julien Ponge, Arthur Navarro, Clément Escoffier, Frédéric Le Mouël

HAL Id: hal-03409277
https://inria.hal.science/hal-03409277

Submitted on 29 Oct 2021

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

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:

RxJava, Reactor and Mutiny. Experiments have been conducted across a series of CPU-bound then I/O bound microbenchmarks to measure the performance impact of the reactive pipelines built with these libraries. As we will see, clever optimization techniques do not necessarily yield better performance on realistic I/O-bound workloads, which is what reactive streams were designed for. Still, these techniques greatly increase development and maintenance costs.

2 Reactive Programming in Java

Java has long provided support for asynchronous types inspired by promises and futures[10]: `Future`, `Completable-Future` and `CompletionStage`. These types encapsulate single operations. The *java.util.stream* package deals with functional stream processing of in-memory collections, not resources with asynchronous I/O. This lead to the *reactive streams* initiative that later influenced Java to adopt the proposed interfaces as part of the standard library, albeit with a still nascent adoption.

2.1 Reactive Streams

Reactive streams is a specification and a protocol for asynchronous data stream processing. It defines a non-blocking back-pressure protocol guaranteeing that a producer throughput respects the consumer processing capacity[16].

Reactive streams are the *lingua franca* for an open and vendor-neutral asynchronous programming ecosystem in Java. For instance, both an event streaming service and a large data store can expose reactive streams compliant clients. Applications can then connect these APIs and build reactive applications enforcing end-to-end back-pressure. Of course, this requires that clients and the application code honor the reactive streams protocol and semantic.
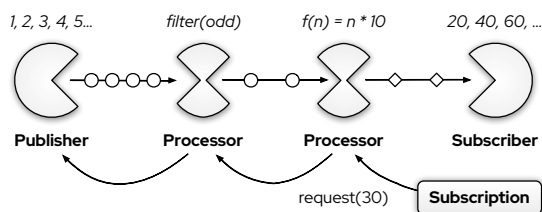

Figure 1. A sample reactive streams pipeline.

Reactive streams define an API and a protocol, captured in implementations as a library and a *technology compatibility kit* (TCK). The API defines 4 components, whose interactions are illustrated by a sample pipeline in Figure 1.

1. A *publisher* produces a potentially infinite sequence of items.
2. A *subscriber* requests a subscription from a publisher, then receives zero or more items. The subscriber can also be notified of a terminal error, and it can also receive a completion signal that marks the end of the stream.
3. A *subscription* is passed to a subscriber as a way to signal the publisher. A cancellation can be requested, after which the publisher eventually stops sending items. A subscription is also used to request a positive number of items, and the publisher should not send more than the requested amount. Unbounded requests are possible by requesting $(2^{63} - 1)$ items.
4. A *processor* is both a publisher and a subscriber, used as an intermediary operator. For example, a processor can transform and filter values, recover from errors, etc.

Publishers, subscribers and processors have to pass the reactive streams TCK. The reactive streams API is included in Java 9 and beyond in the `java.util.concurrent.Flow` interfaces, and ships as an independent set of interfaces in the `org.reactivestreams` library with Java 6 compatibility.

Reactive streams is a low-level protocol, and applications should use higher-level reactive programming libraries. Many compliant libraries exist, such as SmallRye Mutiny, RxJava, or Project Reactor. In addition to the reactive streams API and protocol, they offer a rich set of operators (e.g., transform values, chain operations, manage failures, combine streams, etc.), publisher, and subscriber implementations.

The reactive streams APIs are simple, but the protocol is complex with a broad set of rules regarding signals ordering, serialized concurrent emissions, subscription retention, cancellation, and more. The TCK offers to validate publishers, subscribers and processors against the rules of the reactive streams protocol. Writing *correct* implementations can be surprisingly difficult despite the apparent simplicity of the APIs.

2.2 Libraries

We focus our comparison on three reactive programming libraries for Java: RxJava, Reactor, and Mutiny.

2.2.1 RxJava. RxJava has historically been the first popular library to offer reactive extensions in Java. It is popular in the Android ecosystem, especially to respond to user inputs and network requests in graphical user interfaces. The popularity of RxJava in the Android space is fading as Android has now switched focus to the Kotlin programming language and (Kotlin) internal domain-specific languages for writing reactive code. RxJava usage can be found in other areas for graphical user interfaces or backend development to compose asynchronous I/O operations (e.g., the Vert.x toolkit[7]).

Here is an example with a stream (type `Flowable`) where even numbers are selected, then transformed to a `Record` object, then recorded into a database using an asynchronous `record` method:

```
flowable.filter(n -> n % 2 == 0)
        .map(Record::new)
        .flatMap(s -> record(db, s));
```

RxJava 1 did not support back-pressure and reactive streams, which were an addition of RxJava 2 and now version 3. RxJava brings the concepts from reactive extensions[11] and borrows functional programming terminology to name its operators (e.g., `map`, `flatMap`, `zip`, etc).

RxJava tries to optimize performance by using several techniques.

1. The reactive streams semantics and protocol are not always being respected but *"relaxed"* for interactions between publishers, processors and subscribers originating from the RxJava library[1].
2. Operators can be fused to reduce the number of actual operators that items traverse as they go through RxJava pipelines. The actual fusion depends on the pipeline and operators semantics: in some cases, they can be actually merged as one; in other cases, internal data structures such as queues can be shared, or thread synchronization can be removed.
3. Some operators attempt to pre-fetch data from their upstream publisher, even if their subscriber hasn't requested as much or any item yet. In theory this *may* reduce the number of signals on frequent small batches requests as an operator can cache a larger amount of pre-fetched data, but the actual gain has to be measured against concrete workloads.

2.2.2   Reactor. Reactor is a popular library whose prominent usage is in the Spring Framework community. Its history is closely related to that of RxJava 2 when it adopted the reactive streams specification in the form of the `Flowable` type. The codebases share lots of internal code, with the external APIs differing. RxJava 2 continued the API approach of version 1 and kept Java 6 compatibility to address the Android development market. Reactor focused on Java 8, especially with the support of lambdas, and offers an API reduced to 2 types: `Mono` for single-valued operations, and `Flux` for reactive streams. Reactor exhibits an API with functional programming terminology and idioms, just like RxJava, although it offers a few shortcuts and helper operators with more meaningful names (e.g., the `then` operator).

In fact the previous stream processing example based on RxJava is identical when converted to Reactor:

```
flux.filter(n -> n % 2 == 0)
    .map(Record::new)
    .flatMap(s -> record(db, s));
```

The internal design of Reactor borrows code and techniques found in RxJava, although the code bases have since diverged due to a more sustained development pace of Reactor project. Reactor hence also uses the operator fusing and pre-fetching optimizations from RxJava.

2.2.3   Mutiny. Mutiny is a more recent addition to the reactive Java ecosystem[2]. Mutiny is prominent in the Quarkus framework and Vert.x toolkit communities.

Two leading principles have guided the design of the Mutiny APIs.

1. Bring meaningful operator names for processing asynchronous events rather than borrow from the functional programming terminology.
2. Ensure API *navigability* by offering groups (e.g., `onItem()`, `onFailure()`) to limit the number of proposed methods when using an IDE completion, and show only the relevant methods for a given context (e.g., responding to a failure, responding to an item, etc).

Users of both RxJava and Reactor are overwhelmed by more than a hundred methods when they need an operator. In contrast, Mutiny users see about ten methods each time they need an operator. It is worth noting that this design will sometimes lead to more verbosity. This level of verbosity combined with modern tools (IDE completion typically) is a design choice aiming at improving readability and navigability. The previous example where even numbers from a stream are being selected, transformed into a `Record` object then saved to a database could be written as follows with Mutiny:

```
multi.select().where(n -> n % 2 == 0)
    .onItem().transform(Record::new)
    .onItem()
        .transformToUniAndMerge(s -> record(s, db));
```

The internals of Mutiny are strictly adhering to the reactive streams specification. Mutiny does not try to perform operator fusing, and, as we will see in experiments, this does not harm performance when processing I/O bound workloads (which is what reactive streams are for). It also dramatically simplifies the code base, as operator fusing strategies in RxJava and Reactor require complex protocols for operators to be merged and for internal state and synchronization sharing. Mutiny does not perform pre-fetching either. While it is possible to pass the reactive streams TCK and do pre-fetching as RxJava / Reactor do, this requires internal buffering. Prefetching also requires an internal, library-specific operator negotiation mechanism that increases the code complexity.

2.3   Source Code Metrics

Table 1 shows code metrics for RxJava, Reactor and Mutiny. The metrics have been obtained using the `scc` tool[3] and only

---

[1]See http://reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html.

[2]Disclaimer: some of the paper authors work on this project.
[3]See https://github.com/boyter/scc.

Table 1. Code metrics of the reactive programming libraries using `scc` (excluding tests and documentation).

Library	Java lines of code (LOC)	Number of files	Cyclomatic complexity (CC)	Cyclomatic complexity density (CC/KLOC)[3]
RxJava 3.0.13	100313	907	11750	117.13
Reactor 3.4.8	72858	444	13358	183.34
Mutiny 1.0.0	21177	300	2840	134.10

take the core implementations source code into account, not documentation snippets, tests and complementary modules.

We can see that Mutiny is a more straightforward code base than that of both RxJava and Reactor. Reactor and Rx-Java have comparable cyclomatic complexities, although the Reactor codebase is more condensed with about half the number of files and about 72% the number of effective lines of Java code compared to RxJava. The operator fusing and pre-fetching optimization techniques play a role in explaining the higher complexity of RxJava and Reactor compared to that of Mutiny. Codebases tend to grow bigger and more complex as features are added and bugs are discovered and fixed. Mutiny is a newer project, we must hence assume that its low complexity is in part due to that factor.

The code base of RxJava is the biggest, which can be explained by supporting a broader palette of reactive types: `Completable`, `Maybe`, `Single`, `Observable` and `Flowable`. By contrast Reactor and Mutiny expose just 2 reactive types each: `Mono` / `Flux` for Reactor and `Uni` / `Multi` for Mutiny.

RxJava and Mutiny present better maintenance productivity, whereas Reactor — with a higher cyclomatic complexity density — is more difficult to maintain either in lines of code and in complexity[3]. Mutiny is still 21% the number of lines of code of RxJava, making it a more approachable code base.

3 Benchmarking Reactive Programming Libraries

We compare the performance of Mutiny, Reactor and Rx-Java in both CPU and I/O bound settings. The benchmarks code can be found in the Git repository at https://github.com/jponge/rebls21-paper-benchmarks for review and reproducibility purposes. The library versions used in the benchmarks are that of Table 1.

3.1 Experimental Approach

The benchmarks are written using JMH[4], a sub-project of OpenJDK dedicated to helping writing *better* micro-benchmarks[6]. The Java virtual machine is an adaptive runtime that is especially efficient at optimizing code based on speculation heuristics and runtime profiles[12], so it is very easy to write incorrect benchmarks where dead-code elimination, constant folding or loop unrolling do not measure what the

benchmarks developer intended. JMH provides a harness API to write benchmarks and executes them with various techniques in generated code to defeat common JIT optimizations. We ran benchmarks multiple times because different runs may not trigger the same optimizations, and we tuned the warmup rounds so that the JIT compiler had time to reach a stable state. Using JMH alone does not provide *off-the-shelf* correctness in benchmarks, but it considerably helps to limit the likeliness of such mistakes being made when benchmarks developers are aware of them[6].

The servers used for benchmarking have Intel Xeon CPUs at 3.50GHz, 16 cores and 252GB of RAM. The operating system is using a Linux kernel 4.4.0. The Java virtual machine is an *AdoptOpenJDK*[5] distribution version 11.0.11+9, tuned to use the *Shenandoah* garbage collector[6], 1 GB of heap size and 256MB of stack size. We use Shenandoah for its short GC pauses allowing for better latency and more predictable GC.
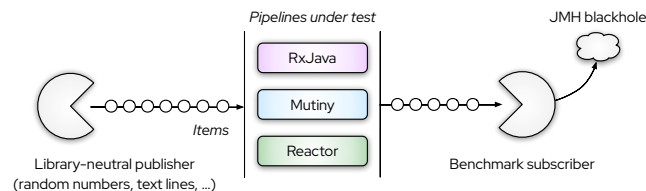
Figure 2. Benchmarking reactive libraries with library-neutral publishers and subscribers.

As we want to measure the performance of Mutiny, Reactor and RxJava, we need to ensure that the variance in benchmarks is reduced to stream processing code in pipelines that solely exercise code from these libraries. To do so, we developed publishers and a subscriber that pass the reactive streams TCK, but that are free from any code from these libraries, as illustrated in Figure 2. This ensures that all libraries are exercised with the same event sources and event consumers. It also prevents any library from doing end-to-end optimizations from its own publishers, such as operator fusing and pre-fetching, which are only possible using internal APIs that are outside of the scope of the reactive streams specification. The subscriber sends events to JMH

---

[4]See https://openjdk.java.net/projects/code-tools/jmh/

[5]See https://adoptium.net from the Eclipse Foundation.
[6]See https://wiki.openjdk.java.net/display/shenandoah/Main.

`Blackhole`, a helper class for consuming values and preventing (but not limited to) constant folding and dead-code elimination as they are the most common mistakes.

There is neither an established benchmark for reactive programming libraries in Java, nor a benchmark for reactive streams implementations. We have developed a benchmark suite split into 3 families: individual operations (CPU bound), multiple-operator pipelines (CPU bound), and I/O bound pipelines. The benchmarks first compare the performance of individual operations commonly used in reactive pipelines: transforming a value with a function, chaining with another asynchronous operation and selecting values. We then run benchmarks where pipelines perform several operations between the initial publisher and the subscriber. In either individual operations or pipelines, the initial publisher generates random numbers. These are CPU-bound benchmarks, and since reactive streams have been designed for asynchronous I/O, we also compare all libraries on representative I/O-bound workloads: composing network requests and processing text lines from a file.

3.2 Single Operation Pipelines

All libraries offer reactive types for modeling single operations: `Mono` in Reactor, `Single`, `Maybe`, `Completable` in RxJava and `Uni` in Mutiny. These types are useful for representing and composing *one-shot* operations such as sending an event to a message queue or doing a database insert. They directly compare to `CompletionStage` in Java, which is a form of *future / promise*[10].

3.2.1 Individual Operators. Here we compare the performance of transforming an event value (`map`) and chaining with another operation (`chain`) on the `Uni` (Mutiny), `Single` (RxJava) and `Mono` (Reactor) types. The benchmark can be found in the `UniIndividualOperators` class of the source code repository. We use Java `CompletionStage` (through the `CompletableFuture` concrete class) as a baseline. As said earlier, it also performs an asynchronous one-shot operation while avoiding the overhead of the reactive streams protocols. The throughput results are in Figure 3.

`CompletableFuture` provides the best throughput by nearly a factor of 2 for both types of operations, which is not surprising since it does not have the overhead of a subscription-based protocol. Here RxJava performs the best, followed by Mutiny and Reactor, which is about 2 times slower than Mutiny on the `map` operation.

3.2.2 Multiple Operators. To better appreciate the performance of the reactive libraries, we need to measure what happens on a pipeline with multiple operators. The benchmark can be found in the `SingleOperationDirectTransformation` class of the source code repository.

We start with a random number from a publisher, then convert it to an absolute value using `Math::abs`, then convert it to a hexadecimal string using `Long::toHexString`.
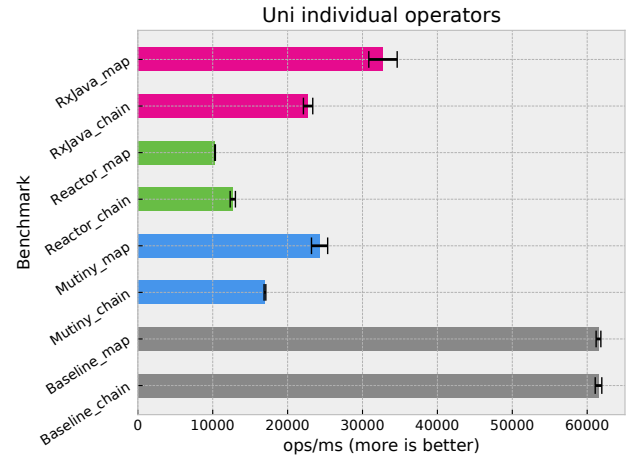

Figure 3. Single operation and individual operators performance.

We then chain the result with an operation to enclose the string in brackets. The Mutiny version of the pipeline is as follows:

```
Uni.createFrom().item(() ->
      ThreadLocalRandom.current().nextLong())
  .onItem().transform(Math::abs)
  .onItem().transform(Long::toHexString)
  .onItem().transformToUni(s ->
      Uni.createFrom().item("[" + s + "]"));
```

We again use `CompletionStage` as a baseline, but we also perform the processing in plain imperative code using direct calls to `Math::abs` and `Long::toHexString` as another baseline. The results are in Figure 4.
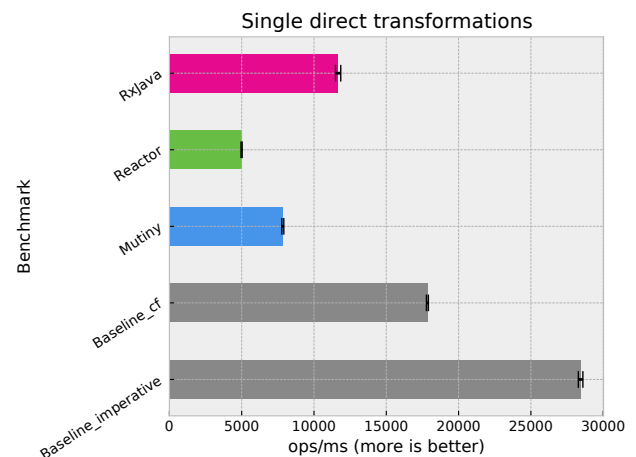

Figure 4. Single operation and multiple operators performance.

With more operations, the subscription-time overhead is less apparent against `CompletionStage`. The relative performance of the 3 reactive libraries is in line with the results on individual operators: Reactor is the slowest, RxJava is the fastest, and Mutiny sits in-between them. The plain imperative baseline also serves as a reminder that boxing values and transforming them in a monadic fashion is not free.

3.3 Event Stream Pipelines

Back-pressured stream pipelines can be constructed with Mutiny `Multi`, Reactor `Flux` and RxJava `Flowable`. They model streams of asynchronous events where back-pressure is managed through the reactive streams request protocol. An example would be receiving events from a message queue. All types define some form of overflow management strategy when items are being received, but there is no outstanding subscriber demand. For instance, items can be buffered, dropped, or an overflow can be a failure that terminates a subscription.

3.3.1 Individual Operators.
We compare the performance of transforming values (`map`), selecting values based on a predicate (`filter`), chaining with a single-valued operation (`mapToOne`), and chaining with a stream-returning operation (`mapToMany`). The benchmark can be found in the `MultiIndividualOperators` class of the source code repository. The results are in Figure 5.
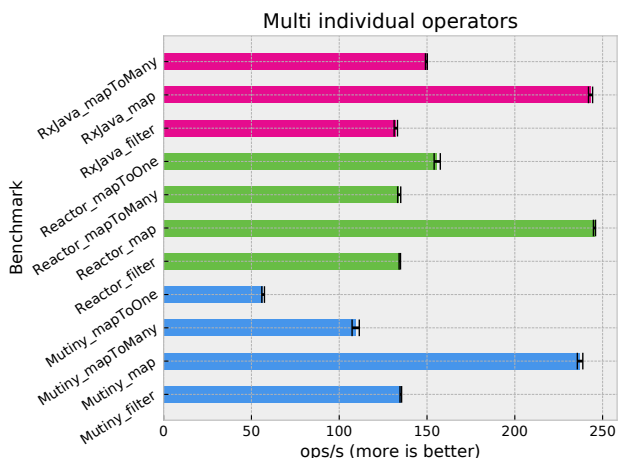

Figure 5. Event stream and individual operators performance.

There is no baseline in this benchmark to focus on just the relative performance of the libraries. The transformation and selection operators (`map` and `filter`) do not show any significant performance difference across the 3 reactive programming libraries.

Mutiny has a slower operation chaining performance compared to RxJava and Reactor for both single-valued and stream cases (`mapToOne` and `mapToMany`). We correlated this performance issue to a bottleneck in the Mutiny subscription mechanism because these operations require subscribing to nested `Uni` and `Multi` objects. This will likely be revisited and addressed in a future release of Mutiny.

Figure 5 does not show a `mapToOne` result for RxJava because the library has a bug with the `flatMapSingle` operator that causes a memory exhaustion when running the benchmarks. This is a significant problem as it is very frequent for applications to perform an asynchronous operation for each item in a stream (e.g., doing a database insert). The exact cause for that problem has yet to be identified, reported and fixed. We also see that the *"relaxed"* reactive streams protocol handling of RxJava does not result in significant performance benefits compared to Reactor.

3.3.2 Multiple Operators.
We now compare the reactive libraries on pipelines with multiple operations. The benchmark can be found in the `StreamDirectTransformation` class of the source code repository.

Pipelines start with a random number publisher, followed by a transformation to an absolute value with `Math::abs` then an odd value selection. Then for each item, we produce a stream of strings with the hexadecimal string representation of the numbers (`Long::toHexString`), followed with 1, 2 and 3 `"!"` characters. Hence a stream $(1, 2, 3, 12)$ would produce $(2, 2!, 2!!, 2!!!, c, c!, c!!, c!!!)$. The Reactor version of the pipeline is as follows:

```
Flux.from(new RandomNumberPublisher())
    .map(Math::abs)
    .filter(n -> n % 2 == 0)
    .concatMap(n -> {
        String s = Long.toHexString(n);
        return Flux.just(s, s + "!", s + "!!",
                                    s + "!!!");
    });
```

Note that we use *stream concatenation semantics* with `concatMap` to preserve ordering, which would not be the case with a `flatMap`.

We use 2 baselines: one with Java collection streams and the equivalent imperative code done in a plain imperative loop. The results are in Figure 6.

The imperative baseline is unsurprisingly the fastest, followed by Java streams that do not need a back-pressure signalling and subscription mechanism, and do not have to deal with potential multi-threaded access in that benchmark. RxJava and Reactor have the same performance, while Mutiny is around 13% slower. This can be explained by the impact of operator fusion in RxJava and Reactor as well as the inner streams subscription performance highlighted in the previous individual operations benchmark. Stream processing of in-memory data is CPU-bound, and as such Java streams are a better choice over any reactive streams based library when a declarative pipeline model is a good fit. Last but not
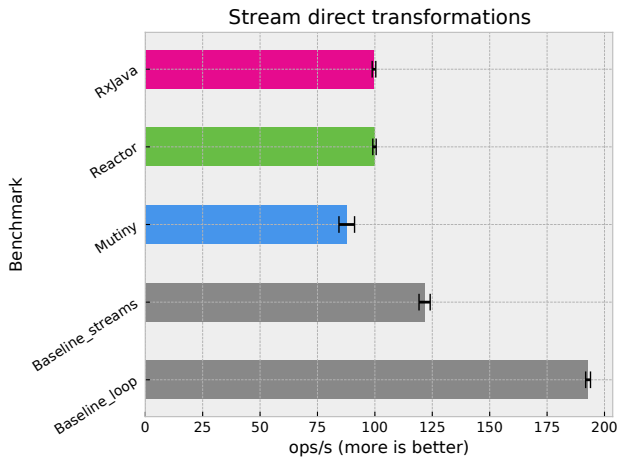
Figure 6. Event stream and multiple operators performance.

least *good old* imperative programming should always be considered as it has evidently the best performance.

3.4 I/O Bound Pipelines

The reactive streams specification was motivated by the need for programming against asynchronous I/O streams. In this section, we look at the performance of Mutiny, RxJava and Reactor with I/O bound pipelines. The first scenario performs I/O on the file system, while the second one performs network requests.

3.4.1 File Processing. The source publisher in this example reads text lines from an electronic transcript of the 19th century book *Les Misérables* by Victor Hugo (the file weights 3.2M). The benchmark can be found in the `TextProcessing` class of the source code repository.

The first operator after the publisher discards blank lines. The next one computes the number of characters in the line. We then chain with a filesystem operation that appends the characters count to a file, and to make it *asynchronous* that operation is dispatched to a worker thread pool. The last operator transforms the characters count to a string and prefixes it with an arrow. We hence have 3 simple in-memory operations (selecting and transforming values) and 2 I/O operations (reading and writing). The Mutiny version of the pipeline is as follows:

```
Multi.createFrom().safePublisher(
    new TextFileLinePublisher(source))
  .select().where(line -> !line.isBlank())
  .onItem().transform(String::length)
  .onItem().transformToUniAndConcatenate(count ->
    Uni.createFrom()
      .completionStage(appendToSink(count)))
  .onItem().transform(count -> "=> " + count);
```

We compared Reactor, RxJava and Mutiny against a baseline of the equivalent imperative code. Figure 7 shows the

results, with Figure 7a showing a box plot of the samples data, and Figure 7b showing a histogram of the processing times distribution. Neither representation shows any statistical benefit of using one reactive library or the other. The imperative code baseline is faster, but when it comes to reactive programming libraries they all exhibit the same performance figures.

3.4.2 Network Requests Processing. The previous benchmark exhibited filesystem I/O, while the next one focuses on network requests. The benchmark can be found in the `NetworkRequests` class of the source code repository.

In this benchmark, the source publisher issues 6 concurrent HTTP requests to fetch the content of the *Les Misérables* book, which is exposed by an HTTP server. The HTTP server is running on a separate server so that the HTTP requests actually go through a network, and not the loopback network interface.

Once all responses have been triggered and collected, they become a stream of 6 HTTP responses. Then, for each item, the HTTP response body (the book text) is extracted, and the final operation computes the total number of characters. The baseline uses `CompletableFuture` objects to perform 6 HTTP requests, then does the same processing as in the reactive pipelines of Mutiny, RxJava and Reactor. All pipelines delegate the HTTP request to a `Completable-Future`-returning method called `performHttpRequest`, and that uses the JDK HTTP client from Java 11 and beyond.

The Mutiny variant of the pipeline is built as follows:

```
Uni.join().all(
      Uni.createFrom().completionStage(
          this::performHttpRequest),
      // (...) times 6
  ).andFailFast()
  .onItem().transformToMulti(list ->
      Multi.createFrom().iterable(list))
  .onItem().transform(HttpResponse::body)
  .onItem().transform(String::length);
```

This benchmark interestingly exposes expressiveness differences between the 3 reactive libraries. Reactor looses parametric types in the pipeline chain, forcing the usage of `cast` operators between steps:

```
.flatMapMany(tup -> Flux.fromIterable(tup.toList()))
.cast(HttpResponse.class)
.map(HttpResponse::body)
.cast(String.class)
.map(String::length);
```

RxJava can produce reactive type objects from a Java `CompletionStage`, but unlike Mutiny and Reactor not from a `Supplier<CompletionStage>`. The nuance is subtle as RxJava effectively caches the HTTP request rather than triggering a new one for each new subscription. To reproduce the correct behavior and trigger requests every time a pipeline subscription happens, we need to use the `defer` operator:
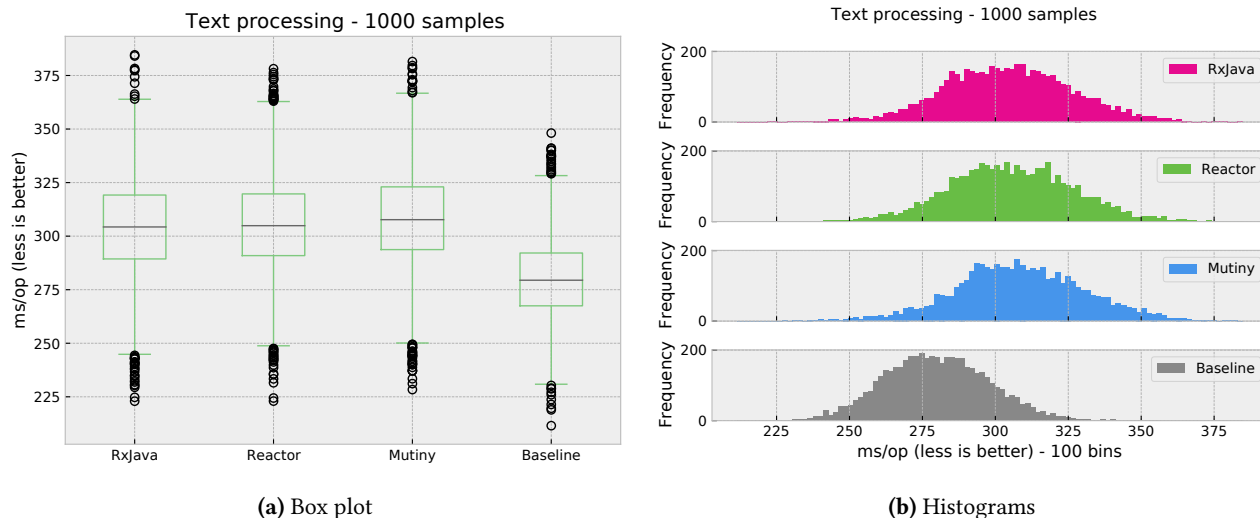
Julien Ponge, Arthur Navarro, Clément Escoffier and Frédéric Le Mouël

(a) Box plot

(b) Histograms

Figure 7. File processing performance.

```
Flowable.zip(
    Flowable.defer(() -> Flowable
        .fromCompletionStage(performHttpRequest())),
    // (...) times 6
)
```

Another interesting issue that was found during the development of this benchmark is the consistency of the HTTP server response times. The initial iteration used the Python embedded HTTP server, but it ended up not responding after serving the file under load, and response times remained fluctuant. The next iteration used a simple Node.js HTTP server, but the latency was too inconsistent due to the adaptive nature of the V8 runtime, both in terms of just-in-time compilation and garbage collection. We instead obtained solid and consistent HTTP response times by running an HTTP server written in Go as it features a non-adaptive runtime.

Figure 8 shows the results. We again used a box plot (Figure 8a) and histogram (Figure 8b) to visualize the data.

Just like in the previous I/O-bound benchmark, there is again no statistical advantage in choosing one reactive library or the other. The performance is very comparable, despite RxJava and Reactor sharing operator fusion and prefetching techniques that Mutiny does not have. Relaxing the reactive streams protocol semantics between RxJava operators does not result in any observable benefit either.

4 Related Work

The reactive streams specification emerged due to the need for coordinating publishers and subscribers of asynchronous, back-pressure enabled data streams[16]. It was heavily influenced by the experience of the Akka actor framework[15] and the RxJava reactive programming library[19] that later

adopted a back-pressured reactive type called `Flowable`. Reactor[20] appeared as a heavily RxJava-inspired reactive programming library, sharing the design of internals, the functional programming operators terminology, yet restricting itself to just 2 reactive types and embracing modern Java constructs at the time (e.g., lambdas and method references). The early designs of Mutiny emerged in 2019, motivated by the need to offer a better developer experience when composing asynchronous operations over reactive types[17], and backed by a fast real-world adoption in projects Quarkus and Vert.x[7, 18]. The reactive streams interfaces have been ported to the Java standard library as part of the `Flow` nested interfaces in Java 9 and beyond, but so far, adoption has remained limited in favor of the original reactive streams library.

All RxJava, Reactor and Mutiny are descendants of the reactive extensions of Erik Meijer[11] that inspired elements of C#, although it should be noted that reactive extensions were never envisioned for back-pressured streams. Reactive extensions build on ideas from *promises and futures* that focused on the composition of asynchronous RPC operations[10]. RxJava is itself part of a family of ports of the reactive extensions idioms to other languages such as RxJs for JavaScript[1].

Since not everything is a stream, RxJava, Reactor and Mutiny all expose a reactive type for single asynchronous operations that are a form of *future*. These 3 reactive programming libraries for Java are thus a mix of *reactive extensions* and *promise / futures* as a programming model, and the *reactive streams protocol* as an interaction model. Moreover, the reactive streams specification does not mandate any specific programming model. For instance Akka actors can cooperate with Mutiny pipelines over reactive streams.
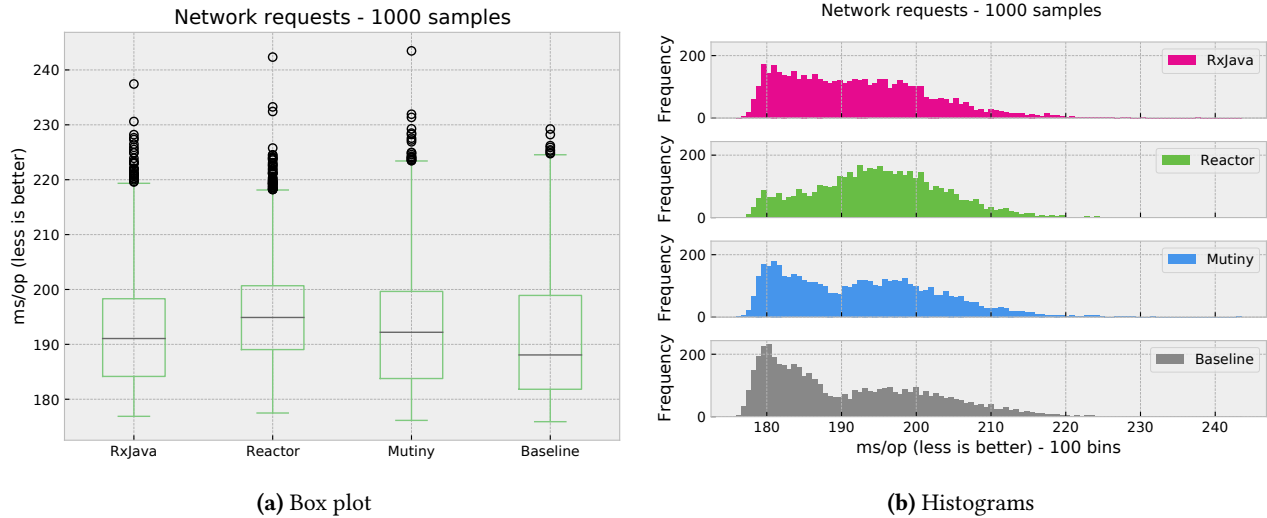
(a) Box plot

(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

operators had any significant effect, and we found a case where memory exhaustion was possible.

Reactive libraries based on reactive streams should not be used for the sole purpose of processing in-memory data such as Java collections. When dealing with collections, the Java streams API offers the ability to build data transformation, selection and aggregation pipelines, including processing parallelization. Reactive libraries inevitably pay the cost of the reactive streams protocol (e.g., demand signalling, multi-threading and serialization requirements) that was primarily designed for asynchronous I/O. Java `CompletionStage` is an efficient framework-neutral type to model asynchronous operations, but it is not subscription-based like with the reactive libraries types so pipeline constructions cannot be cached.

The I/O bound cases depict a more realistic picture of the impact of RxJava, Reactor and Mutiny when used to process what reactive streams were made for. We ran two benchmarks: one that performs operations on the filesystem, and one that performs network requests. In both benchmarks, there is no statistical evidence that either of the libraries performs any better. We could not exhibit any evidence that the operator fusing and pre-fetching techniques employed by RxJava and Reactor could have any favorable impact in realistic workloads where non-trivial I/O operations are involved. Such optimizations seem to have a minor effect in CPU-bound workloads, but, again, there are already better alternatives in the Java standard library.

The engineering costs associated with developing optimization techniques such as operator fusing and pre-fetching is questionable in light of code metrics such as the lines to be maintained and the cyclomatic complexity estimates of Table 1. The simpler code base of Mutiny is within range of the performance of RxJava and Mutiny in CPU-bound cases, and it performs just as well in realistic I/O bound workloads.

There is potential for exploring alternative library implementation and operation techniques, and possibly improve performance. Still, would the development costs be any justified, especially given the results on I/O bound workloads compared to the non-reactive baselines?

Acknowledgments

References

[1] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. ACM, Virtual USA, 15–24. https://doi.org/10.1145/3427763.3428313

[2] Sadek Drobi. 2012. Play2: A New Era of Web Application Development. *IEEE Internet Computing* 16, 4 (July 2012), 89–94. https://doi.org/10.1109/MIC.2012.84

[3] G.K. Gill and C.F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering* 17, 12 (1991), 1284–1288. https://doi.org/10.1109/32.106988

[4] Brian Hayes. 2008. Cloud computing. *Commun. ACM* 51, 7 (July 2008), 9–11. https://doi.org/10.1145/1364782.1364786

[5] Jonas Bonér, Dave Farley, Roland Kuhn, Martin Thompson. 2014. The Reactive Manifesto. https://www.reactivemanifesto.org/

[6] Julien Ponge. 2014. Avoiding Benchmarking Pitfalls on the JVM. *Oracle Java Magazine* (Aug. 2014). https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html

[7] Julien Ponge. 2020. *Vert.x in Action: Asynchronous and Reactive Java.* Manning Publications.

[8] Dan Kegel. 1999. The C10K problem. http://www.kegel.com/c10k.html

[9] Roland Kuhn, Brian Hanafee, and Jamie Allen. 2017. *Reactive design patterns.* Manning Publications.

[10] Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices* (1988), 8.

[11] Erik Meijer. 2012. Your mouse is a database. *Commun. ACM* 55, 5 (May 2012), 66–73. https://doi.org/10.1145/2160718.2160735

[12] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, USA.

[13] Julien Ponge and Mark Little. 2019. Scalability and Resilience in Practice: Current Trends and Opportunities. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, Lyon, France, 267–2670. https://doi.org/10.1109/SRDS47363.2019.00037

[14] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Phoenix AZ USA, 31–47. https://doi.org/10.1145/3314221.3314637

[15] Raymond Roestenburg, Rob Bakker, and Rob Williams. 2016. *Akka in Action.* Manning Publications.

[16] Reactive Streams Special Interest Group. 2014. Reactive streams specification. https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.3/README.md#specification

[17] Red Hat and contributors. 2021. Mutiny. https://smallrye.io/smallrye-mutiny/

[18] Red Hat and contributors. 2021. Quarkus. https://quarkus.io/

[19] RxJava contributors. 2021. RxJava. https://github.com/ReactiveX/RxJava

[20] VMWare and contributors. 2021. Project Reactor. https://projectreactor.io/

[21] Bill Williams. 2012. *The Economics of Cloud Computing: An Overview For Decision Makers [Book]*. Cisco Press. https://www.oreilly.com/library/view/the-economics-of/9780132904186/

[22] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–29. https://doi.org/10.1145/3360610