POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Performance Evaluation of Kafka Clients Using a Reactive API

Supervisor
prof. Marco Torchiano

Candidate
Andrea Amato

Internship Tutors
dott. Alessandro Zanon
dott. Andrea Malan

Academic year 2022-2023

Summary

This thesis evaluates the performance of a core-banking application that consumes messages from a Kafka topic, and after performing validation and a transformation stores the results in another Kafka topic. Performances are evaluated on two different versions of the application: traditional and reactive. The focus is on the way the two applications interact with Kafka.

Both versions are developed using Spring: the traditional version uses Spring MVC while the reactive one uses Spring WebFlux. Spring WebFlux internally uses Project Reactor. Both versions consume and produce messages from/to Apache Kafka, however in the reactive version messages are consumed and produced using functional APIs provided by Reactor Kafka. Internal metrics of Kafka clients are collected through the JMX reporter.

The obtained results show that the reactive implementation is able to achieve higher throughput and use fewer resources with respect to the traditional implementation.

The implication of this result is that a company that develops software using a reactive approach is able to accommodate a higher number of requests with equal resources poured into the system. Alternatively, an organization could achieve the same throughput but use fewer resources.

Acknowledgements

I would like to express my deepest gratitude to my parents and sister for their immense support during my academic journey.
This endeavor would not have been possible without the guidance and inspiration of my internship tutors, Alessandro Zanon, and Andrea Malan.
Thanks should also go to Iriscube, the company in which this thesis project has been carried out.
In Iriscube, I had the pleasure of working with exceptional people who made me feel welcome and part of a family.

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface

J2SE Java 2 Platform, Standard Edition

JMX Java Management Extensions

JSON JavaScript Object Notation

JVM Java Virtual Machine

MOM Message-Oriented Middleware

MVC Model-view-controller

POJO Plain Old Java Object

RMI Remote Method Invocation

RPC Remote Procedure Call

Chapter 1

Introduction

1.1 Context

The context of this thesis is the development of event-driven architectures for core-banking applications. Core-banking applications process daily transactions and post updates to accounts as well as other financial records. These applications are the core of banking services the banks offer to mobile and desktop users or headquarters/branch customers. Practical examples of these services are bank transfers, accounting, and handling money via credit cards.

1.1.1 Company Overview

This thesis project has been carried out with Iriscube Reply. Iriscube Reply is one of the companies belonging to the Reply S.p.A. group. Reply is a company specializing in Consulting, System Integration, and Digital Services, dedicated to the conception, design, and implementation of solutions based on new communication channels and digital media. Reply supports the leading industrial groups in defining and developing business models enabled by new technological and communication paradigms, such as Artificial Intelligence, Big Data, Cloud Computing, Digital Communication, and the Internet of Things, to optimize and integrate processes, applications, and devices. Reply's value proposition aims to promote customer success by introducing innovation along the entire value chain, thanks to the knowledge of specific solutions and the consolidated experience on the main core issues of the various industrial sectors. Reply provides the following services:

- Strategic, communication, process, and technological consultancy.

- System integration, making the most of the potential of technology by combining business consultancy with innovative technological solutions with high-added value
- Application Management consisting of management, monitoring, and continuous evolution of application assets.

Iriscube Reply is focused on IT consultancy and the development of innovative web and mobile solutions in the world of the Finance Sector with particular reference to multi/omnichannel. In the context of banking and insurance, Iriscube Reply develops information portals, internet and mobile banking, native and hybrid applications to support financial advisors, social digital wallets, intelligent systems for contact centers, wealth management platforms, digital bio-metric systems, machine learning, and more. Iriscube Reply provides services to some of the leading and major Italian and European banks [1].

1.1.2 A Data-Ingestion Application

One of the services provided by Iriscube Reply is a data-ingestion application for one most important banks in Italy. This application consumes a high amount of requests coming from an Apache Kafka message broker. The incoming messages represent inbound or outbound operations performed on a bank account. For each received request, the data-ingestion application performs validation and maps the incoming event to an output event in a format compatible with the destination application i.e. the core-banking platform.

1.1.3 Problem Definition

During my experience with Iriscube Reply, I was assigned the task of setting up a load scenario to measure the performance of the data-ingestion application. One of the most important performance metrics is throughput. However, the data ingestion application is basically a Kafka client. Kafka clients can be consumers or producers. The data ingestion application acts as both consuming events from a Kafka topic and producing the corresponding output event to another topic. In this event-based asynchronous interaction, we should separate the throughput in terms of messages consumed per second and messages produced per second. Since production depends on consumption, the ingestion phase can become a bottleneck for the production phase.

1.2 Purpose

This thesis work aims to evaluate and compare the performance of different implementations of the same data-ingestion application. We will define two base implementations: traditional and reactive. In turn, the reactive implementation is declined into other implementations which try to exploit the thread scheduling model provided by Project Reactor. Therefore, another purpose of this thesis project is to optimize the reactive implementation to achieve higher throughput. Moreover, we will try to understand the trade-offs and overhead introduced by each solution by analyzing the resource consumption of the process on which the application is run.

1.3 State of the Art

In the literature, there is a lack of research on the reactive implementation of Kafka clients.

Le Noac'h *et al.* studied how the ingestion phase can become a bottleneck of the processing pipeline. The ingestion phase in Kafka relies on producers and consumers. They highlighted how a variation of *batch-size* can bring better performance while increasing the number of nodes in the Kafka cluster causes a drop in the throughput due to Kafka internal synchronization [2].

Hesse *et al.* analyzed how the configuration of a data sender influences the ingestion rate of Apache Kafka. They mainly tuned two parameters: *batch-size* and *acks*. The *batch-size* is used to lower the number of requests and increase the throughput at the expense of latency. The *acks* producer property is used to select the level of acknowledgments of sent messages. Surprisingly they discovered that when the producer does not wait for acknowledgments of messages, the Kafka Broker ingests fewer messages [3].

Van Dongen and Van den Poel benchmarked the scalability of four popular stream processing frameworks under different types of workloads. The scalability metrics taken under consideration were latency, throughput, and CPU and memory utilization. The considered workloads scenarios were constant rate, periodic burst, and burst at startup [4]. In a later work, they extended the benchmark of scalability of the stream processing frameworks by analyzing the influencing factors in the scalability. This work analyzed how the latency, throughput, and resource bottlenecks are influenced by the framework design, pipeline design, Kafka cluster layout, scaling direction, resource allocation, and the characteristics of data [5].

Vyas *et al.* evaluated the impact of polling interval and the number of partitions on the performance of Kafka producer and consumer API. They concluded that increasing the number of partitions improved the throughput. In addition, reducing the poll interval can improve the performance of the consumer. However, an overly small polling interval value can cause data loss during ingestion [6].

1.4 Main Contributions

The first contribution of this thesis is the empirical observation of higher performance attained by implementing a Kafka client using a reactive API. This is achieved by defining a clear bench-marking plan and selecting meaningful metrics to monitor performance. Another contribution is the exploration of possible optimizations and trade-offs introduced by exploiting the thread scheduling model provided by Project Reactor.

1.5 Outline

Chapter 2 aims to present the background upon which this thesis work is based. We give an essential picture of distributed systems. Then we introduce the concepts behind reactive systems. We proceed to present Apache Kafka, which is the message broker used in the event-driven architecture under study. Finally, since the application developed is written in Java, we briefly describe the technology used to monitor and control Java applications.

Chapter 3 dives into the design adopted to benchmark the different flavors of the data-ingestion application. We define the metrics, the different versions of the pipeline, and the workloads used to evaluate performances.

Chapter 4 presents the results obtained during benchmarking. By analyzing the resulting graphs, we try to evaluate how the different versions of the pipeline scale under different workloads and what are the trade-offs of the solutions.

Chapter 5 draws conclusions about the thesis work, highlighting limitations and future research directions.

Chapter 2

Background

2.1 Distributed Systems

We can define a distributed system as "a collection of autonomous computing elements that appears to its users as a single coherent system" [7]. From this definition, we can extract two key features of a distributed system. The first one is that computing elements are autonomous i.e. they can act independently from each other. The second one is that users interact with a distributed system as if it were a single system i.e. the distribution is transparent to its users (people or processes). Being autonomous, the elements need to collaborate to achieve a common goal. How this collaboration is established is the core of distributed systems development. The collection of nodes is typically organized as an overlay network in which nodes communicate by exchanging messages. Often there is no global clock, and this implies the need for synchronization between nodes.

2.1.1 Middleware

A middleware is a software layer placed on top of the operating system that provides facilities to the distributed system. The services provided by a middleware include communication between applications, security, accounting, and recovery from failures. These services are offered in a networked environment.

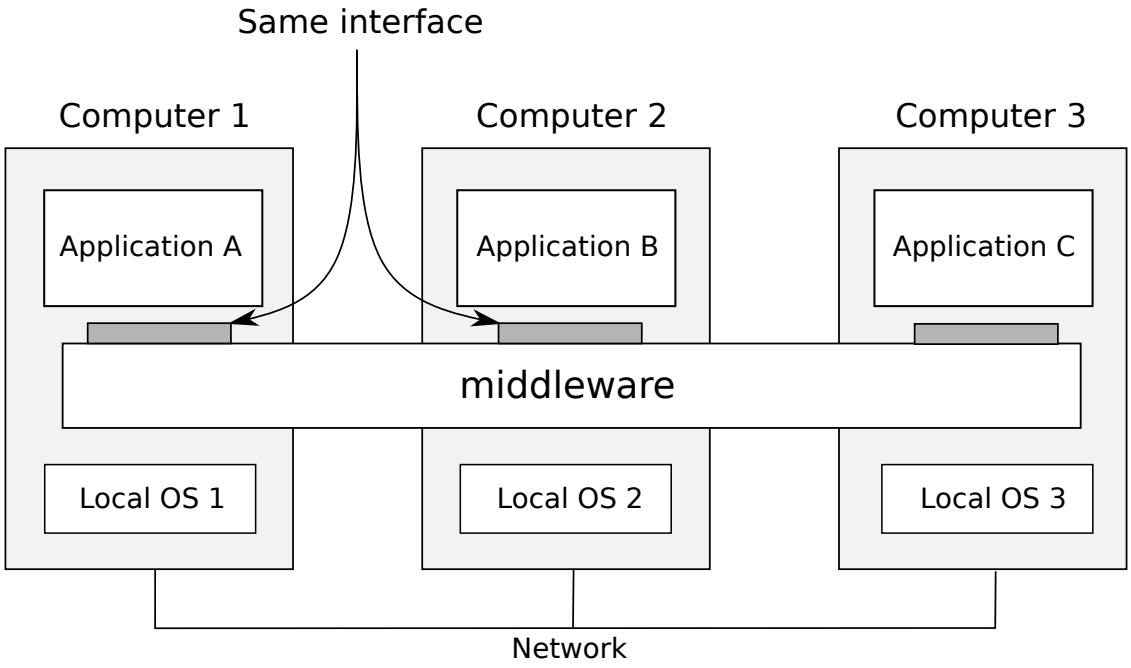

Figure 2.1. Middleware layer

2.1.2 Scalability

One of the design goals that should be met when developing a distributed system is scalability. Scalability has different dimensions, but the one we will focus on is size scalability. Size scalability is defined as the ability of the distributed system to accommodate an increasing number of requests without a significant loss of performance. The major causes of bottlenecks in size scalability are the computational capacity, the storage capacity, and the network between the user and the distributed system. Two possible approaches can be followed to improve scalability. The first one is known as scaling up or vertical scaling and consists of improving the capacity of the server or network. The second one is known as scaling out or horizontal scalability and is achieved by deploying more machines.

Hiding communication latencies

When a node of a distributed system receives a request, this typically triggers one or more sub-requests to other nodes. In order to produce a response, the node must wait for the results of the sub-requests. This implies a waste of resources since the element in charge of the response is idle as long as

the sub-requests are completed and cannot accommodate other incoming requests. A possible solution consists in creating a new thread that will handle the request. This thread may block receiving the responses of sub-requests, but as a new request arrive, the server can create another thread to handle the new request. Alternatively, we can hide these communication latencies by avoiding waiting for the response from the sub-requests. This, in turn, implies an asynchronous approach. When receiving a request, we do not wait for the response, but we register a handler to be invoked once the response is received.

2.1.3 Communication

When studying a distributed system, one of the most important aspects is how processes on different machines communicate with each other. We have two main communication middleware: Remote Procedure Call (RPC) and Message-Oriented Middleware (MOM).

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is ideal for client-server interaction and is used by an application to send a request to another application as if it were a local procedure call. RPC requires that the caller and callee are both up and running during the communication and implies also a referential coupling.

Remote Method Invocation (RMI)

Java provides an RPC mechanism through Remote Method Invocation. RMI operates on objects instead of functions. A Java client can invoke a method of an object owned by another Java Virtual Machine.

2.1.4 Message-Oriented Middleware (MOM)

There are cases in which we cannot make any assumption about the execution state of the receiving application during communication. In addition, the synchronous nature of RPC implies that the client remains blocked until a response is received. To overcome these limitations, a different approach is needed: Message-Oriented Middleware (MOM). Communication happens by sending a message to a logical point. Messages are appended in queues: this enables communication to be asynchronous. An application can manifest its interest in a specific queue. The middleware takes care of delivering the

messages to the interested subscribers. A MOM can be used to overcome the drawbacks of the RPC mechanism since it does not require the communicating parties to be simultaneously active, and they do not need to know each other.

2.1.5 Message Brokers

In MOM systems, the conversion of messages is handled by special network nodes called message brokers. A message broker is basically another application that acts as an application-level gateway converting incoming messages so that they can be understood by the destination applications. An application can publish a message on a certain topic. The message is then sent to the message broker. Applications can manifest their interest in messages published on a specific topic by subscribing to that topic. The message broker will deliver the messages published on a topic to the applications that have subscribed to that topic.

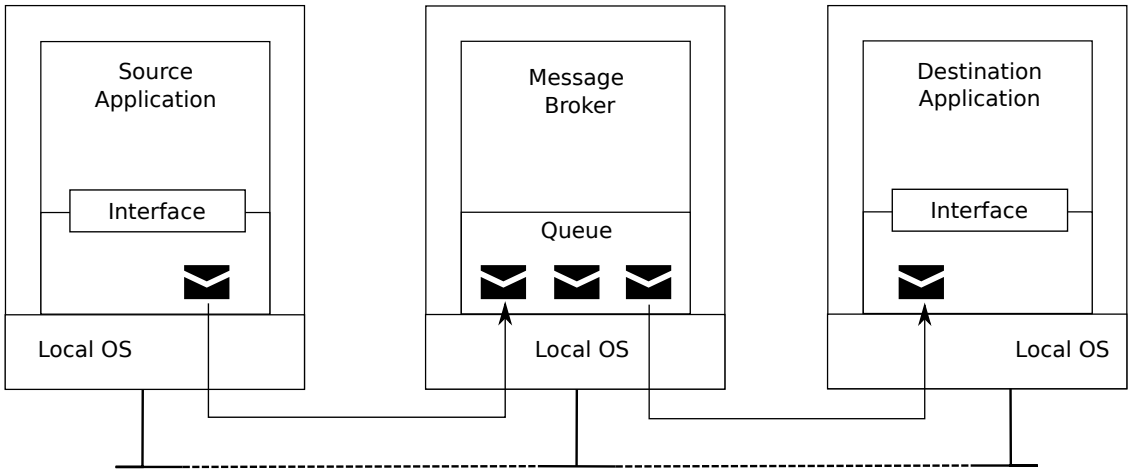

Figure 2.2. Message broker in a MOM system

2.1.6 Architectural Styles

To manage the complexity of a distributed system is necessary to organize the system according to an architectural style. An architectural style is defined based on how components interact with each other. A component is a module that exposes an interface. The component can be replaced while the system is operating as long as its interface remains stable. In a distributed system,

some components may be temporarily down. Another important concept in an architectural style is that of the connector. A connect provides services to allow communication between components through procedure calls or message passing. By combining components and connectors according to different configurations, we can get different architectural styles: layered architectures, object-based architectures, resource-centered architectures, and event-based architectures. In real-world scenarios, these architectures are often combined. In this thesis, we focus on event-based architectures — also known as publish-subscribe architectures — which use the MOM for communication.

2.1.7 Publish-subscribe Architectures

A publish-subscribe architecture is characterized by loose dependencies between processes. In such an architecture, processing and coordination are strongly separated. Different types of coordination models are possible depending on the referential and temporal coupling of processes. Referential coupling happens when a process has to explicitly reference the other process in order to communicate. Temporal coupling means that processes have to be up and running during communication. Using Apache Kafka as a message broker, a publish-subscribe architecture can achieve referentially decoupled and temporally decoupled coordination. In this way, processes do not need to know each other, and communication can take place even if processes are not executing at the same time. In order to achieve temporal decoupling, Apache Kafka provides a retention mechanism for published messages. The interface of components in an event-based architecture is message-centric. A message-centric interface is fixed and has a single-operation e.g. `send(message, topic)`. Its design is about messages and topics.

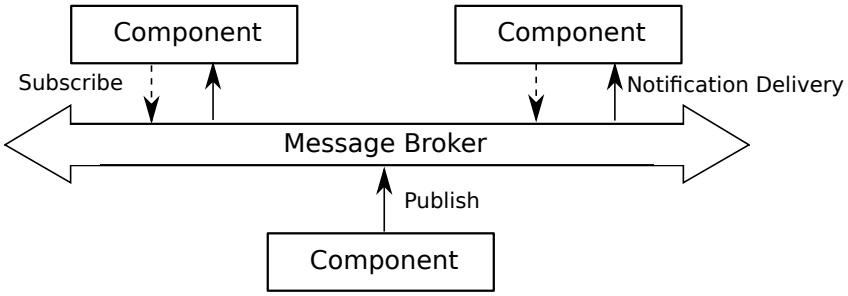

Figure 2.3. Publish-subscribe architecture

2.2 Reactive programming

Today's distributed system should be able to react to changes in demand and in the availability of both internal and external components of the system. In order to achieve this goal, the system should evolve to be reactive. According to the Reactive Manifesto, reactive systems are: responsive, elastic, resilient, and message-driven [8].

Elastic

Elasticity is defined as the ability to stay responsive in spite of a varying workload. As more users use the system, its throughput should increase to keep up with the demand, while as the need goes down, the throughput should automatically shrink. One way of achieving elasticity is by employing horizontal or vertical scalability. However, scalability is limited by synchronization and resource bottlenecks. Moreover, the additional resources or instances poured into the system should be able to adjust to the demand: if the demand is low, the consumption of the resource should decrease in order to reduce business expenses. In the Java world, thread pools are used to increase parallel processing. Under high loads, this technique is inefficient since when using a request-response model, threads spend most of the time blocked, waiting, and wasting CPU resources.

Responsive

Elasticity is fundamental for the system to be responsive. Responsiveness is defined as the ability of the system to respond timely (i.e. with an acceptable latency) to user requests.

Message Driven

To achieve better resource utilization, message-driven communication should be adopted. The message-driven communication model enables communication to be asynchronous and non-blocking. Non-blocking means that a thread will get access to a resource only if it is immediately available. Otherwise, the thread, instead of waiting, will do other useful work. When the resource returns to be available, the thread will be notified.

Resilient

Employing message-driven communication will also enable the system to be resilient. A resilient system stays responsive even in case of failure.

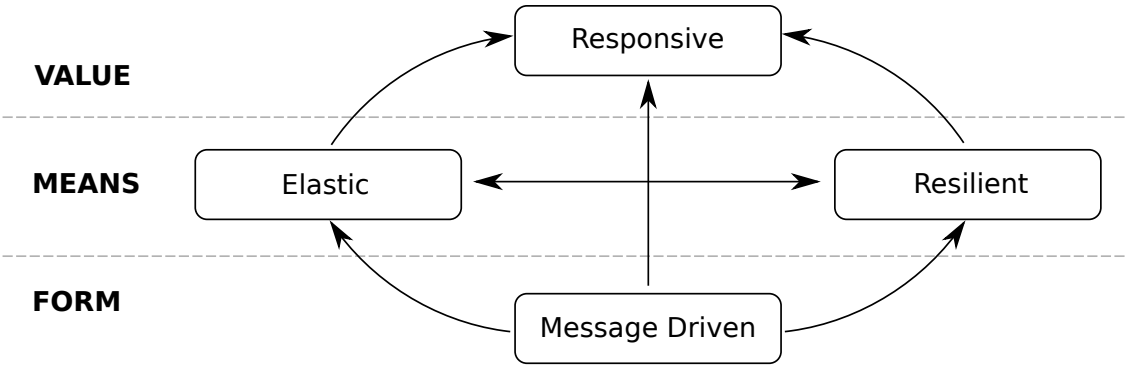

Figure 2.4. Reactive Manifesto

2.2.1 Reactive Streams

Reactive Streams represent the specification for reactive programming patterns. Reactive Streams were introduced to solve the problems of inconsistencies between APIs and flow control [9].

Regarding the API's inconsistency problem, several choices are available when it comes to reactive libraries. These libraries, such as RxJava, can provide asynchronous, non-blocking communication but have different APIs. In order to make these independent libraries compatible, we need to provide a custom adaptation that may contain defects and require extra maintenance.

To understand the flow control problem, we need to distinguish between pull and push models. In the pull model, the consumer requests data from the publisher one at a time. The pull model is inefficient when working with the network boundaries since large communication latencies sum up, and the consumer remains idle until the response is received. The batching of the requests can improve performance in a limited way. In the push model, we improve performance by allowing the publisher to push data as they are available. In this way, the idle time due to the communication latency is limited to the first request. However, the push model introduces the problem of flow control since we should avoid the publisher from overwhelming the subscriber. According to the Reactive Manifesto [8], a reactive system

should be elastic; this means that the consumer needs a mechanism to properly respond to the load. The needed mechanism is known as backpressure control. The flow control problem is solved with the introduction of the four interfaces defined by the Reactive Streams specification [10]. These interfaces allow backpressure control by adopting a hybrid push-pull model. In the following, we give a description of the Reactive Stream interfaces.

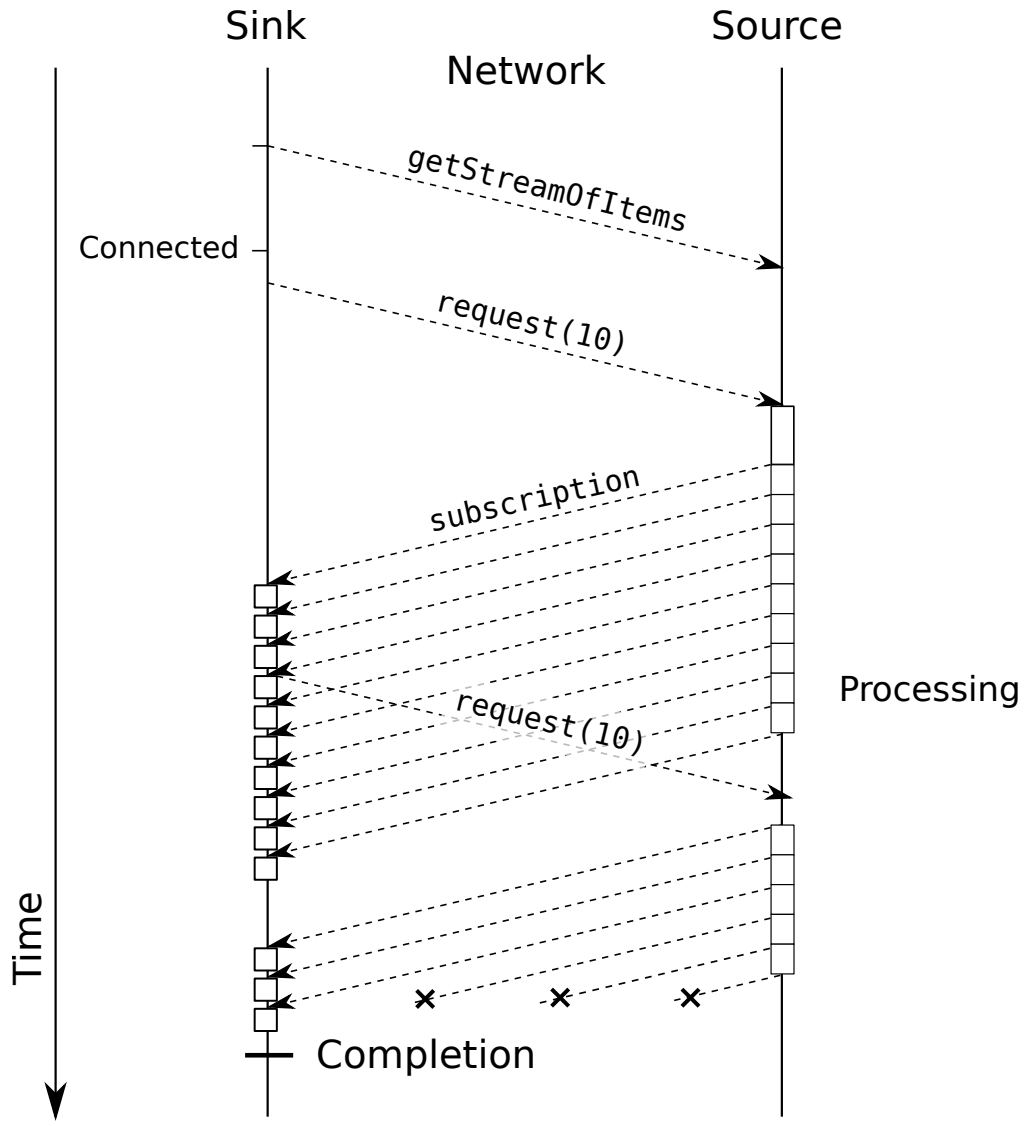

Figure 2.5. Hybrid push-pull model

Publisher

The `Publisher` interface represents the entry point for a connection between the publisher and the subscriber. The publisher interface provides one single method used to register a `Subscriber`.

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

Subscriber

The `Subscriber` interface introduces the `onSubscribe` method to notify the `Subscriber` about a successful subscription. The `onSubscribe` method receives a parameter of type `Subscription` that represents a contract between publisher and subscriber.

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

Subscription

The `Subscription` interface provides methods to control the production of elements. The `cancel()` method is used to request the publisher to stop sending data. The `request` method is used by the `Subscriber` to signal the amount of data that should be pushed by the `Publisher`. In this way, the `Subscriber` has control over the number of elements it will receive. Unlike the pure push model, the request method allows a hybrid push-pull mechanism that, in turn, enables control over the backpressure.

```
public interface Subscription {
  public void request(long n);
  public void cancel();
}
```

Processor

The `Processor` is a combination of `Publisher` and `Subscriber`. The `Processor` allows the addition of intermediate processing stages between the `Publisher`

and the `Subscriber`, which respectively represent the start and end points of the pipeline.

```
public interface Processor<T, R> extends Subscriber<T>,
 Publisher<R> {
}
```

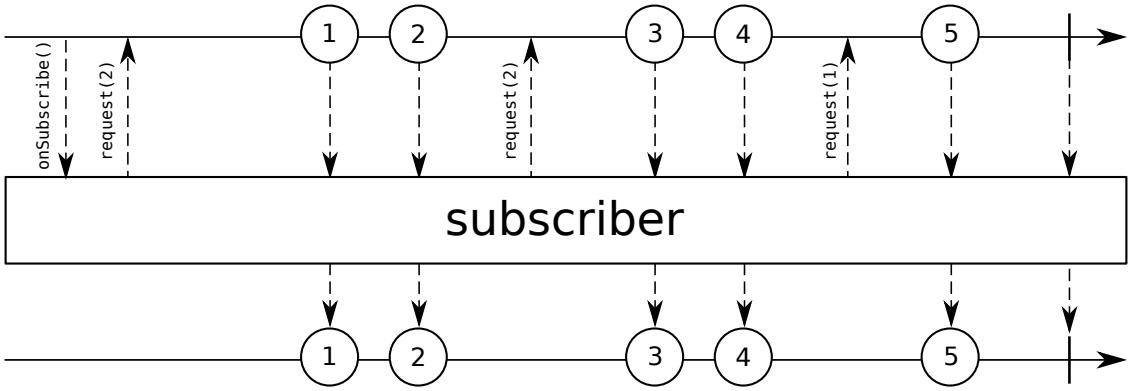

Figure 2.6. Backpressure control

2.2.2 Project Reactor

The Reactive Streams specification only defines APIs and rules for asynchronous stream processing with non-blocking back pressure. Project Reactor has become the most state-of-the-art implementation of the Reactive Streams specification. The Reactor library allows building asynchronous pipelines devoid of callback hell and deeply nested code. In fact, the Reactor library improves code readability and enables the composability of the workflows. Complex workflows can be built by chaining the operators provided by the Reactor API. These operators are the biggest added value over the basic Reactive Streams specification. Since Reactor implements the Reactive Streams specification, it also provides all common mechanisms for backpressure propagation: push only, pull only, and push-pull [9].

Project Reactor provides two implementations of the `Publisher<T>` interface: `Flux<T>` and `Mono<T>`. `Flux` is a reactive stream that can produce zero, one, or many elements. Potentially an infinite amount of them. While `Mono` represents a reactive stream that can produce at most one element i.e. zero or one. The purpose of `Mono` is to enable more efficient use of resources in the case of an application API that returns at most one element.

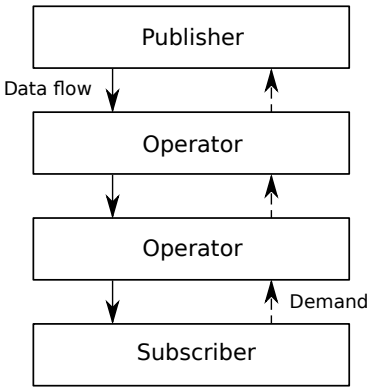

Figure 2.7. Data flow and demand signal propagation

The actual data flow from the publisher to the subscriber is triggered only when the subscriber creates a subscription. `Flux` and `Mono` provide overloads of the `Publisher<T>` subscribe method. Such overrides return an instance of the `Disposable` interface. A reactive stream may be finished by the producer emitting `onError` or `onComplete` signals, by the subscriber through the `cancel()` method of the `Subscription` instance, or by using the `dispose()` method of the `Disposable` instance.

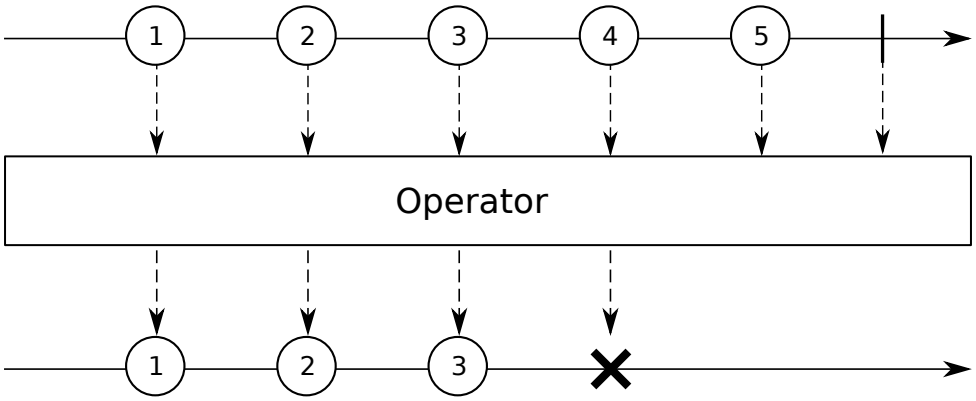

Figure 2.8. Transformation of a Flux stream

Operators

Project Reactor provides operators to manipulate and transform reactive streams [11]. For example, if we want to transform an existing reactive sequence, we may use the `map` operator. The `map(Function<? super T,?`

`extends V> mapper)` method processes elements one by one. After the transformation, the whole sequence changes its type from `Flux<T>` to `Flux<V>`.

If we need to filter a reactive sequence, one of the most useful operators is the `filter` operator, which returns a `Flux` whose elements satisfy the given predicate.

Other operators provide a way of peeking elements into a sequence without modifying the final sequence. In order to add additional behavior (side-effect) when the `Flux` emits an element, we use the `doOnNext(Consumer<? super T> onNext)` method.

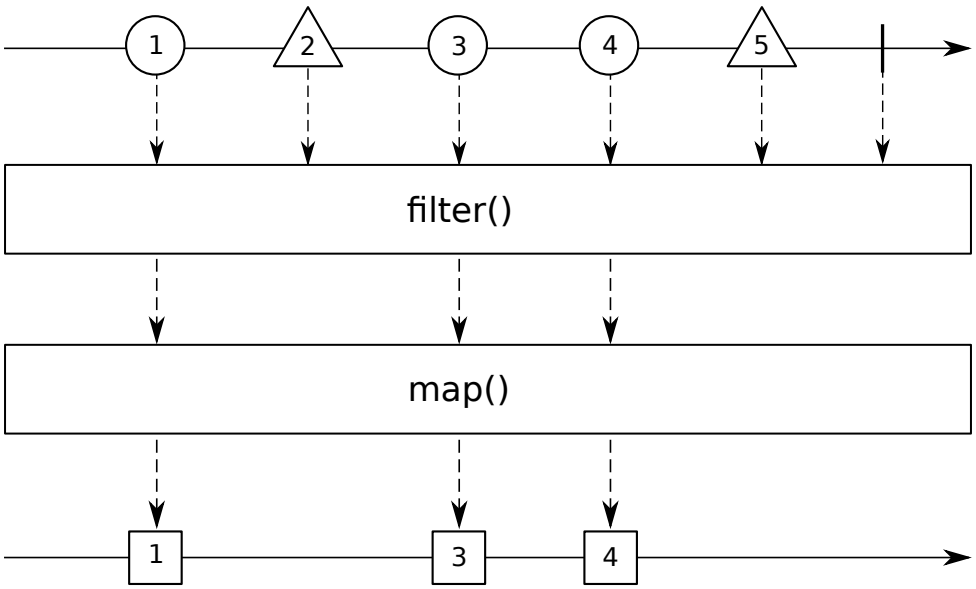

Figure 2.9. Example of operators applied to a Flux

2.2.3 Reactive Streams Life Cycle

In Project Reactor, the life cycle of reactive types is composed of three steps: assembly-time, subscription-time, and run-time.

Assembly-time

Reactor provides a fluent API to process elements and build complex flow. The process of building complex flow is named assembling. The assembling is characterized by immutability i.e. each operator of the flow produces a new object. Assembly-time composes Fluxes into each other, chaining

the Publishers; each Publisher wraps the previous one. Under the hood, Reactor tries to improve the overall performance of the stream by possibly replacing one operator with another according to the stream type.

Subscription-time

The subscription-time phase is triggered once we subscribe to a Publisher. The subscription process starts by subscribing to the top wrapper of the chain of Publishers. This subscription triggers the execution of the subscribe method for each inner Publisher. After subscription-time, we will get a sequence of Subscribers wrapped inside each other. Optimizations are also possible, as in the case of assembly-time.

Run-time

During run-time Publisher and Subscriber exchange signals. The first two are the onSubscribe signal and the request signal. When the onSubscribe method is called by the top Publisher wrapper, a chain of Subscriptions is passed to the Subscriber. In order to receive elements, the Subscriber calls the request method of the received Subscription. One of the optimizations that are applied during run-time, is the reduction of the number of signals exchanged.

2.2.4 Thread Scheduling in Project Reactor

In this section, we describe the main abstractions for multi-threading execution provided by Project Reactor.

The publishOn operator

The Flux<T> publishOn(Scheduler scheduler) operator moves part of the run-time phase to a specified worker from the provided Scheduler instance. The publishOn operator introduces an asynchronous boundary within the flow of operators. This means that the flow can be split into parts that will be processed independently from each other. The split parts are linked by a queue from which the dedicated worker can consume and process messages one by one.

Note that the elements of a Reactive Stream are not processed concurrently i.e. events in the queue are processed in strict order by only one worker. Nonetheless, with the publishOn operator, we can potentially speed

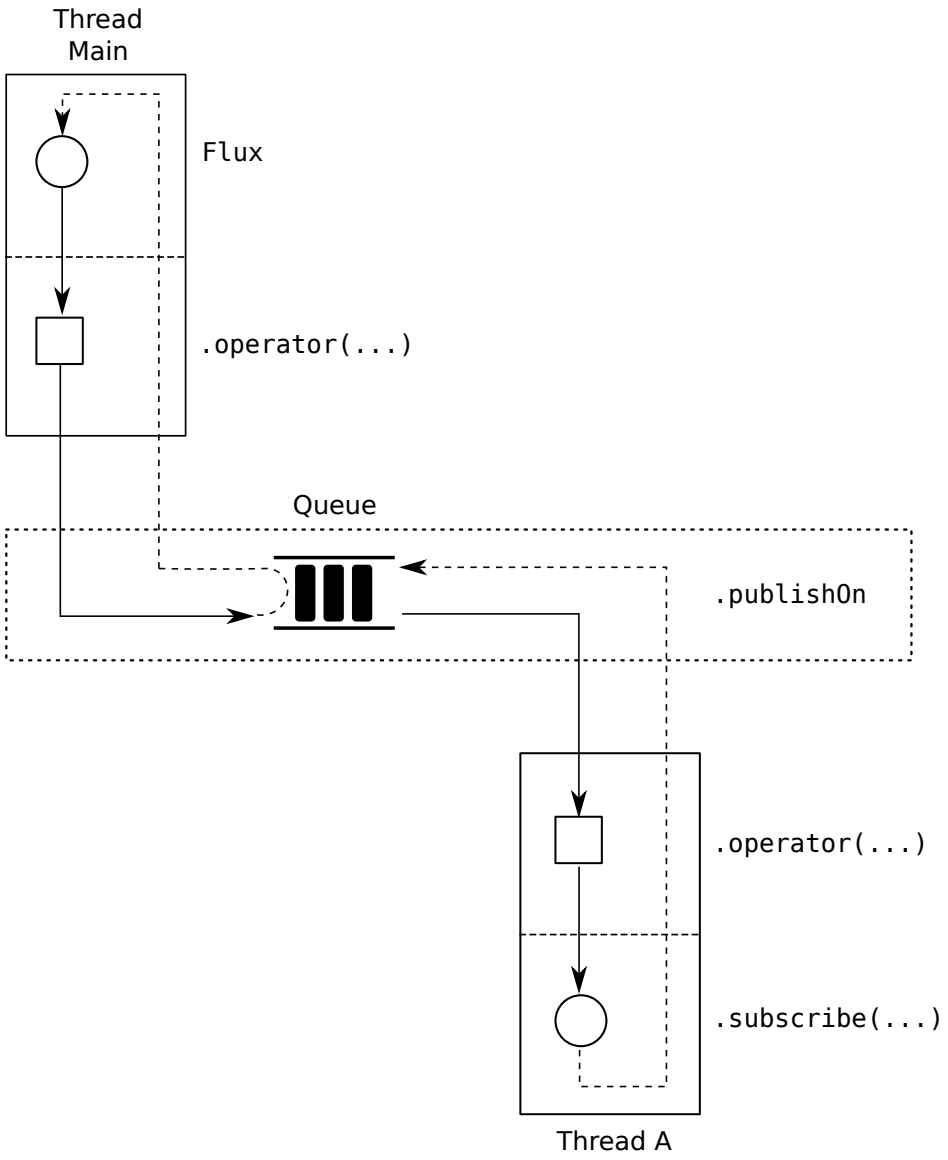

Figure 2.10. Internals of the publishOn() operator

up the processing by using parallel processing. Parallel processing is achieved by allowing the left-hand side of the asynchronous boundary to work independently from the right-hand side. So the left-hand side does not have to wait for the completion of the right-hand side.

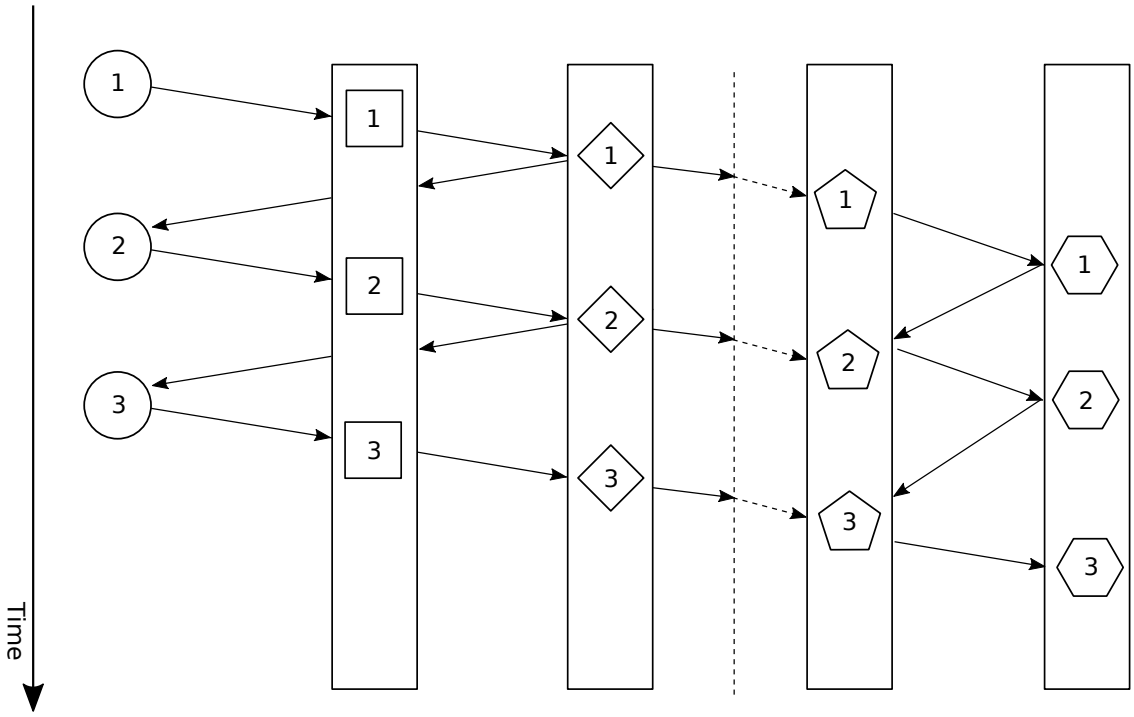

Figure 2.11. Parallelization with the publishOn() operator

The subscribeOn operator

The `Flux<T> subscribeOn(Scheduler scheduler)` provides us a way to specify the worker on which the subscription chain will take place. In other words, it changes the worker on which the `subscribe` method is executed and starts generating data. While the `publishOn` operator affects only the downstream execution, the `subscribeOn` operator specifies the behavior of the upstream execution.

The parallel operator

The `parallel()` operator is part of the `Flux` API. The `Flux` on which the `parallel` operator is invoked is transformed into a `ParallelFlux`. `ParallelFlux` represents a group of Fluxes over which the elements of the source `Flux` are partitioned. The `ParallelFlux<T> runOn(Scheduler scheduler)` operator allows applying `publishOn` to the internal Fluxes in order to distribute the processing of elements between different workers. Each element of the `Flux` is processed one by one, while Fluxes belonging to the `ParallelFlux` are processed concurrently.

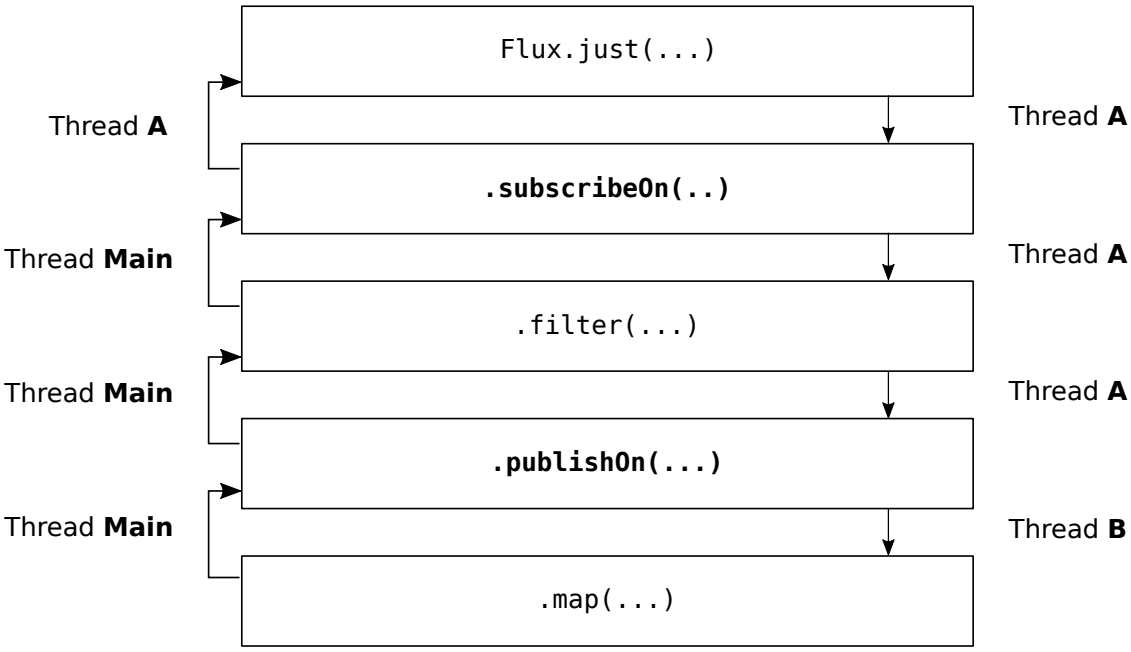

Figure 2.12. The subscribeOn() operator

Scheduler

In Reactor, the `Scheduler` interface represents a worker or a poll of workers. Conceptually, a `Worker` is a thread, but it is not necessarily backed by a `Thread`. The different flavors of `Scheduler` are provided by the `Schedulers` factory methods [12]. For example, `Schedulers.parallel()` can be used for short-lived CPU-intensive tasks. `Schedulers.boundedElastic()` is more suitable for long-lived tasks i.e. blocking tasks.

2.3 Apache Kafka

Every enterprise works with data. Data is constantly received, analyzed, and manipulated in order to create other data. Data need to be moved from where it is created to destinations where it can be analyzed. This pipeline is a crucial process in a data-driven enterprise, and its efficiency has become important as the data itself since it enables an organization to be responsive to users.

2.3.1 Publish-Subscribe Pattern

Apache Kafka is based on a publish-subscribe pattern. In the publish-subscribe pattern, the sender publishes a message not directed to any specific receiver. However, the publisher categorizes the message according to a certain class. The receiver can subscribe to a specific class of messages, and a message broker will take care of delivering the messages to all interested subscribers.

2.3.2 What is Apache Kafka?

Apache Kafka was originally developed by people at LinkedIn to address data pipeline issues. Kafka has been defined as an "event streaming platform" [13]. Messages sent to Kafka are retained in a database commit log so past messages can be replayed by subscribers. Data can be also distributed throughout the system in order to support scalability and protection against failures.

2.3.3 Messages and Batches

A message is a unit of data in Kafka. A message has not any specific format or meaning to Kafka and consists of an array of bytes. A message can have a key, represented also as a byte array. The key is used to write messages to partitions in a controlled manner: messages with the same key will end up in the same partition.

To achieve higher throughput, messages are collected into batches. The messages composing a batch are published into the same topic and partition. When it comes to batching messages, we have a trade-off between latency and throughput: by increasing the batch size, we can process more messages per unit of time, but it takes longer for a single message to be transmitted.

2.3.4 Topics and Partitions

Kafka messages are organized into topics. A topic can comprise several logs called partitions where messages are appended. The message ordering is guaranteed only within a single partition and not across partitions. Partitions can be hosted on different servers so that the topic can scale horizontally. Moreover, partitions can be replicated to provide redundancy.

2.3.5 Kafka Clients: Producers and Consumers

There are two types of Kafka clients: producers and consumers. Producers publish messages addressing a specific topic. If a message has no key, the producer will evenly distribute messages into the partitions of a topic.

Consumers subscribe to one or more topics and consume messages in the same order they were produced in each partition. The consumer keeps track of the read messages by checking the offset of each message. In fact, Kafka adds to each produced message an offset represented by an increasing integer value. The offset is unique within a partition, and later messages will have a greater offset. By saving the next possible offset for each partition, a consumer can stop and restart without losing its position in the log.

Consumers are organized in a consumer group. A consumer group is composed of one or more consumers that received data from the same topic. The group guarantees that each partition is consumed only by one member. The association between a consumer with a partition is known as the ownership of the partition. In case of a consumer failure, the corresponding partition(s) will be reassigned to the remaining members.

2.3.6 Brokers and Clusters

A broker is a single Kafka server. After receiving messages from producers, the broker assigns an offset to them and stores them on disk. The broker is responsible for delivering the messages to consumers that fetch messages from partitions.

Kafka brokers can be organized into a cluster. One of the brokers will be elected controller of the cluster. The controller serves administrative functions like monitoring brokers' failure and assigning partitions to brokers. A partition has a single owner in the cluster. The owner is called the leader of the partition. To provide redundancy, a partition can be replicated; in this case, the replicated partitions are assigned to other brokers which become followers of the partition. In order to publish a message, a publisher

must connect to the leader of the partition, while a consumer can fetch even from the followers. One important aspect of Apache Kafka is that messages are retained for some time before being deleted. This enables a temporal decoupling between producers and consumers. The retention policy can be configured with custom rules according to the application's needs. If a consumer fails or must be stopped for maintenance, no data are lost [14].

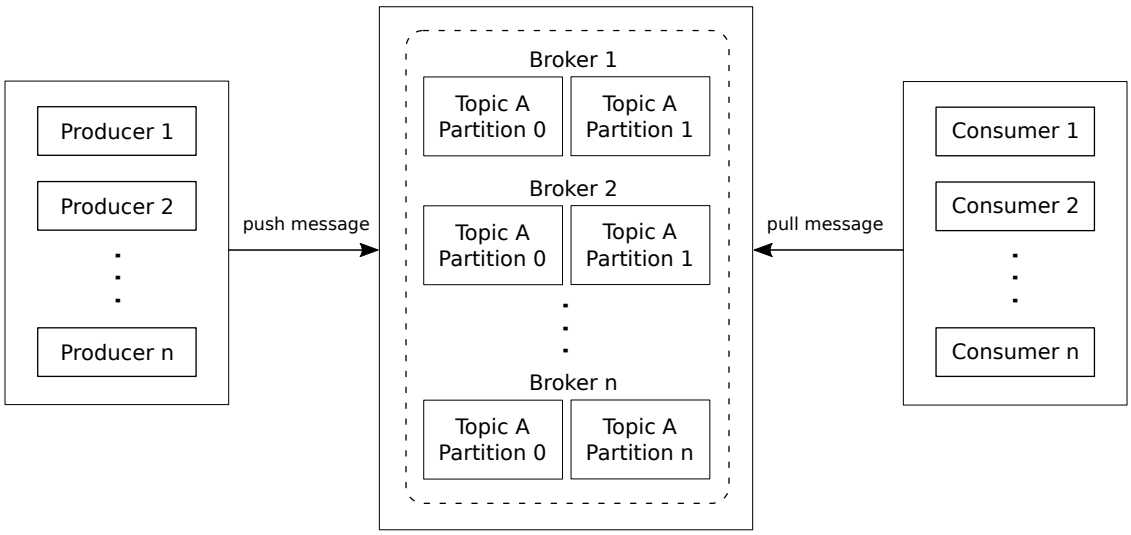

Figure 2.13. Apache Kafka Architecture

2.3.7 Spring for Apache Kafka

Spring for Apache Kafka [15] is a project that utilizes Spring core concepts for the development of Kafka-based applications. It is possible to create and configure a topic using the `TopicBuilder` class. We can utilize such a class to create a `NewTopic` Bean:

```
@Bean
public NewTopic topic() {
  return TopicBuilder.name("my-topic-name")
  .partitions(8)
  .replicas(2)
  .build();
}
```

`KafkaTemplate` class wraps a producer and provides several methods to send messages to topics. Example of a method to send messages:

```
public ListenableFuture<SendResult<K, V>> send(String topic,
 @Nullable V data)
```

A producer factory should be configured and provided to the template's constructor:

```
@Bean
public ProducerFactory<String, outDTO> producerFactory() {
  return new DefaultKafkaProducerFactory<>(producerConfigs());
}

@Bean
public Map<String, Object> producerConfigs() {
  Map<String, Object> props = new HashMap<>();
  props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "
 localhost:9092");
  props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
 StringSerializer.class);
  props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
 JsonSerializer.class);

  return props;
}

@Bean
public KafkaTemplate<String, outDTO> kafkaTemplate() {
  return new KafkaTemplate<>(producerFactory());
}
```

The send method of `KafkaTemplate` returns a `ListenableFuture`. This means that the sending operation is asynchronous (non-blocking) and we can attach success/error callbacks to the `ListenableFuture`:

```
public void sendToKafka(OutDTO data) {
  kafkaProducerTemplate.send(outTopic, data)
  .addCallback(
  // success callback,
  // error callback
  );
}
```

A bean method can be annotated as `@KafkaListener` in order to receive messages. By default, it will receive messages from topics on a single thread.

```
public class Listener {

  @KafkaListener(id = "my-consumer-id", topics = "my-topic")
  public void listen(InDTO data) {
    ...
  }
}
```

Example of consumer configuration:

```
@Bean
public ConsumerFactory<String, InDTO> consumerFactory() {
  return new DefaultKafkaConsumerFactory<>(consumerConfigs());
}

@Bean
public Map<String, Object> consumerConfigs() {
  Map<String, Object> props = new HashMap<>();
  props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "
 localhost:9092");
  props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
 StringDeserializer.class);
  props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
 JsonDeserializer.class);

  return props;
}
```

2.3.8 Reactor Kafka

Reactor Kafka provides a reactive API for consuming and producing messages from/to Kafka using a functional API. Reactor Kafka is based on

Project Reactor and the Kafka Producer/Consumer API.

> "The value proposition for Reactor Kafka is the efficient utilization of resources in applications with multiple external interactions where Kafka is one of the external systems. End-to-end reactive pipelines benefit from non-blocking back-pressure and efficient use of threads, enabling a large number of concurrent requests to be processed efficiently. The optimizations provided by Project Reactor enable the development of reactive applications with very low overheads and predictable capacity planning to deliver low-latency, high-throughput pipelines." [16]

Reactor Kafka exposes two main interfaces for producing and consuming messages: `KafkaSender` and `KafkaReceiver`.

Reactive Kafka Sender

`KafkaSender` is used to send messages. `KafkaSender` is thread-safe, so it is possible to start multiple senders improving the throughput. Each `KafkaSender` is associated with one `KafkaProducer`. In order to configure the `KafkaSender`, an instance of `SenderOptions` is passed to its builder.

```
SenderOptions<String, OutDTO> senderOptions = SenderOptions.
 create(producerProps);
KafkaSender<String, OutDTO> sender = KafkaSender.create(
 senderOptions);
```

After creating a `KafkaSender` instance, we can create a flux of messages to be sent to Kafka. These messages are represented as `SenderRecords`, which are Kafka `ProducerRecords` with extra metadata for matching send results to records. A `ProducerRecord` includes a key/value pair and the name of the Kafka topic where the message will be sent. When the send operation completes or fails, it generates a `SendResult` for the record. To send the outbound `Flux` to Kafka, we use the `subscribe()` method to request the upstream to send the records to Kafka.

```
sender.send(outboundFlux)
.doOnError(e-> logger.error("Send failed", e))
.doOnNext(r -> logger.info("Message #%d send response: %s\n",
 r.correlationMetadata(), r.recordMetadata()))
.subscribe();
```

Reactive Kafka Receiver

Messages stored in Kafka topics are consumed using the reactive receiver `KafkaReceiver`. A `KafkaReceiver` instance is associated with an instance of `KafkaConsumer`. Since the underlying `KafkaConsumer` cannot be accessed simultaneously by multiple threads, `KafkaReceiver` is not thread-safe. An instance of `ReceiverOptions` is used to create a receiver.

```
ReceiverOptions<String, InDTO> receiverOptions =
ReceiverOptions.<String, InDTO>create(consumerProps)
.subscription(Collections.singleton(topic));
```

After configuring the necessary receiver options, a new `KafkaReceiver` instance can be generated using those options to consume inbound `Flux` of messages. When the inbound `Flux` has been subscribed to, an instance of the `KafkaConsumer` will be created.

```
Flux<ReceiverRecord<String, InDTO>> inboundFlux =
KafkaReceiver.create(receiverOptions)
.receive();
```

The `Flux` delivers each incoming message as a `ReceiverRecord`. A `ReceiverRecord` is composed of the `ConsumerRecord` obtained from `KafkaConsumer` and a committable `ReceiverOffset`. Once the message has been processed, the offset must be acknowledged since unacknowledged offsets will not be committed.

```
inboundFlux.subscribe(r -> {
  logger.info("Received message: %s\n", r);
  r.receiverOffset().acknowledge();
});
```

2.4 Java Management Extensions (JMX)

The complexity brought by distributed systems reflects in the management of these. We have to choose the most suitable management solution, what standards our solution should conform and the amount of effort required to enable the management of the application [17]. Java Management Extensions (JMX) was designed to address these issues in the context of applications written for the Java platform. Since version 5.0 JMX has been part of J2SE and a trademark of Oracle Corporation. To manage a system, we need to monitor and control its resources. The JMX architecture allows us to make the resources of a system manageable and, as consequence, to make the system manageable. The JMX architecture is composed of three levels: instrumentation level, agent level, and distributed level.

2.4.1 The Instrumentation Level

The instrumentation level is the closest to the application. This level instruments the resources to become manageable according to four different strategies: standard, dynamic, model, and open. A resource to be managed by JMX must provide a management interface. The management interface comprehends five metadata:

- Attributes
- Constructors
- Operations
- Parameters
- Notifications

The JMX agent will interact with a resource through this interface.

MBean

A resource instrumented to be manageable through JMX is called MBean (the "M" stands for manageable). To be managed, a resource must be compliant with four requirements:

- the resource state must be described through getters and setters

- the resource must be instrumented by one of the possible JMX MBean types (i.e. standard, dynamic, model, or open). This will make a resource an MBean
- the MBean must provide a public constructor
- the MBean must not be abstract

For this thesis, the details of the JMX MBean types along with the JMX notification are not described, and we refer the reader to the existing literature.

2.4.2 The Agent Level

The Agent Level comprises the MBean server and the JMX agent services (the M-Let service, monitoring services, the timer service, and the relation service).

The MBean server

The MBean server is the registry for MBeans. This registry is accessed by the JMX agent through the `MBeanServer` interface. To register an MBean in the registry, the MBean must be associated with an object name. The object name uniquely identifies the MBean within the registry and is implemented by the `ObjectName` JMX class. The object name assigned to the MBean is used for the communication between the JMX agent and the MBean. The MBean server receives requests through its implementation of the `MBeanServer` interface. If the agent wants to retrieve values of an MBean's attributes, it passes the corresponding object name to the appropriate method of the `MBeanServer` interface. The object name is then used by the MBean server as a lookup into the registry, and the corresponding values are returned to the agent.

2.4.3 The Distributed Level

The distributed level is the outmost level in the JMX architecture. The JMX agents are made available to the external environment through this level. There are two ways of enabling distributed interaction:

- using adapters that permit access to MBeans through various protocols, such as HTTP

- using JMX agents connectors which expose the agent API to distributed technologies like Java RMI

To access the MBean server, we can use protocol adapters and connectors. Connectors are more suitable for the development of a management application. Connectors provide a client from which to make an RPC to the MBean server of the JMX agent.

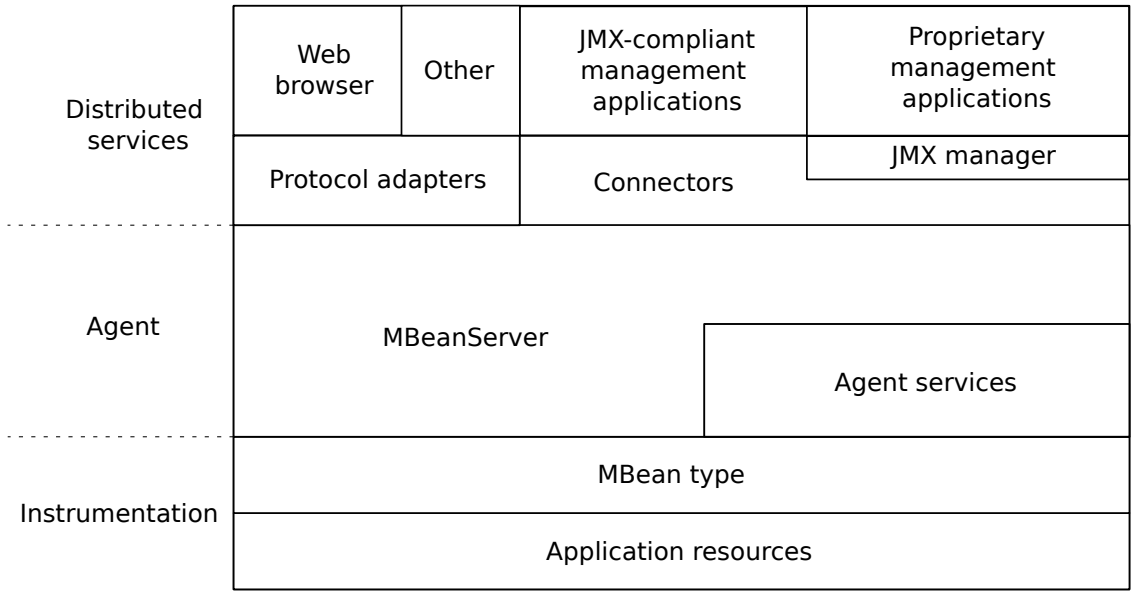

Figure 2.14. JMX architecture

The MBeanServer interface

The MBean server provides methods for creating, registering, manipulating, and finding MBeans through its `MBeanServer` interface. In this thesis, we focus on methods used to get values of attributes of an MBean. The method `Object getAttribute(ObjectName name, String attribute)` returns the value of the attribute of the MBean represented by the object name.

2.4.4 Communicating with JMX agents using the RMI connector

A connector includes two components: one residing in the JMX agents, and the other is made available to client-side applications. The monitoring application communicates with the JMX agent using the client-side component

to contact the server-side component. The RMI connector is an MBean that provides an RMI client. Through the RMI client is possible to connect to the RMI server and invoke the methods declared in the `MBeanServer` interface.

Chapter 3

Benchmark Design

3.1 Benchmark Design

The system under test is a core-banking application that performs data ingestion. The data-ingestion application consumes events from a Kafka topic, maps the input event into an output event, and publishes the result of the mapping into another topic for consumption by an external bank platform. In order to simulate the source of the input events, an *event generator* application has been created.

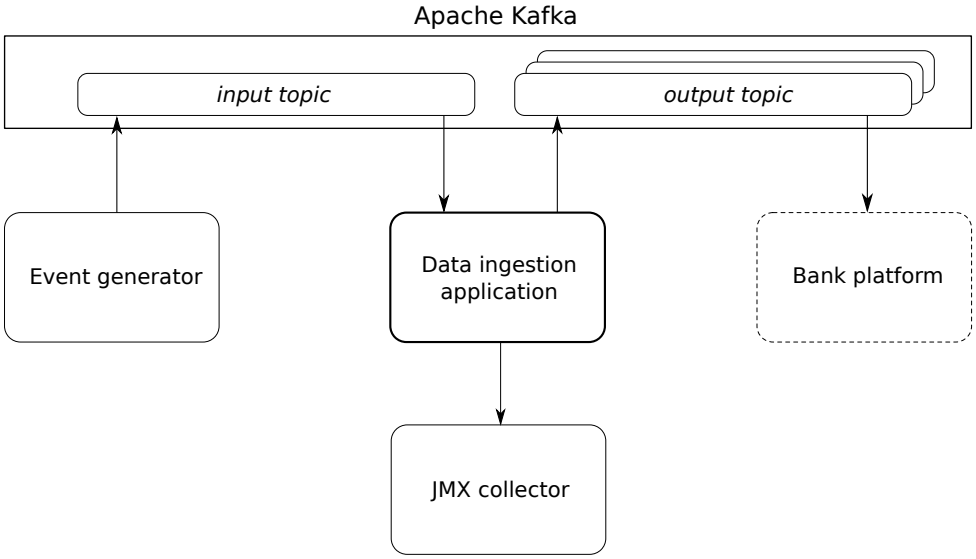

Figure 3.1. Interaction model of data pipeline and monitoring

3.1.1 Data

The data used in the benchmark comes from the core-banking domain. The input event is called a regulation request i.e. a transaction representing an inbound or outbound operation performed on a bank account. The event generator publishes events that follow the real structure of a regulation request. For the sake of simplicity, the published regulation requests are all valid. This means that the result of the validation performed by the data-ingestion application will be always successful. The rate of published data can be set in the parameter of the event generator. After validation, the regulation request is mapped into an output event that represents the formal request to the bank platform.

3.1.2 Processing Pipeline

In the processing pipeline, each input event is mapped to an output event. The processing pipeline of the system under test is composed of the following operations:

1. Ingest: messages are consumed from a Kafka input topic.
2. Parse: the JSON corresponding to the message is parsed into a POJO.
3. Validate: the POJO is validated against the business rules.
4. Mapping: the validated POJO is mapped into an output event.
5. Publishing: the output event is published onto the output topic.

In the actual implementation, the output event is built by querying external services. In our implementations of the pipeline, we mock the response of these services.

3.2 Kafka Clients

The data-ingestion application is basically a Kafka client. Kafka clients can consume messages from Kafka topics or can publish messages addressing a Kafka topic. The data-ingestion application performs both operations. In fact, after the consumption of a regulation request event, the latter is validated and mapped into an output event to be published into another Kafka topic. Kafka consumer and producer configuration is kept as default for both

traditional and reactive implementations. Exceptions are the message deserializers and serializers that are JSON for both versions. Moreover, the *acks* setting of the producers is set to *"all"*. This provides the strongest available guarantee for record acknowledgment. This means that the messages sent by the producer will not be lost.

3.2.1 Traditional Pipeline

The traditional version of the data-ingestion application is a Kafka client implemented using the Spring for Apache Kafka API [15]. The annotation `@KafkaListener` provides a mechanism for converting the incoming message, that is in a JSON format, into the POJO whose type is specified in the listener parameter. The input parameter of the listener represents an input event received from the Kafka topic specified in the parameters of the annotation. If the regulation request event received does not contain any validation errors, the message is mapped into a `CreatePostingInstructionBatchRequest` representing the output event. The bean `kafkaProducerTemplate` of type `KafkaTemplate<String, CreatePostingInstructionBatchRequest>` is used to send the output message to a topic from which the bank platform is listening. We attach success and failure callbacks to the send operation in order to log its result.

```
1  @KafkaListener(id = "default", topics = "${CONSUMER_TOPIC}")
2  private void consumeRegulationRequestEvent(
       RegulationRequestEvent regulationRequestEvent) {
3    logger.info("[REGULATION-REQUEST] Event received: " +
       regulationRequestEvent);
4    List<String> validationErrors = validateRegulationRequest(
       regulationRequestEvent);
5
6    if (CollectionUtils.isEmpty(validationErrors)) {
7      PerforabilityResponse perforabilityResponse =
       mockPerforabilityResponse();
8      CreatePostingInstructionBatchRequest request =
       createPostingInstructionBatchRequest(regulationRequestEvent,
       perforabilityResponse);
9
10     kafkaProducerTemplate.send(outTopic, request)
11     .addCallback(
12       result -> logger.info("[REGULATION-REQUEST] Posting
       request:\n{}", new Gson().toJson(request)),
13       (e) -> logger.error("something bad happened while
       consuming : {}", e.getMessage())
```

```
14      );
15    }
16  }
```

3.2.2 Concurrent Traditional Pipeline

Spring for Apache Kafka provides two implementations of `MessageListenerContainer`:

- `KafkaMessageListenerContainer`
- `ConcurrentMessageListenerContainer`

In the previous pipeline, we used the `KafkaMessageListenerContainer`, which is the default. The `KafkaMessageListenerContainer` receives all messages on a single thread.
Now we will introduce the `ConcurrentMessageListenerContainer`, which enables multi-threaded consumption by delegating one or more `KafkaMessageListenerContainer`.

```
1  @Bean
2  KafkaListenerContainerFactory<ConcurrentMessageListenerContainer
       <String, RegulationRequestEvent>>
3  kafkaListenerContainerFactory() {
4    ConcurrentKafkaListenerContainerFactory<String,
       RegulationRequestEvent> factory =
5    new ConcurrentKafkaListenerContainerFactory<>();
6    factory.setConsumerFactory(consumerFactory());
7    factory.setConcurrency(8);
8    return factory;
9  }
10
11 @Bean
12 public ConsumerFactory<String, RegulationRequestEvent>
       consumerFactory() {
13   return new DefaultKafkaConsumerFactory<>(consumerConfigs(),
14   new StringDeserializer(),
15   new JsonDeserializer<>(RegulationRequestEvent.class, false));
16 }
17
18 @Bean
19 public NewTopic topicIn1() {
20   return TopicBuilder.name("classic_in_topic")
21   .partitions(8)
22   .build();
23 }
```

We provide an instance of `ConcurrentMessageListenerContainer` using a `ConcurrentKafkaListenerContainerFactory`. We set the concurrency level i.e. the number of consumers in the consumer group, to 8. The choice of this number is because the reactive pipeline using the `parallel()` operator (which we introduce in section 3.3.3), uses a pool of workers sized to the number of CPU cores (in our case, 8). We also define eight partitions during the topic creation so that partitions are evenly assigned to the consumer group members.

3.2.3 Reactive Pipeline

Since the data-ingestion application consumes messages to be transformed and sent to Kafka, we build the reactive pipeline according to a source and sink model. The source is represented by the `sender.send()` operation, the sink is implemented by the `receiver.receive()` operation that is given in input to the sender. The `receiver` is of type `KafkaReceiver<String, RegulationRequestEvent>` while the `sender` is a `KafkaSender<String, CreatePostingInstructionBatchRequest>`. The `receive()` method starts a Kafka consumer that consumes records from a Kafka topic. Consumed records are returned in a `Flux<ReceiverRecord<K, V>`. Then we apply the validation in the `filter()` operator. Consumed records are transformed into the corresponding `CreatePostingInstructionBatchRequest` and mapped into a `SenderRecord` required by the `send()` operation. The actual data flow from `Publisher` to `Subscriber` is triggered only once the `subscribe()` method is invoked.

```
1  private Flux<SenderResult<ReceiverOffset>> pipeline() {
2
3    return sender
4      .send(receiver
5        .receive()
6        .doOnNext(m -> logger.info("[REGULATION-REQUEST] Event
       received: " + m.value()))
7        .filter(m -> isValidRequest(m.value()))
8        .map(m -> SenderRecord.create(transform(m.value()), m.
       receiverOffset())))
9      .doOnNext(m -> m.correlationMetadata().acknowledge());
10 }
11
12 public ProducerRecord<String,
       CreatePostingInstructionBatchRequest> transform(
       RegulationRequestEvent r) {
```

```
13    PerforabilityResponse perforabilityResponse =
       mockPerforabilityResponse();
14    CreatePostingInstructionBatchRequest c =
       createPostingInstructionBatchRequest(r, perforabilityResponse
       );
15    return new ProducerRecord<>(outTopic, c.getId().toString(), c)
       ;
16  }
17
18  @Override
19  public void run(String... args) {
20    pipeline().subscribe();
21  }
```

3.3 Thread Scheduling in the Reactive Pipeline

Starting from the base implementation of the reactive pipeline, we investigate how to further improve its performance by exploiting the features provided by Reactor for multi-threading execution. We will explore three modifications of the reactive stream processing: using the `publishOn` operator, combining the `subscribeOn` and `publishOn` operators, and using the `parallel` operator. The `Schedulers` employed in the different versions are the ones provided by the `Schedulers.parallel()` and `Schedulers.boundedElastic()` factory methods. The `parallel()` scheduler is optimized for short-lived CPU-intensive tasks. This is the case of the `filter` and `map` operations in our reactive pipeline. In the real scenario, where an external service is queried in order to build the mapped event, `Schedulers.boundedElastic()` would be more appropriate. However, in our pipeline, the query to the external service is mocked.

3.3.1 Using the publishOn operator

A first optimization consists of introducing an asynchronous boundary within the flux of the consumed events. We decouple the logging of the received event from the validation and transformation. The filter (validation) and map (transformation) operations are executed by a different poll of workers, optimized for non-blocking CPU-intensive tasks.

```
1  private Flux<SenderResult<ReceiverOffset>> pipeline() {
2    return sender
```

```
3       .send(receiver
4         .receive()
5         .doOnNext(m -> logger.info("Event received:" + m.value()))
6         .publishOn(Schedulers.parallel())
7         .filter(m -> isValidRequest(m.value()))
8         .map(m -> SenderRecord.create(transform(m.value()), m.
      receiverOffset())))
9       .doOnNext(m -> m.correlationMetadata().acknowledge());
10  }
```

We want to highlight that in the previous code, the event *n* will be validated and mapped before the event *n+1* i.e. a strict ordering exists for all events. This is a property of Reactive Streams named *serializability*. However, the logging of received events is independent of the processing that follows the `publishOn` operator. Summarizing, the main optimization in the previous code consists of changing the worker on which the CPU-intensive operations will be executed.

3.3.2 Using subscribeOn and publishOn operators

Further optimization of the previous pipeline consists in changing the worker on which the subscription happens. This means we change the worker that invokes the `subscribe()` method. We remember that when using reactive programming, the actual data flow from `Publisher` to `Subscriber` is triggered only by subscribing. We introduce the `subscribeOn()` operator to select a more convenient worker on which the subscription will happen. In this version of the pipeline, we use the `Schedulers.boundedElastic()` scheduler that is optimized for blocking tasks and longer execution.

```
1   private Flux<SenderResult<ReceiverOffset>> pipeline() {
2     return sender
3       .send(receiver
4         .receive()
5         .subscribeOn(Schedulers.boundedElastic())
6         .doOnNext(m -> logger.info("Event received:" + m.value()))
7         .publishOn(Schedulers.parallel())
8         .filter(m -> isValidRequest(m.value()))
9         .map(m -> SenderRecord.create(transform(m.value()), m.
      receiverOffset())))
10      .doOnNext(m -> m.correlationMetadata().acknowledge());
11  }
```

3.3.3 Using the parallel operator

Since Reactor Fluxes are characterized by serializability, elements of the `Flux` are processed one by one. If we want to process Fluxes in parallel, we can use the `parallel()` operator provided by the `Flux` API.

```
1  private Flux<SenderResult<ReceiverOffset>> pipeline() {
2    return sender
3      .send(receiver
4        .receive()
5        .parallel()
6        .runOn(Schedulers.parallel())
7        .doOnNext(m -> logger.info("Event received:" + m.value()))
8        .filter(m -> isValidRequest(m.value()))
9        .map(m -> SenderRecord.create(transform(m.value()), m.
     receiverOffset())))
10     .doOnNext(m -> m.correlationMetadata().acknowledge());
11 }
```

After applying the parallel() operator on line 6, the `Flux` of incoming events is split into parallel sub-streams represented by the `ParallelFlux` abstraction. The elements of the source `Flux` are balanced over the group of Fluxes. The `runOn` operator allows the application of the `publishOn` operator on each `Flux` belonging to the `ParallelFlux`. In the above code, we use the `Schedulers.parallel()` scheduler, which provides a pool of workers for CPU-bound tasks. The size of the pool depends on the number of CPU cores. We want to emphasize that while processing the incoming `Flux` happens in parallel, we still have a single Kafka consumer.

3.4 Metrics

For each run, we compute several metrics: run-time, record consumed rate, records lag, record send rate, and memory/CPU utilization of the process. Consumer and producer metrics are supported by Apache Kafka. More precisely, the Java clients use Kafka Metrics, which is a metrics registry exposing metrics via JMX. The easiest way to browse the available metrics with JMX is to launch *jconsole* and select the kafka client [18]. However, we will build a custom Java monitor that will collect the metrics of interest directly querying JMX. By default Apache Kafka disables remote JMX [19]. We can enable remote JMX programmatically by selecting the JMX port and specifying security properties:

```
-Dcom.sun.management.jmxremote=true -Dcom.sun.management.
   jmxremote.port=9091 -Dcom.sun.management.jmxremote.
   authenticate=false -Dcom.sun.management.jmxremote.ssl=false
```

```
JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/
   rmi://localhost:" + port + "/jmxrmi");
JMXConnectorFactory.connect(url, null);
JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
jmxc.connect();
MBeanServerConnection mBeanServerConnection = jmxc.
   getMBeanServerConnection();
```

3.4.1 Run-time

We define run-time as the difference between the first observation timestamp of consumption and the last observation timestamp of production. This run-time metric indicates how much time the data-ingestion application takes to consume, process, and publish the entire amount of messages published by the event generator in each run.

3.4.2 Records Consumed Rate

The records consumed rate metric represents the average number of records consumed per second. In order to get this metric, we use the *records-consumed-rate* attribute of the *kafka.consumer:type=consumer-fetch-manager-metrics,client-id="client-id"* MBean exposed by the Consumer Fetch Metrics of Apache Kafka [20]. The *client-id* is the id assigned during the configuration of the Kafka consumer.

```
Object recordsConsumedRate;
ObjectName consumerMBean = new ObjectName("kafka.consumer:type=
   consumer-fetch-manager-metrics,client-id=consumer-default-1")
   ;
recordsConsumedRate = mBeanServerConnection.getAttribute(
   consumerMBean, "records-consumed-rate");
```

3.4.3 Max Records Lag

The *records-lag-max* attribute of the *kafka.consumer:type=consumer-fetch-manager-metrics,client-id="client-id"* MBean represents the maximum lag

in terms of records. It is a good indicator of the capability of the consumer to keep up with the producer.

```
recordsLagMax = mBeanServerConnection
  .getAttribute(consumerMBean, "records-lag-max");
```

3.4.4 Record Send Rate

The record send rate metric represents the average number of records sent per second. We get this metric by fetching the *record-send-rate* attribute of the *kafka.producer:type=producer-metrics,client-id="client-id"* MBean exposed by the Producer Sender Metrics of Apache Kafka [21]. The *client-id* is the id assigned during the configuration of the Kafka producer.

```
Object recordSendRate;
ObjectName producerMBean = new ObjectName("kafka.producer:type=
   producer-metrics,client-id=producer-1");
recordSendRate = mBeanServerConnection.getAttribute(
   producerMBean, "record-send-rate");
```

3.4.5 Memory Utilization

We collect the heap memory usage of the data-ingestion application by scraping the JMX metrics exposed by the process. The memory MBean is identified by the *java.lang:type=Memory* ObjectName [22]. The attribute representing the heap memory usage is *HeapMemoryUsage*. In particular, we are interested in the amount of memory used (in bytes) during the execution, so we will fetch the *used* field [23].

```
CompositeData cd;
Object heapMemoryUsage;
ObjectName memoryMBean = new ObjectName("java.lang:type=Memory")
   ;
heapMemoryUsage = mBeanServerConnection.getAttribute(memoryMBean
   , "HeapMemoryUsage");
cd = (CompositeData) heapMemoryUsage;
Object heapMemoryUsed = cd.get("used")
```

3.4.6 CPU Utilization

The CPU usage of the JVM is retrieved from the *ProcessCpuLoad* attribute of the *java.lang:type=OperatingSystem* MBean [24].

```
Object processCpuLoad;
ObjectName operatingSystemMBean = new ObjectName("java.lang:type
    =OperatingSystem");
processCpuLoad = mBeanServerConnection.getAttribute(
    operatingSystemMBean, "ProcessCpuLoad");
```

3.5 Workloads

Following the approach used by [4], we evaluate the performance of the different implementations of Kafka clients under two processing scenarios: constant rate and burst at startup.

3.5.1 Constant Rate Workload

We use this workload to stress Kafka clients. The event generator publishes events at a high rate so that the Kafka clients will not be able to keep up with the demand, and events will be queued on Kafka for several seconds before being processed by the Kafka clients. The event generator will publish events with a constant rate of 1 000 records per second for 80 seconds.

3.5.2 Burst at Start-Up Workload

With this workload, we simulate the scenario in which the Kafka client resumes consuming events after a downtime. We publish a burst of events, and after the event generator completes its production, we start the Kafka client that will consume events from the earliest offset. In the case of single-threaded consumption, the burst dimension is 100 000 events, while in multi-threaded consumption, 200 000 events.

3.6 Infrastructure

The data ingestion application, the event generator, the JMX collector, and the Apache Kafka cluster are hosted on the same machine. The Apache Kafka cluster is deployed on a docker container.

Chapter 4

Results

In this section, we visualize and analyze data obtained from the run of different versions of the data-ingestion application according to two workloads: constant rate and burst at startup. The flavors of the data-ingestion application are of two main types: traditional and reactive. In turn, we evolve the reactive flavor into three other flavors that exploit the thread scheduling model provided by Project Reactor. For each version, we present some metrics related to the performance of consumer, producer and the JVM on which the data-ingestion application is run. In the following, we give a brief description of the data-ingestion flavors used:

- **Traditional**: using the abstractions provided by the Spring for Apache Kafka API.
- **Reactive**: consumes and produces events using the Reactor Kafka API.
- **Concurrent Traditional**: same as *Traditional* but implementing multi-threading consumption
- **publishOn**: same as *Reactive*, but changing the worker on which part of the pipeline is processed.
- **subscribeOn + publishOn**: same as *publishOn*, but changing also the worker on which the subscription happens.
- **parallel**: same as *Reactive*, but transforming the consumption `Flux` into `ParallelFlux` in order to process them in parallel.

We distinguish in our comparison between *traditional* and *reactive* implementations, single-threaded consumption, and multi-threaded consumption.

Concurrent Traditional and *parallel* are multi-threaded implementations. However, in the case of the *parallel* version, the Kafka consumer is one, but the reactive stream of consumption is processed in parallel.

4.1 Constant Rate Workload

During constant rate workload, the event generator publishes events to a Kafka topic at a high constant rate. In our benchmark, the event generator publishes 1 000 events per second for 80 seconds. A total of 80 000 events are produced for each run.

4.1.1 Run-time

We remember that in our data pipeline, incoming events are consumed, validated, transformed, and finally published to an output topic. We consider our pipeline completely processed when all the events are published to the output topic for the consumption of the external bank platform. The *traditional* implementation is able to consume, process, and publish all events within about 130 seconds from the start of the workload. The base *reactive* implementation achieves better performance because the entire pipeline is handled in less than 118 seconds. The single-threaded flavors evolved from the reactive implementation present an even lower run-time.
The multi-threaded implementations show a similar run-time value, although the *parallel* flavor is almost 3 seconds slower with respect to *Concurrent Traditional* . It is worth noticing that under a constant rate workload, the multi-threaded implementations perform worse with respect to the optimized single-threaded reactive implementations.

Table 4.1. Run-time – Constant Rate Workload

version	Run-time
Traditional	130.74 s
Reactive	117.48 s
publishOn	112.49 s
subscribeOn + publishOn	108.47 s
Concurrent Traditional	116.01 s
parallel	118.53 s

4.1.2 Consumer Metrics

Figure 4.1 shows the consumption rate of single-threaded consumers over time. In all versions, the throughput presents a slow start during the first seconds of consumption, and then it grows steeply. All reactive flavors perform better with respect to the traditional one.

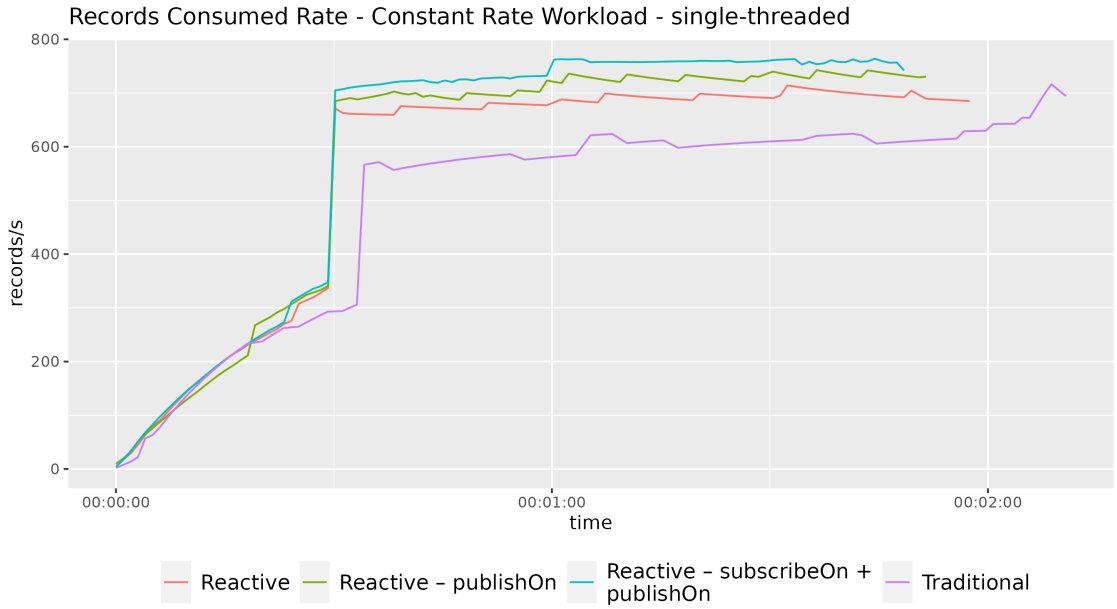

Figure 4.1. Records consumed rate with constant rate workload — single-threaded

Figure 4.2 illustrates the records consumed rate in case of multi-threaded consumption. The *parallel* implementation achieves lower values of maximum consumed rate with respect to *Concurrent Traditional* . However, the *parallel* flavor reaches the maximum throughput faster than *Concurrent Traditional*. This makes the *parallel* version achieve a higher average consumption rate, as shown in table 4.2.

Figures 4.3 and 4.4 visualize how the consumption rate observations are distributed among the different flavors. The initial part of observations is included, except for the *Concurrent Traditional* flavor, below 400 records per second. The cross mark represents the median value for the specific implementation. We can see that in the case of single-threaded consumption, we can increase the record consumption rate by almost 22% with respect to the *traditional* implementation.

Regarding multi-threaded consumption, the *parallel* flavor seems slower than *Concurrent Traditional* . The median of records consumed rate of *parallel* is 3.8% smaller than *Concurrent Traditional* . Moreover, *Concurrent Traditional* can reach higher maximum consumption values.

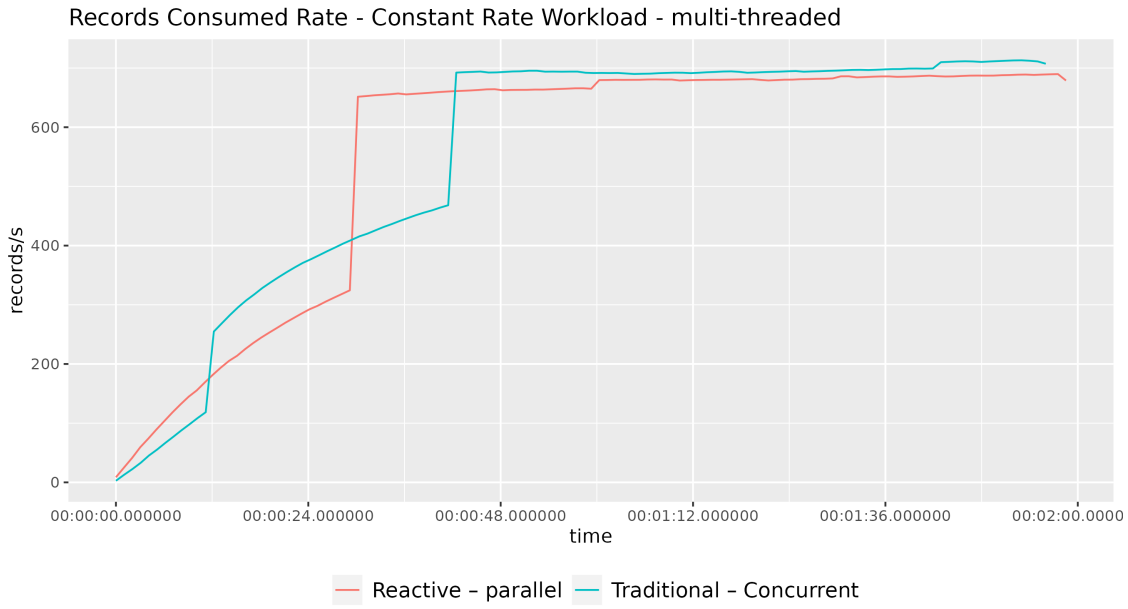

Figure 4.2. Records consumed rate with constant rate workload — multi-threaded

The Kafka metric *records-lag-max* is illustrated in figure 4.5. After some time, all single-threaded implementations, except for *subscribeOn + publishOn* , present a linear behavior in the growth of record lag. An increasing value of this metric is an indication that the consumer is not able to keep up with the producer i.e. the event generator. The *traditional* implementation presents the worst performance. The *reactive* base implementation achieved a substantial improvement. The *publishOn* version register even lower values of record lag. The *subscribeOn + publishOn* implementation presents zero lag. The figure of record lag in the case of multi-threaded consumption is not shown. In fact, the *parallel* and *Concurrent Traditional* versions present zero lag, meaning that the consumers are fully capable of keeping up with the production.

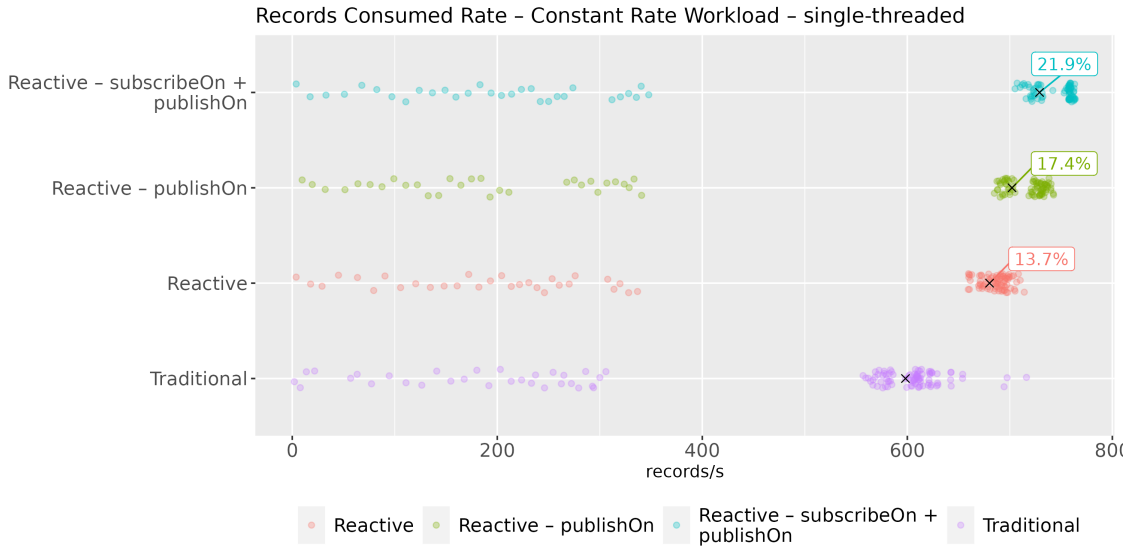

Figure 4.3. Distribution of records consumed rate with constant rate workload — single-threaded

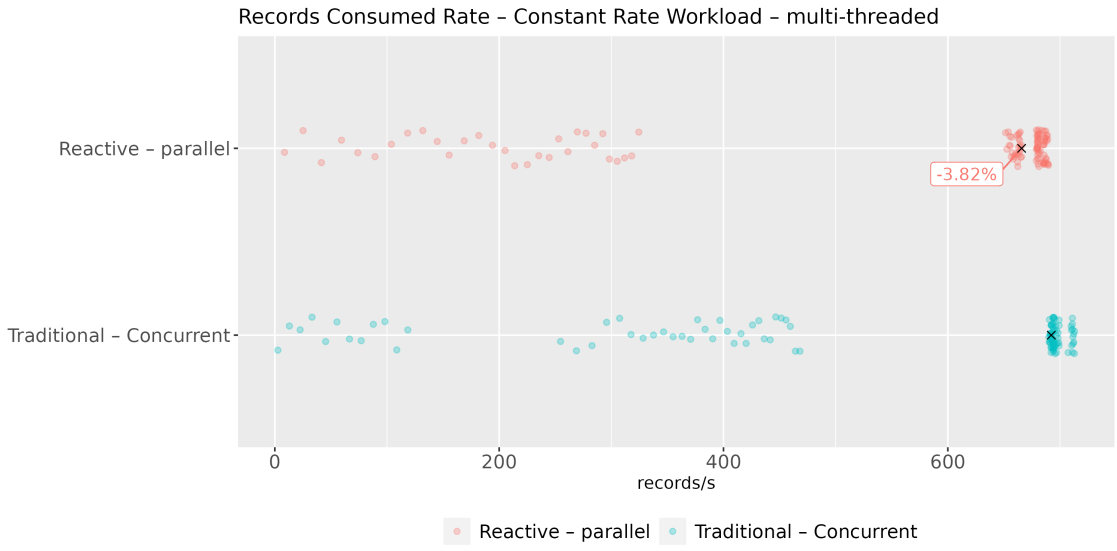

Figure 4.4. Distribution of records consumed rate with constant rate workload — multi-threaded

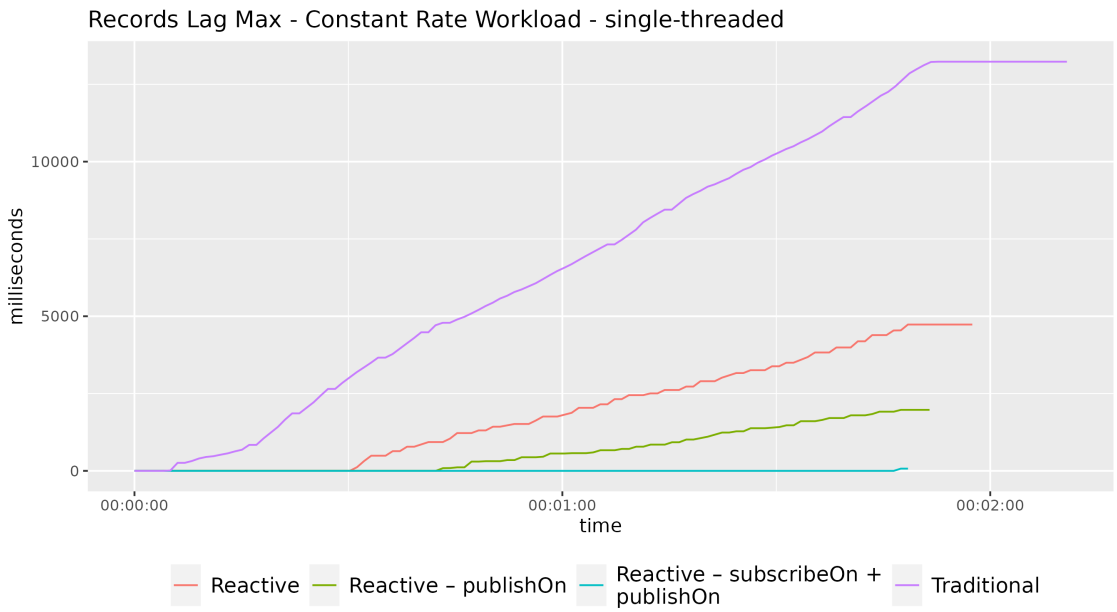

Figure 4.5. Records lag max with constant rate workload — single-threaded

Table 4.2. Consumer Performance – Constant Rate Workload

version	Mean Records Consumed Rate	Median Records Consumed Rate	Max Records Consumed Rate
Traditional	495.15	598.19	716.22
Reactive	552.07	680.23	714.09
publishOn	571.44	702.13	742.65
subscribeOn + publishOn	592.39	728.98	763.94
Concurrent Traditional	550.28	692.23	713.02
parallel	554.27	665.79	689.75

4.1.3 Producer Metrics

Among JMX metrics exposed by the Kafka producer, we consider the *record-send-rate* metric. The evolution of the record send rate over time is similar

to that of the records consumed rate in the previous section. However, in the case of single-threaded consumption, the reactive flavors achieve an even greater speed-up. The reactive *subscribeOn + publishOn* implementation can reach a median value of record send rate 25% higher than *traditional* .

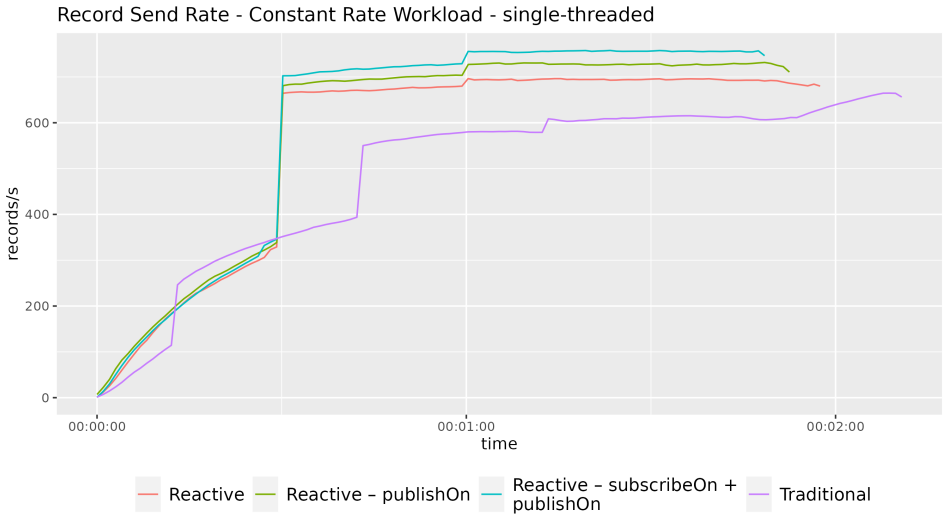

Figure 4.6. Record send rate with constant rate workload — single-threaded

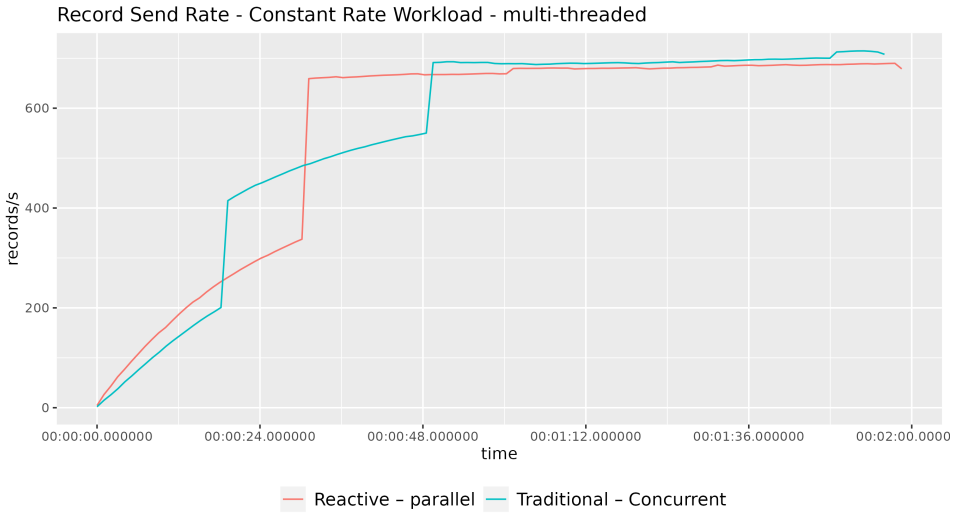

Figure 4.7. Record send rate with constant rate workload — multi-threaded

The gap between multi-threaded implementation diminishes. The median production value of *parallel* is 2.87% smaller than *Concurrent Traditional* . However, as in the consumption case, *parallel* achieves a greater mean record send rate with respect to *Concurrent Traditional* .

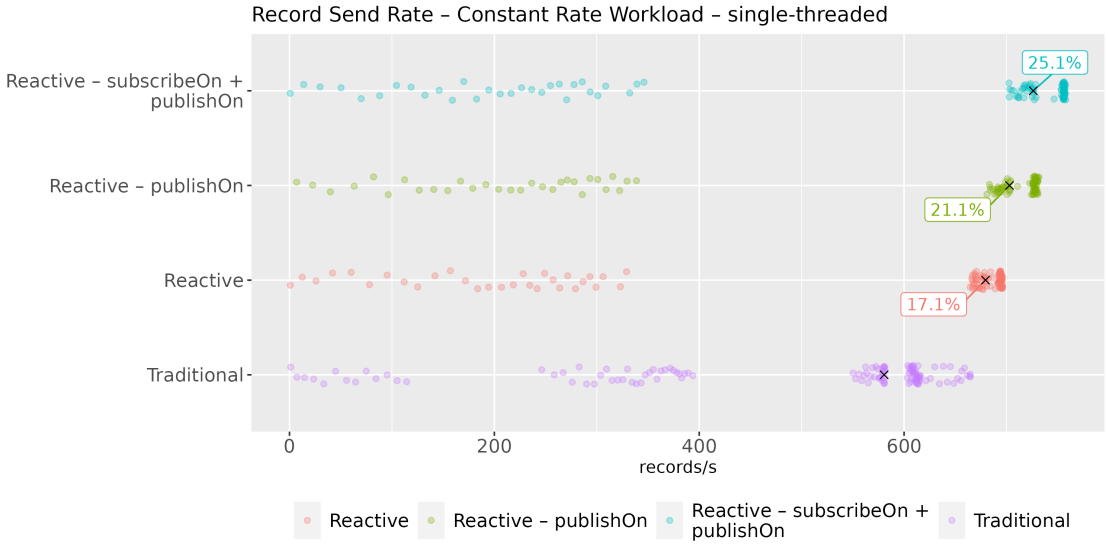

Figure 4.8. Distribution of record send rate with constant rate workload — single-threaded

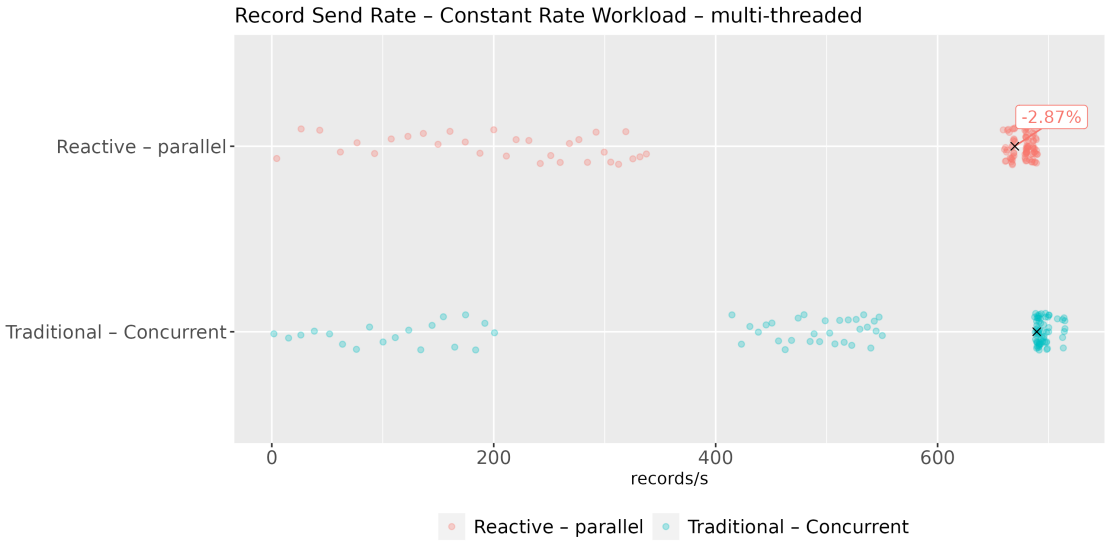

Figure 4.9. Distribution of record send rate with constant rate workload — multi-threaded

Table 4.3. Producer Performance – Constant Rate Workload

version	Mean Record Send Rate	Median Record Send Rate	Max Record Send Rate
Traditional	488.01	580.50	664.68
Reactive	559.69	679.51	696.38
publishOn	579.48	702.90	731.61
subscribeOn + publishOn	591.02	726.12	757.81
Concurrent Traditional	547.16	689.35	714.76
parallel	554.00	669.57	689.87

4.1.4 JVM Metrics

CPU usage over time is shown in figures 4.10 and 4.11. Figures 4.12 and 4.13 shows its distribution. Traditional implementations reach the highest CPU loads in both single-threaded and multi-threaded cases. The *reactive* base version achieves the lowest CPU load. The *subscribeOn + publishOn* implementation can use 10% of CPU less than *traditional* . At the same time, the *parallel* implementation achieves median values of CPU usage 12% smaller with respect to *Concurrent Traditional* .

Figures 4.14 and 4.15 illustrate mean values of heap memory usage and their dispersion in relation to the mean ($\mu \pm \sigma$).
In the case of single-threaded consumption, the highest mean memory usage value is reached by the *traditional* implementation. The *publishOn* version presents the highest maximum value and the widest dispersion in recorded values. This is likely due to the queue introduced by the `publishOn()` operator in the reactive pipeline. Finally, the *subscribeOn + publishOn* flavor presents the lowest mean and max memory utilization values. The *subscribeOn + publishOn* implementation can use on average 27% less memory with respect to *traditional* .
Regarding multi-threaded consumption, the *parallel* implementation use, on average, 50% less memory than *Concurrent Traditional* .

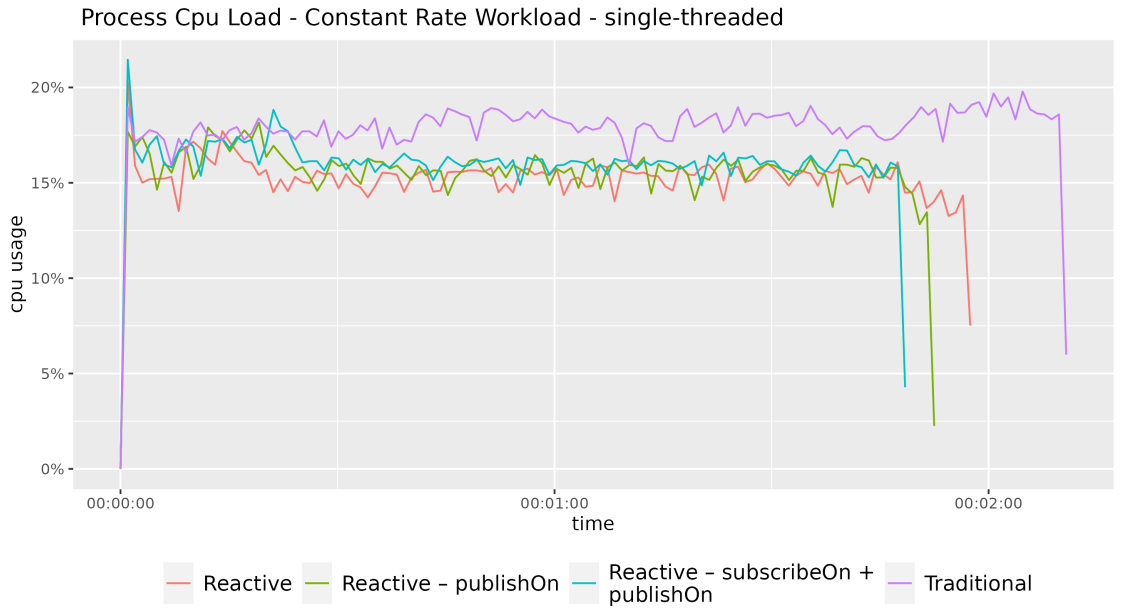

Figure 4.10. Process CPU load with constant rate workload — single-threaded

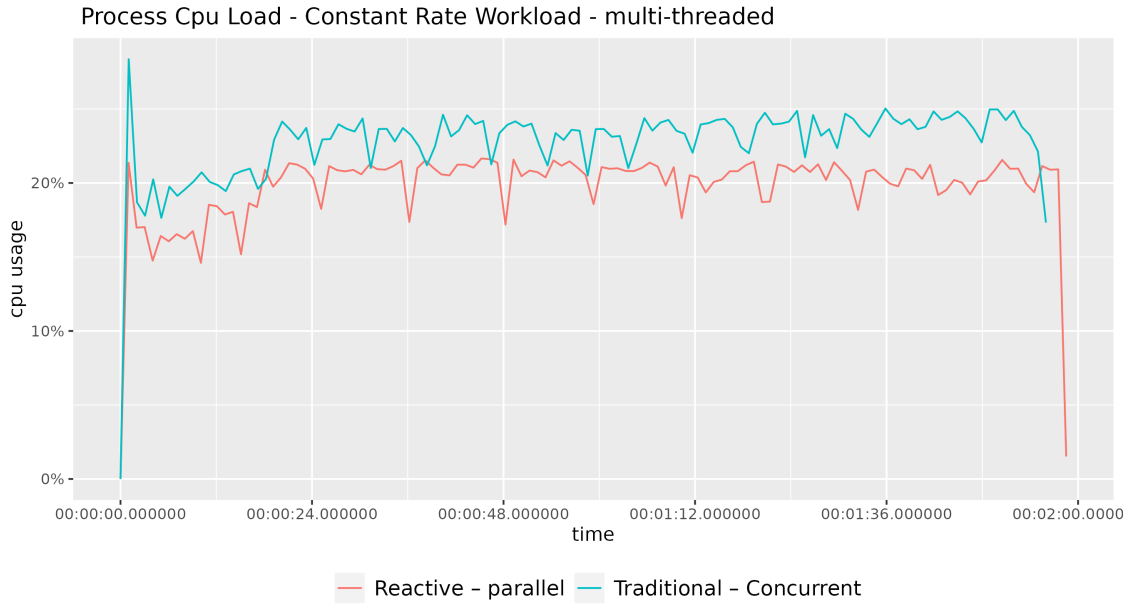

Figure 4.11. Process CPU load with constant rate workload — multi-threaded

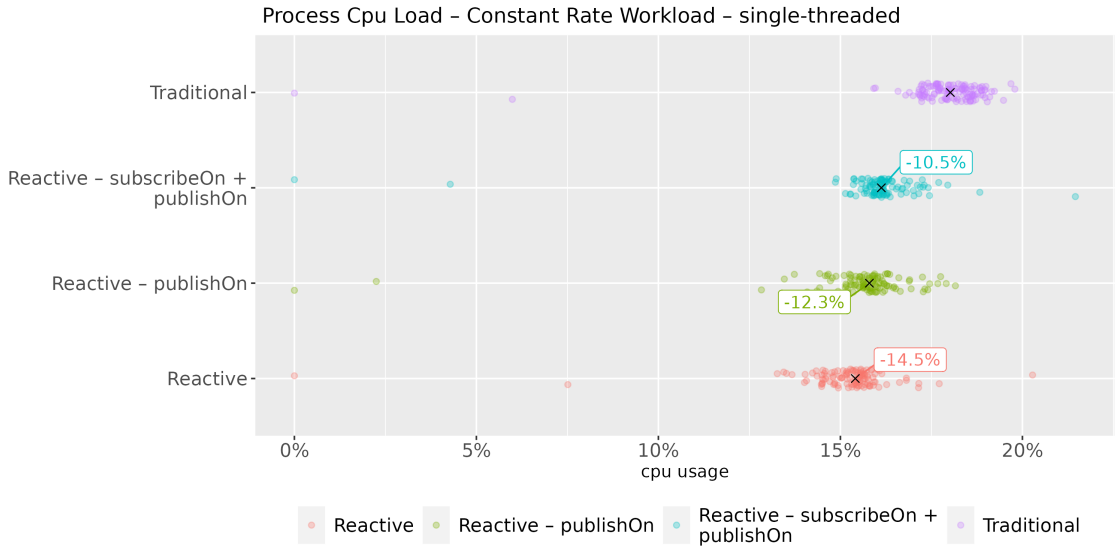

Figure 4.12. Distribution of process CPU load with constant rate workload — single-threaded

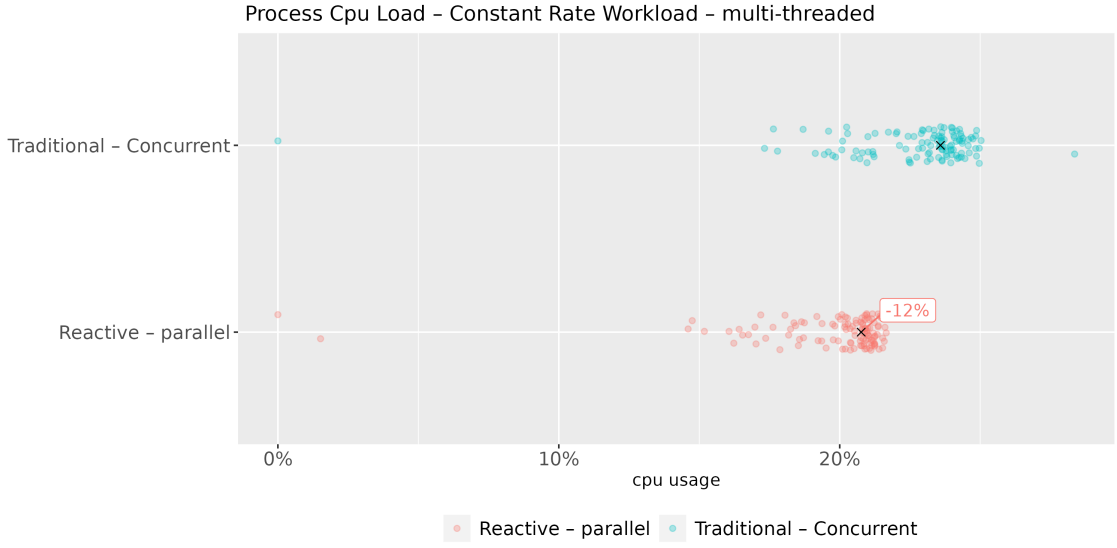

Figure 4.13. Distribution of process CPU load with constant rate workload — multi-threaded

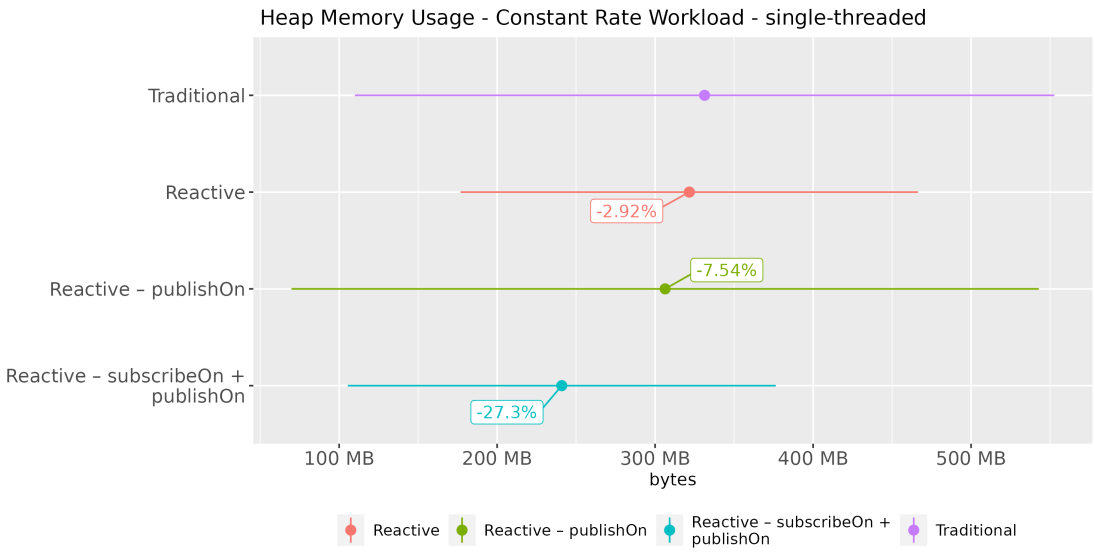

Figure 4.14. Heap memory usage with constant rate workload — single-threaded

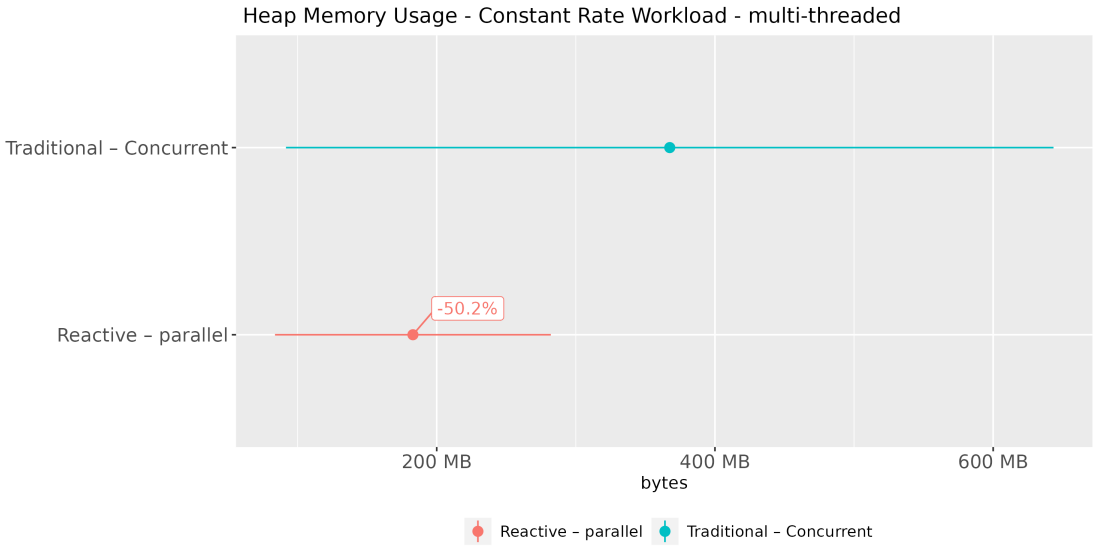

Figure 4.15. Heap memory usage with constant rate workload — multi-threaded

Table 4.4. Process Performance – Constant Rate Workload

version	Mean Memory Usage	Mean CPU Load	Max Memory Usage	Max CPU Load
Reactive	321.7MB	15.15%	670.1MB	20.28%
publishOn	306.3MB	15.53%	1.1GB	18.16%
subscribeOn + publishOn	240.9MB	15.98%	532.6MB	21.46%
Traditional	331.3MB	17.83%	1GB	19.79%
parallel	182.9MB	19.73%	405.3MB	21.66%
Concurrent Traditional	367.5MB	22.74%	1.3GB	28.36%

4.2 Burst At Start-Up Workload

During burst at start-up workload, 100 000 events — 200 000 in the multi-threaded case — are sent by the event generator while the data-ingestion application is down. Once the event generator has finished the publication, the data-ingestion application is launched. With burst at start-up workload, we simulate a failure of the data-ingestion application. After the failure, the application is restarted, but in the meantime, a great number of events have been published and not yet handled.

4.2.1 Run-time

The *traditional* version presents the longest run-time; this flavor consume, process, and publish the entire burst of events in about 120 seconds. The reactive implementations achieve a more significant speed-up. The *subscribeOn + publishOn* implementation processes the entire burst in about 104 seconds. Under burst workload, the multi-threaded implementations present a significant advantage with respect to the single-threaded implementations. The *Concurrent Traditional* flavor achieves a run-time of 90 seconds. The *parallel* flavor reaches an even lower run-time, 72 seconds.

Table 4.5. Run-time – Burst Workload

version	Run-time
Traditional	120.29 s
Reactive	111.27 s
publishOn	107.26 s
subscribeOn + publishOn	104.25 s
Concurrent Traditional	90.05 s
parallel	72.32 s

4.2.2 Consumer Metrics

Among the single-threaded flavors, the *subscribeOn + publishOn* version provides the best performance. As we can see from figure 4.18, the reactive *subscribeOn + publishOn* implementation reaches a median consumption rate

12% higher than *traditional* .
The records lag performance graphs are not presented because they are less meaningful in the case of the burst workload.
In the case of multi-threaded consumption, the *parallel* implementation records a median consumption rate almost 19% higher than *Concurrent Traditional*, as illustrated in figure 4.17.

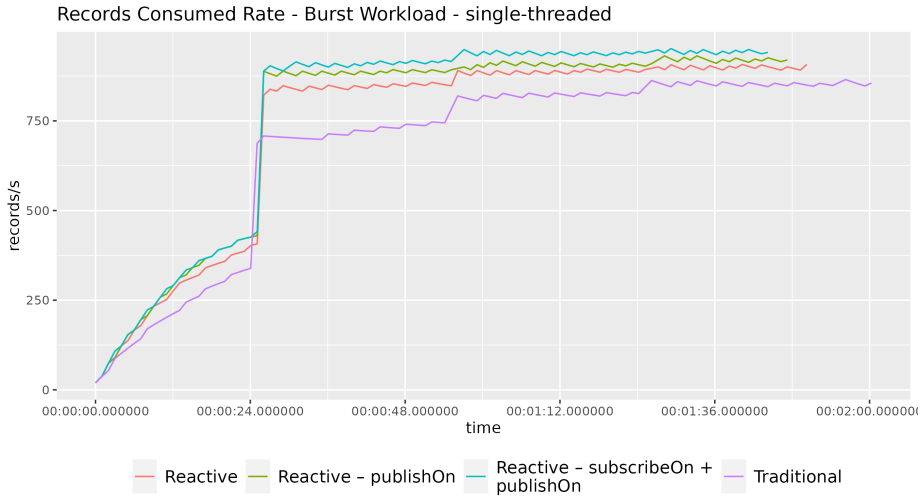

Figure 4.16. Records consumed rate with burst workload — single-threaded

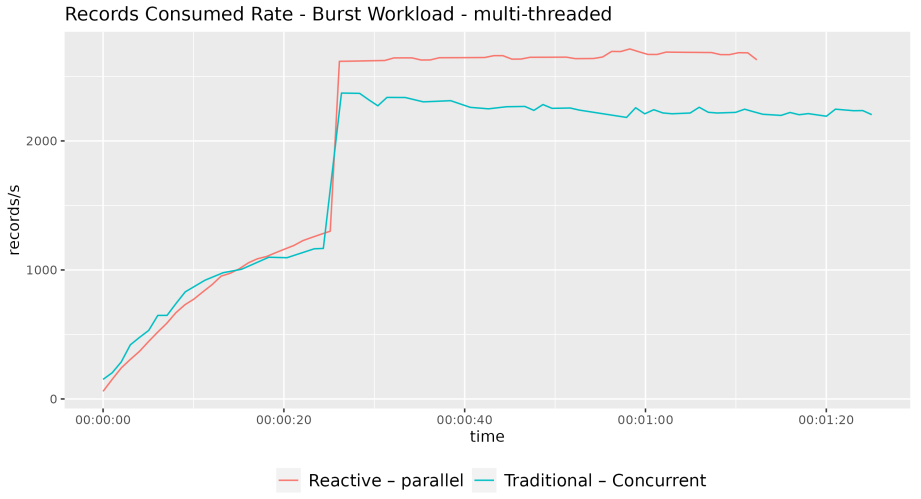

Figure 4.17. Records consumed rate with burst workload — multi-threaded

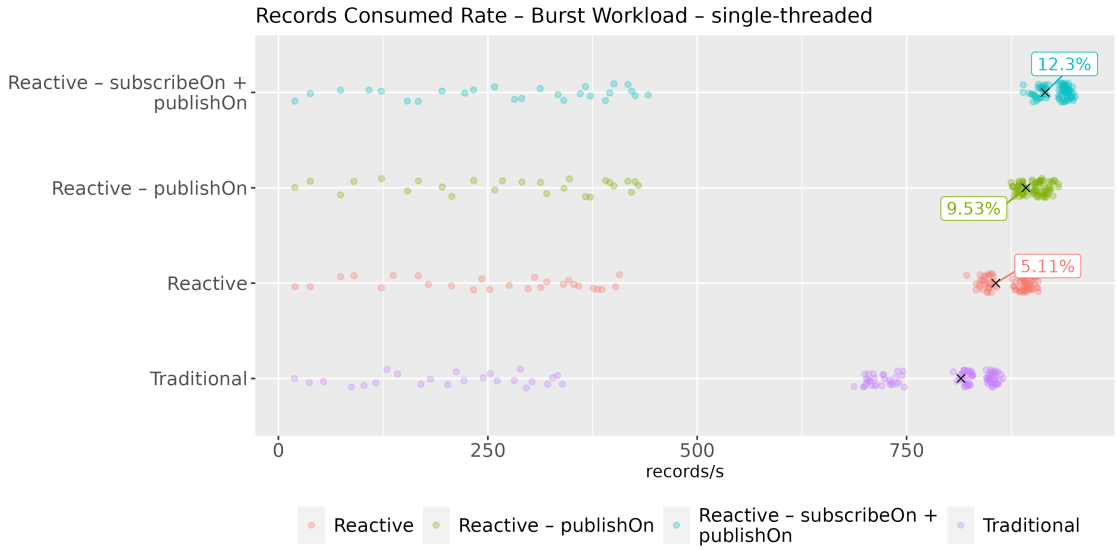

Figure 4.18. Distribution of records consumed rate with burst workload — single-threaded

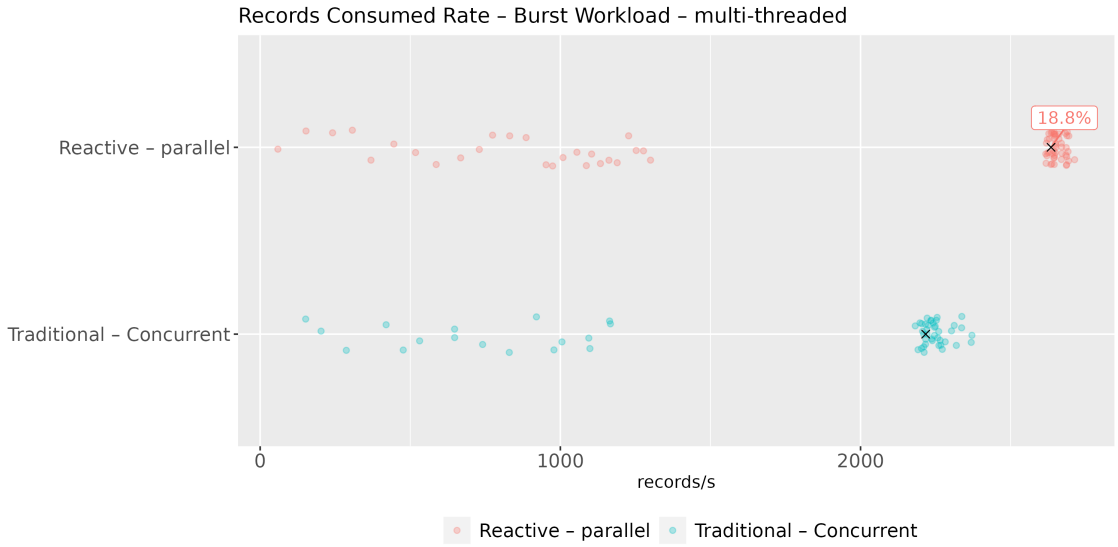

Figure 4.19. Distribution of records consumed rate with burst workload — multi-threaded

Table 4.6. Consumer Performance – Burst Workload

version	Mean Records Consumed Rate	Median Records Consumed Rate	Max Records Consumed Rate
Traditional	677.88	814.88	865.24
Reactive	728.61	856.48	907.42
publishOn	747.30	892.52	931.32
subscribeOn + publishOn	762.77	915.45	951.56
Concurrent Traditional	1.80×10^3	2.22×10^3	2.37×10^3
parallel	2.00×10^3	2.63×10^3	2.71×10^3

4.2.3 Producer Metrics

Concerning single-threaded implementations, as in the case of consumption, the *subscribeOn + publishOn* flavor attains higher throughput with respect to the other versions.
Regarding multi-threaded implementations, the *parallel* flavor achieves a much higher record send rate with respect to *Concurrent Traditional* . The gap in production is more evident than the gap in previous consumption. The median record send rate of *parallel* is 41.5% higher than *Concurrent Traditional* .

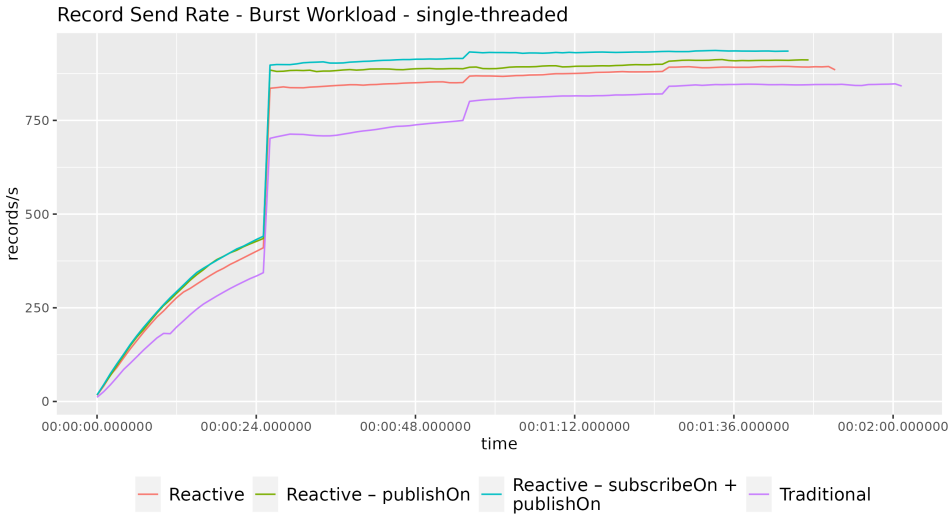

Figure 4.20. Record send rate with burst workload — single-threaded

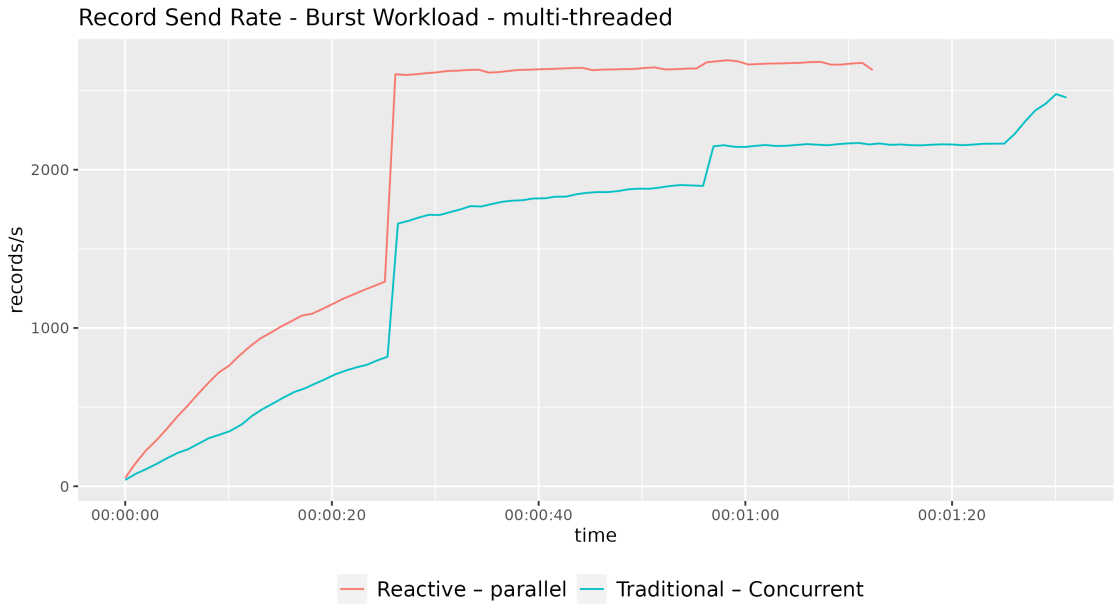

Figure 4.21. Record send rate with burst workload — multi-threaded

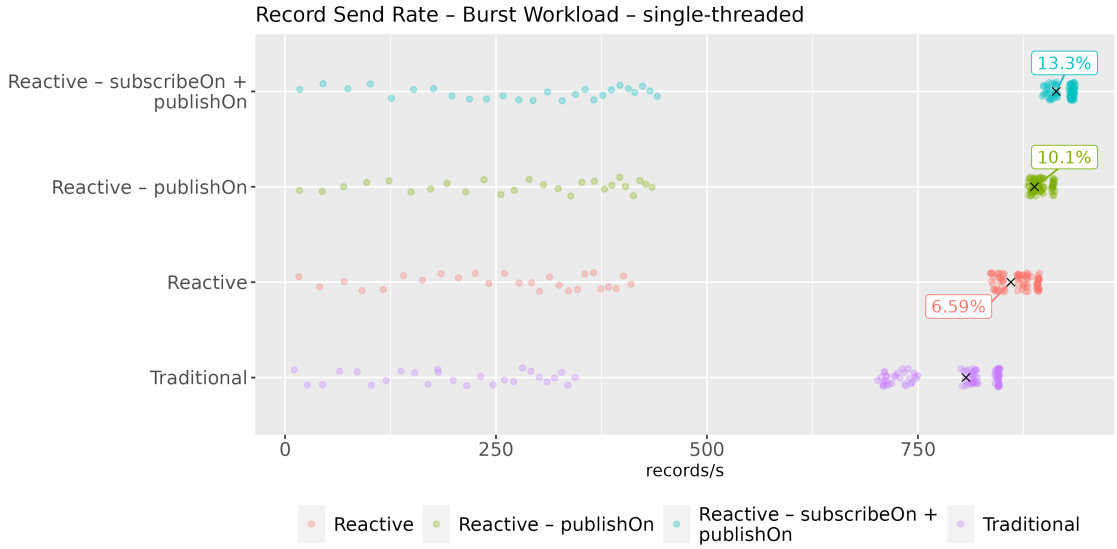

Figure 4.22. Distribution of record send rate with burst workload — single-threaded

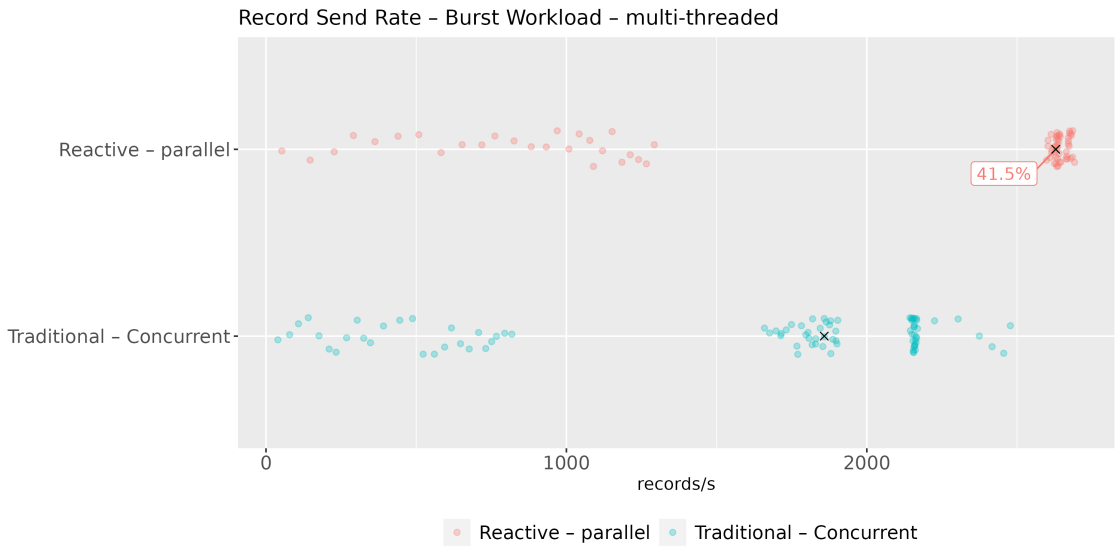

Figure 4.23. Distribution of record send rate with burst workload — multi-threaded

Table 4.7. Producer Performance – Burst Workload

version	Mean Record Send Rate	Median Record Send Rate	Max Record Send Rate
Traditional	670.29	806.95	847.55
Reactive	726.88	860.17	894.08
publishOn	745.31	888.16	912.42
subscribeOn + publishOn	762.84	913.94	936.63
Concurrent Traditional	1.57×10^3	1.86×10^3	2.48×10^3
parallel	1.99×10^3	2.63×10^3	2.69×10^3

4.2.4 JVM Metrics

As shown in figure 4.24, the *traditional* implementation consumes a higher percentage of CPU with respect to the single-threaded reactive flavors. The reactive implementations can attain a CPU usage median value 15–17% lower than *traditional* .
On average, the *parallel* implementation uses slightly more CPU resources with respect to *Concurrent Traditional* . However, the *Concurrent Traditional* flavor reaches higher peaks in CPU utilization.
From the memory usage perspective, both single-threaded and multi-threaded traditional implementations consume more memory resources with respect to all reactive flavors. Under burst workload, the *publishOn* and *subscribeOn + publishOn* implementations can save up to 16% memory on average with respect to *traditional* . In the multi-threaded case, the advantage is even greater — the *parallel* version uses, on average, 22.5% less memory than *Concurrent Traditional* .
The addition of the `publishOn()` or `subscribeOn()` operators in the reactive pipeline seems to introduce a slight overhead in CPU usage with respect to the base *reactive* implementation. At the same time, these operators bring off a more significant saving in memory usage.

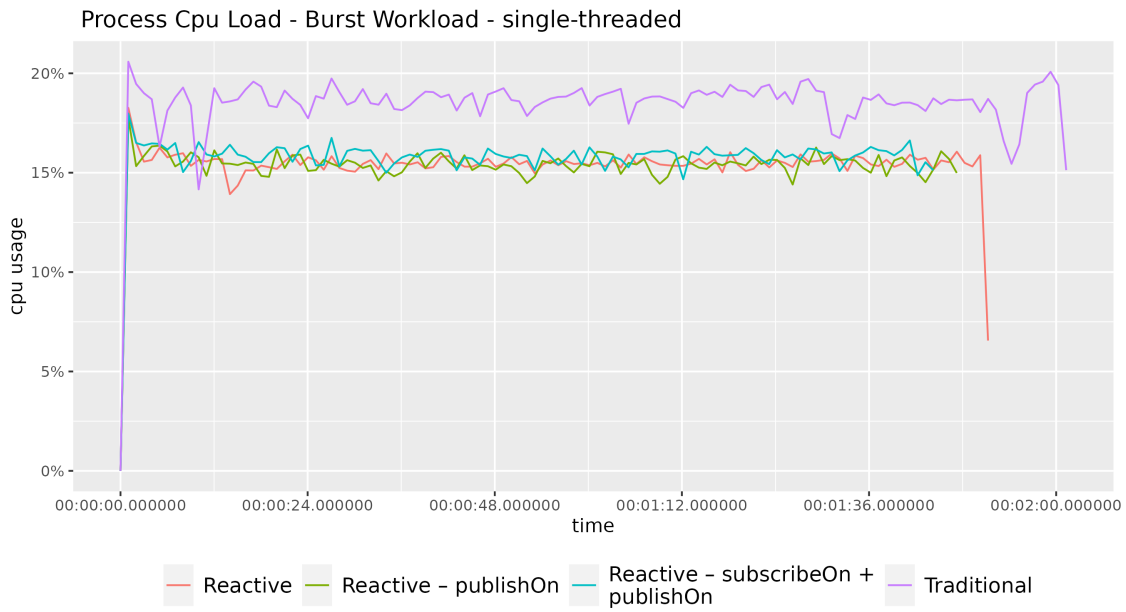

Figure 4.24. Process CPU load with burst workload — single-threaded

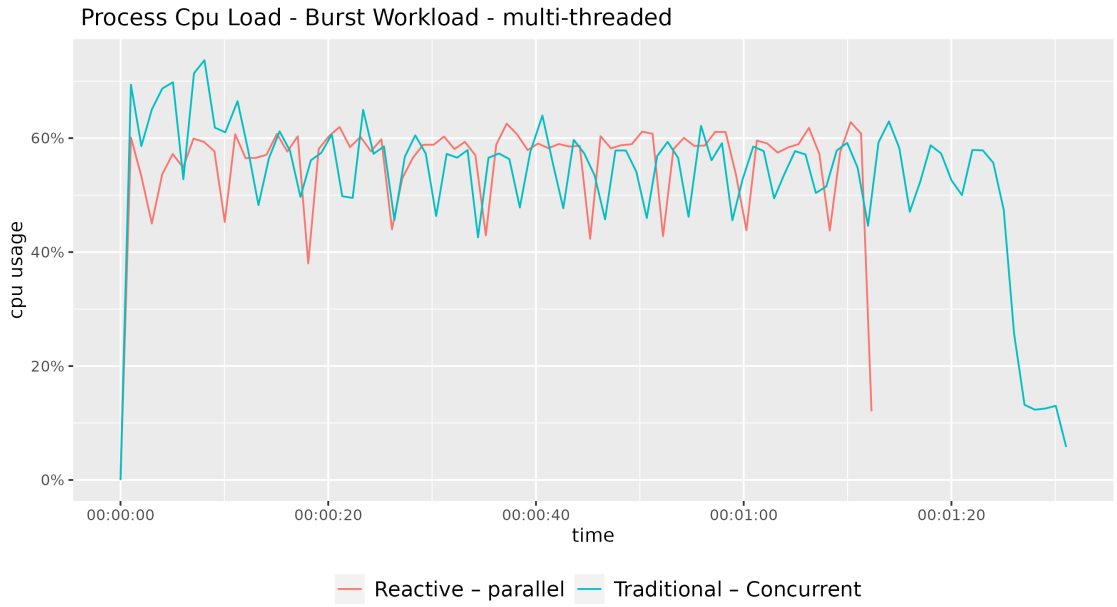

Figure 4.25. Process CPU load with burst workload — multi-threaded

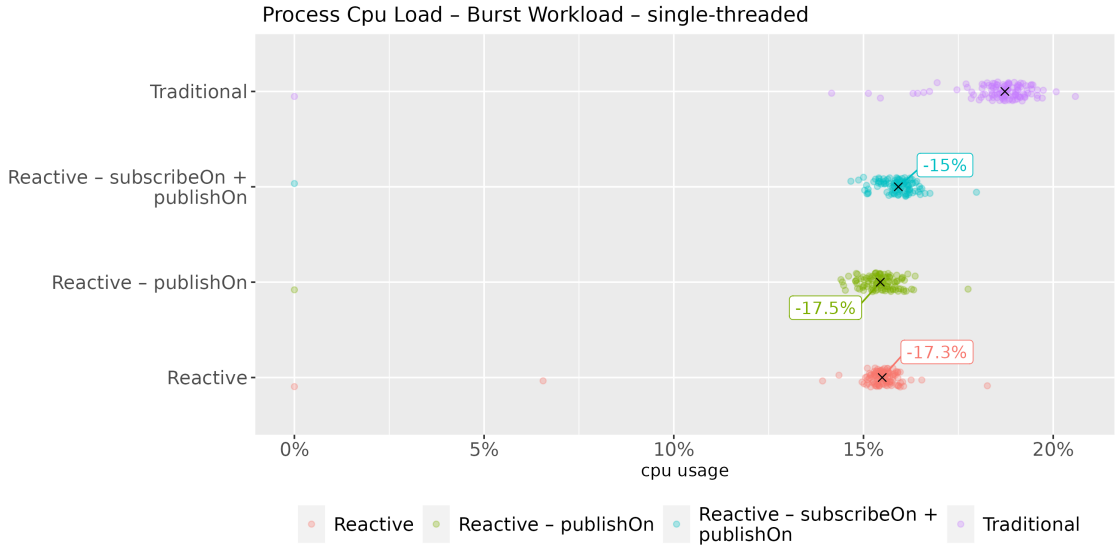

Figure 4.26. Distribution of process CPU load with burst workload — single-threaded

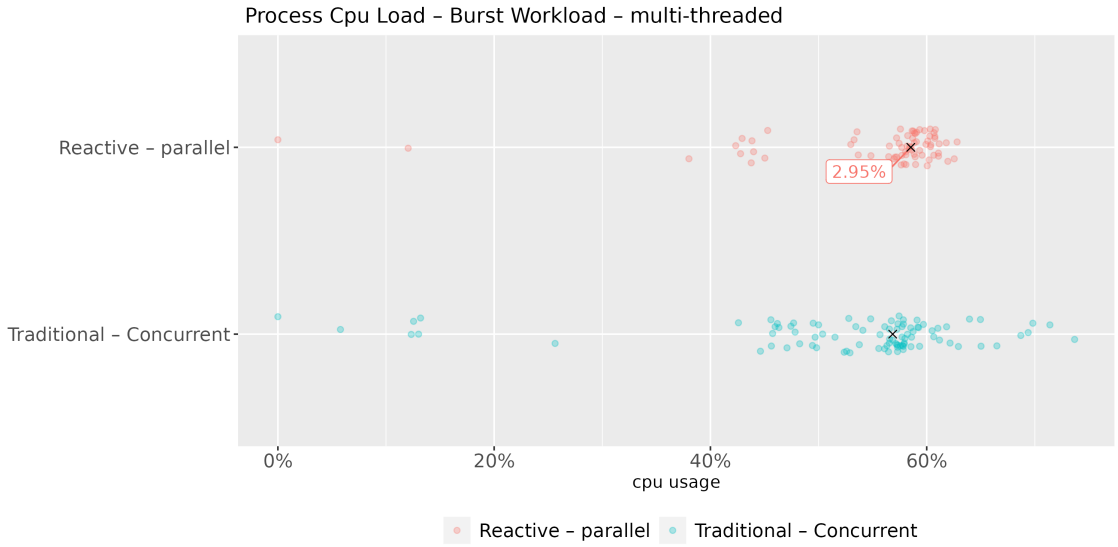

Figure 4.27. Distribution of process CPU load with burst workload — multi-threaded

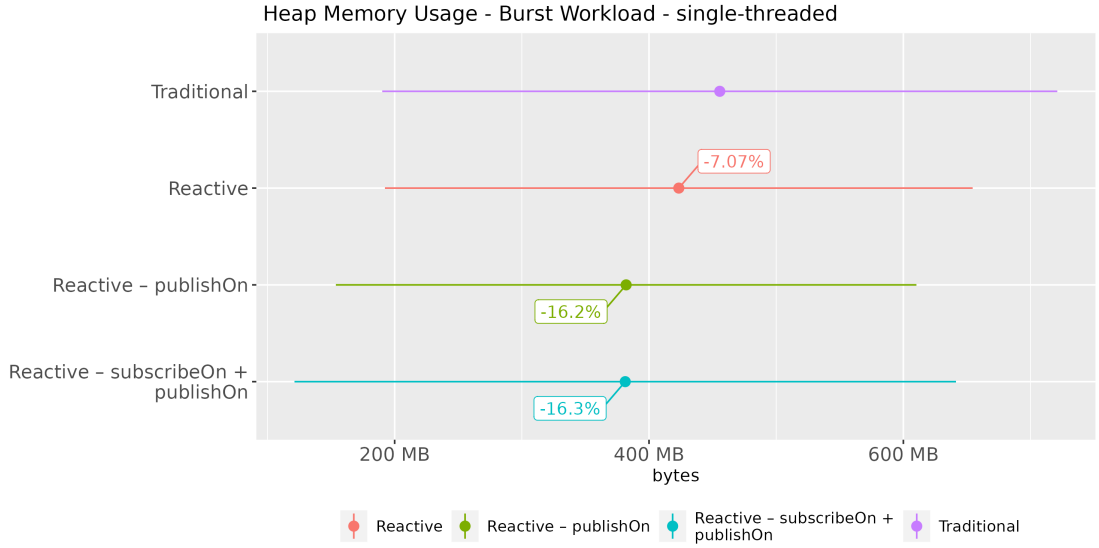

Figure 4.28. Heap memory usage with burst workload — single-threaded

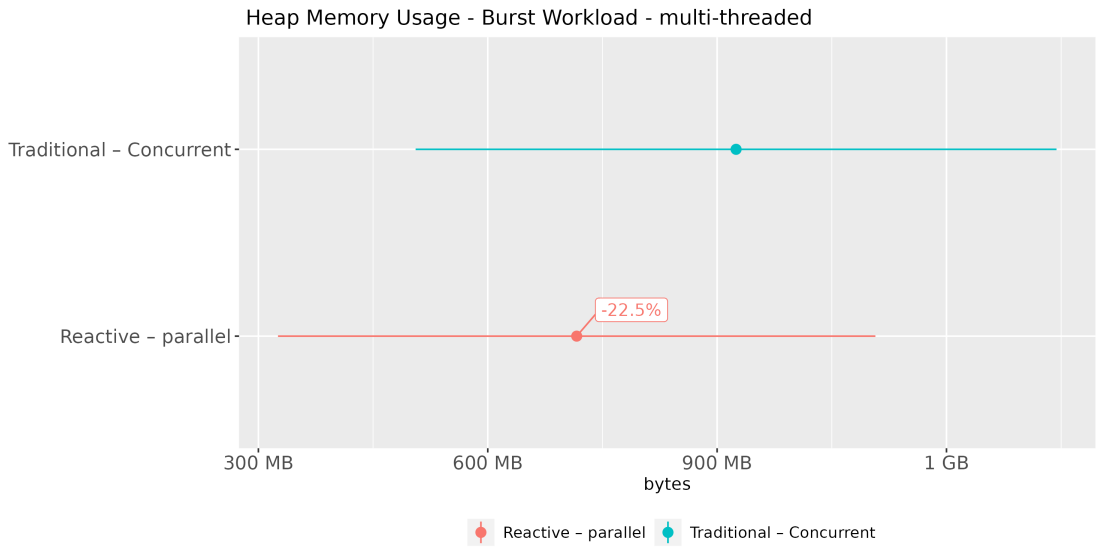

Figure 4.29. Heap memory usage with burst workload — multi-threaded

Table 4.8. Process Performance – Burst Workload

version	Mean Memory Usage	Mean CPU Load	Max Memory Usage	Max CPU Load
Reactive	423.4MB	15.31%	1GB	18.26%
publishOn	382MB	15.31%	968MB	17.76%
subscribeOn + publishOn	381.3MB	15.74%	970.7MB	17.98%
Traditional	455.6MB	18.45%	1.2GB	20.58%
Concurrent Traditional	924.8MB	52.83%	1.8GB	73.68%
parallel	716.3MB	55.35%	1.4GB	62.81%

Chapter 5

Conclusion and Future Research Directions

5.1 Limitations

We identify two main limitations of this thesis work. The first one is relative to the infrastructure adopted. The Apache Kafka message broker is deployed to the same machine on which the data-ingestion application is run. In a real scenario, usually, the message broker is deployed on the cloud, so the communication is limited by network latencies. This is not the case in this thesis.

The second one concerns the data-ingestion application itself. The traditional flavor of the data-ingestion application reproduces the real one. This work of reproduction has some limits. The queries of the external services in order to build a response are mocked. Moreover, the actual data-ingestion application is based on the bank's proprietary code built upon the Spring framework. In our case, the traditional flavor is constructed directly in Spring.

5.2 Future Research Directions

Future research could focus on the results obtained from a more realistic scenario in which a full-fledged Kafka cluster is deployed on the cloud and has multiple brokers. Moreover, it would be interesting to study the maintainability of the reactive approach with respect to the traditional one.

5.3 Conclusion

The first objective of this thesis is to evaluate and compare the performance of traditional implementations of Kafka clients with implementations that use a reactive approach.

Under constant rate workload, the single-threaded reactive implementations attain higher consumption and production rate and lower record lag with respect to the traditional implementation while presenting a lower footprint in terms of CPU and memory usage. Regarding multi-threaded implementations, the *Concurrent Traditional* version attains slightly higher consumption and production rate with respect to the reactive *parallel* . However, the multi-threaded reactive implementation can save a significant amount of CPU and memory resources than the traditional one.

Under burst at start-up workload, the advantage of using a multi-threaded implementation of the data-ingestion application is more evident. The same considerations done for single-threaded implementations in the case of constant rate workload are applicable in the case of burst workload. While something changes in multi-threaded consumption. In this case, the reactive *parallel* implementation achieves a much higher consumption and production rate at the cost of a slightly higher CPU load with respect to *Concurrent Traditional* . Memory usage is still lower in the multi-threaded reactive version.

The increase in performance of the reactive pipeline with respect to the traditional one is more significant during the production phase than the consumption one.

The second objective is to explore the thread scheduling model of the *reactive* version in order to improve performance further and discover trade-offs. Benchmarks show that we can obtain better performance by controlling the worker on which the subscription and part of the processing happen. In particular, the introduction of the `subscribeOn()` and `publishOn()` operators allow to attain an increase in consumer/producer throughput while decreasing heap memory consumption with respect to the plain *reactive* implementation. The speed-up achieved by changing the worker causes overheads in CPU usage.

Bibliography

[1] [Online]. Available: https://www.reply.com/iriscube-reply/it/

[2] P. Le Noac'h, A. Costan, and L. Bouge, "A performance evaluation of apache kafka in support of big data streaming applications," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, Dec. 2017.

[3] G. Hesse, C. Matthies, and M. Uflacker, "How fast can we insert? an empirical performance evaluation of apache kafka," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, Dec. 2020.

[4] G. van Dongen and D. Van Den Poel, "Evaluation of stream processing frameworks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1845–1858, Aug. 2020.

[5] G. Van Dongen and D. Van Den Poel, "Influencing factors in the scalability of distributed stream processing jobs," *IEEE Access*, vol. 9, pp. 109 413–109 431, 2021.

[6] S. Vyas, R. K. Tyagi, C. Jain, and S. Sahu, "Performance evaluation of apache kafka – a modern platform for real time data streaming," in *2022 2nd International Conference on Innovative Practices in Technology and Management (ICIPTM)*. IEEE, Feb. 2022.

[7] A. S. Tanenbaum and M. Van Steen, *Distributed Systems*. Maarten Van Steen, Jan. 2023.

[8] https://www.reactivemanifesto.org/.

[9] O. Dokuka and I. Lozynskyi, *Hands-on reactive programming in Spring 5: build cloud-ready, reactive systems with Spring 5 and Project Reactor*. Birmingham, UK: Packt Publishing, 2018.

[10] Reactive-Streams, "Reactive-streams/reactive-streams-jvm: Reactive streams specification for the jvm." [Online]. Available: https://github.com/reactive-streams/reactive-streams-jvm/

[11] "Core 3.5.3." [Online]. Available: https://projectreactor.io/docs/core/

release/api/
[12] [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html
[13] N. Garg, *Apache Kafka*. Birmingham, England: Packt Publishing, Oct. 2013.
[14] R. S. Gwen Shapira, Todd Palino and K. Petty, *Kafka: The Definitive Guide*. O'Reilly Media, Nov. 2021.
[15] G. Russell, A. Bilan, B. Kunjummen, J. Bryant, S. Chacko, and T. Fernandes, "Spring for apache kafka," https://docs.spring.io/spring-kafka/reference/html/.
[16] R. Sivaram, M. Pollack, and O. Dokuka, "Reactor kafka reference guide," https://projectreactor.io/docs/kafka/release/reference/.
[17] J. S. Perry, *Java Management extensions*. O'Reilly Media, 2002.
[18] [Online]. Available: https://kafka.apache.org/documentation/#monitoring
[19] [Online]. Available: https://kafka.apache.org/documentation/#remote_jmx
[20] [Online]. Available: https://kafka.apache.org/documentation/#consumer_fetch_monitoring
[21] [Online]. Available: https://kafka.apache.org/documentation/#producer_sender_monitoring
[22] [Online]. Available: https://docs.oracle.com/cd/E17802_01/j2se/j2se/1.5.0/jcp/beta1/apidiffs/java/lang/management/MemoryMBean.html
[23] [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/management/MemoryUsage.html
[24] [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/management/OperatingSystemMXBean.html