INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

A Comprehensive Survey of Asynchronous API Approaches in Concurrent I/O Scenarios

Diogo Paulo de Oliveira Rodrigues

Licenciado em Engenharia Informática e de Computadores

Relatório preliminar para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Março, 2022

Contents

1

Introduction

As state-of-the-art non-blocking IO tools and technologies continue to evolve, new asynchronous idioms emerge across different programming platforms. Examples of these include Dotnet Async Enumerables [1], JS asynchronous generators [4], Kotlin Flow [5], among others.

Simultaneously, some idioms provide pipelining API´s [2] that allow developers to build queries over sequences of data elements. This pipeline refers to a series of operations applied to a sequence of elements where each operation in the pipeline takes the output of the previous operation as input. A sequence pipeline with asynchronous data processing may be also denoted as reactive stream pipeline.

The research work that I describe in this dissertation aims to analyze what are the intrinsic overheads behind a reactive stream pipeline processing. While there are advantages regarding their conciseness, expressiveness, and readability, will these pipelines incur any overhead?

To answer this question we will analyze the behavior of different reactive stream pipelines technologies dealing non-blocking IO.

While non-blocking IO pipe lining enabling technologies, allows programmers to exploit concurrency without explicitly managing threads, it introduces new challenges and potential performance bottlenecks. The seamless integration of non-blocking IO with reactive stream pipelines presents a complex interplay of factors that may impact performance and resource utilization.

This dissertation aims, to retrieve performance metrics in several technologies and discuss how these technologies behave in different use cases.

1.1 Motivation

Reactive stream pipelines, it's a concept behind several powerful tools that allows the orchestration of a series of operations on a live feed of sequence of elements in continuous non-blocking IO operations. The ability to perform asynchronous data processing through these pipelines provides developers with a concise, expressive, and readable way to handle complex tasks.

However, with these benefits come potential challenges. The integration of non-blocking IO with reactive stream pipelines introduces a complex interplay of factors that may significantly impact performance and resource utilization. While the industry is rapidly adopting these idioms for their expressiveness and ease of use, there is a growing need to understand the underlying overhead and bottlenecks. Misunderstanding or overlooking these aspects can lead to inefficient code, wasted resources, and ultimately a poor user experience.

Moreover, the variety of technologies offering reactive stream pipeline processing adds another layer of complexity. Different approaches may behave differently in various use cases, leading to uncertainty when selecting the right technology for a specific project. Understanding how these technologies perform in different scenarios is vital for informed decision-making and effective utilization.

This work aims to provide a clear picture of how and when to use each technology and strategy, along with an understanding of their respective benefits and drawbacks in comparison to one another.

1.2 Goals

This research is guided by the following central objectives:

1. **Non-blocking IO**: Investigate state-of-the-arts idioms and technologies that enable processing in non-blocking IO use cases.

2. **Baseline Behavior Analysis**: Analyze the behavior of a baseline approach, for example, a piece of code that makes no use of any pipeline of operations,

the logic is made through direct implementation of algorithms using non-abstract programming. Understanding the trade-offs of such an approach, including verbosity, complexity, and maintainability, against performance benefits.

3. **Technology Comparison**: Evaluate and contrast various idioms and technologies that take advantage of non-blocking IO, focusing on how different paradigms (e.g., pull vs. push) concepts and how they perform under various conditions.

4. **Reactive Streams Technologies**: Investigate the practical implementation of reactive stream pipelines using different technologies.

We expect that a baseline approach without the use of any pipeline technology, perform better than any other alternative, despite its verbosity, complexity, and difficult maintainability.

On the other side, we would like to understand which approach that provides an asynchronous pipeline API that performs better, for example in the same language depending on the task.

Lastly, this work aims to have results and conclusions from different programming languages implementations of asynchronous pipelining Tecnhologies. This is crucial to understand, for example, if the same technology e.g. *Reactor* performance differs depending on the programming language; on the other side, its interisting to know too, if any programming enviroment performs significantly better than the others in this particular set of JAVA, Kotlin, C#, and JavaScript.

We aim to observe how performance behaves in these different scenarios to uncover the distinct advantages and disadvantages of each approach. This in-depth examination is intended to culminate in practical conclusions that can guide informed decision-making in future projects.

1.3 Outline

This dissertation is organized into distinct chapters that systematically guide the reader through the research, concepts, methodologies, and findings:

- **Chapter 1: Introduction** - This initial chapter provides an overview of the research context, the motivation behind the study, the primary goals, and the approach taken.

- **Chapter 2:State-of-the-Art** - Here, we explore the foundational concepts related to reactive stream pipelines and provide a comprehensive review of the existing technologies and strategies in JAVA (including Kotlin), C#, and Javascript. The chapter aims to establish the current landscape and set the stage for our research.
- **Chapter 3: Benchmarking of Non-blocking IO Processing Algorithms** - This chapter presents the results obtained from the implementations and provides a thorough analysis and discussion of the findings. It includes comparisons between various reactive technologies and other strategies, and how they perform against baseline approaches.
- **Chapter 4: Conclusion** - The final chapter will summarize the main insights derived from the study, highlight the contributions made, discuss the limitations, and propose directions for future research.

2

State of the Art

This section, aims to show the background and the current state of the art on the subject related with asynchronous pipelining operations involving asynchronous data flows. On the section **??**, will be made an overview on previously developed work made on this subject, then, on section **??**, will be made a characterization of the key concepts related to it. By last, on section **??**, are presented and explained several technologies representative of the state of the Art on how asynchronous operation pipelining are implemented in different programming realities, e.g. on Kotlin, JAVA and C#.

2.1 Background

From the end of 80's to the beginning of the 2000's, with the acceleration of Moore's Law in hardware and network bandwidth development; the creation of the web as we know today through the wide spread of use of the HTTP protocol, and the support from new operative systems to multithreading support; the necessity of high responsiveness servers started to grow. This increase in demand of new ways to handle data through parallelism involving asynchronous flow processing, caused the necessity of design new programming models.

Taking the wave initiated by the Gang of Four in [3], where 23 patterns were compiled to deal with object-oriented problems; a group of researchers wrote a paper [7], where they identify the characteristics that high available server must

have, and how the use of asynchronous IO pipe lining design patterns can help to achieve these characteristics, as described bellow:

- Concurrency - The server must process multiple client requests simultaneously.

- Efficiency - The software design must be built aiming the use the least hardware resources as possible.

- Simplicity - The code of the solution must be easy to understand, modular and avoid own built design patterns as possible.

- Adaptability - The system must be totally decoupled from client implementations, allowing it to be easily used by any client independently of the underlying technologic realities. To achieve this, may be used standards e.g. [8] or SOAP.

To achieve some of these properties, the authors propose the *Proactor Pattern* [7]. In their opinion, traditional concurrency models, until that date, failed to fully achieve the enumerated properties.

On the other side, before presenting the *Proactor Pattern*, are identified two major existing non-blocking models, namely: *multithreading* and *reactive event dispatching*.

On multithreading, the paper refers that one of the most direct solution of this approach, is the handling of multiple requests by creating a new thread every request. Each request will then be fully processed and the recently created thread is then be disposed after the work is done.

This solution has several serious issues. Firstly, creating a new thread per request is highly costly in terms of computational resources, because are involved context switches between user and kernel modes; secondly, must be taken in account synchronization to maintain data integrity. Then, the authors warn about the fact that the IO retrieved data is mainly memory-mappped, which rises the question: What happens when the data obtained through IO becomes greater than the system memory can hold? The system stalls until more memory becomes available!?

One possible solution is asynchronous pipelines that will be discussed ahead in this dissertation. On last, if the server receives a high demand of requests, the server easily blocks in the process of creating and disposing threads.

To avoid this issue, the authors, recommended the use of dynamic thread pools to process requests, where each request will be linked to a pre-existing thread, avoiding all the overhead of creating and disposing a thread per request; however, issues related with memory-mapping and overhead due to the switching of data between different threads maintains.

Another traditional concurrency model identified by the authors of the paper, is the *Reactive Synchronous Event Dispatching*. In this model, a *Dispatcher*, with a single thread in a loop, is constantly listening requests from clients and sending work requests to an entity named *Handler*. The *Handler*, will then process the IO work Synchronously and request a connection to the client in the *Dispatcher*. When the requested connection is ready to be used, the *Dispatcher* notifies the *Handler*. After the notification, the *Handler* asynchronously sends the data, that is being or has been obtained through IO, to the client.
Although the authors identifying that this approach is positive, because decouples the application logic from the dispatching mechanisms, besided with the low overhead due the use of a single thread, the authors identify several drawbacks with this approach. Firstly, since IO operation are synchronous, the code for this approach is complex because must be set in place mechanisms to avoid IO blocking through hand off mechanisms. Then, if a request processing blocks, the processing of another requests may be impacted.

To keep the positive points but mitigating the identified issues of previous approaches, is suggested the *Proactor Pattern*. This pattern is very similar to the *Reactive Synchronous Event Dispatching*, however, after the requests processed by a single threaded *Completion dispatcher*, the IO work is then dispatched asynchronously to the underlying OS IO subsystems, where multiple requests can be processed simultaneously. For the result to be retrieved, is previously registered a callback in the *Completion Dispatcher* and the OS has the responsibility to queue the finished result in a well known place.

Finally, the *Completion Dispatcher* has the responsibility to dequeue the result placed by the OS and call the correct previously registered callback. With this, this model creates a platform that provides: decoupling between application and processing mechanisms, offers concurrent work without the issues inherent with the use of threading mechanisms and since IO is managed by the OS subsystems,

is avoided code complexity in handling possible blocking and scheduling issues.

The *Proactor Pattern*, creates the ground for several models used by modern platforms that use a reduced number of threads to process client requests and parallel mechanisms to do the heavy work in the background. Some of these technologies, are, for example: *Javascript NODE.JS*, *Spring Webflux*, *vertx* and others.

From what was explained until now, is evident the tendency followed by software architects in terms of asynchronous processing from non-reactive to event driven approaches and trying to make the code simpler and easier to maintain. Initially the systems were non-reactive, where each request had to be processed in a specific thread and that thread blocked until something got ready to go further. The code was usually very complex and hard to maintain. Then, with the asynchronous systems based on events with the introduction of callback systems inspired in patterns like the *Reactor* or *Proactor* the software design started to become more event driven, allowing the servers to be more efficient in responsiveness, resources optimization and code easier to read and maintain by the average programmer.

However, are some limitations in these asynchronous models. For example, if the data to be processed is bigger than the memory available at a given moment or if the data to be calculated is from a source that produces data at a constant rate that must be processed in real time, these models work badly. The traditional models fail to comply these objectives because are mostly eager by design or not comply with the notion of a continuous source of information that requires to be processed in real time. Taken this in account, projects like project Reactor, Asynchronous Enumerable provided by Microsoft or papers like [6], try to deal with these issues, by providing API's that merge the concepts of Fluent API's, functional programming and code syntax that tries to resemble synchronous code, being the complexity inherent with asynchronous models implementations hidden from the programmer since the asynchronous events are treated like, for example, a `Stream` in JAVA would be treated.

2.2 Asynchronous flow key concepts and design alternatives

With the development of several approaches related to asynchronous IO processing, and with the growing necessity to build a name set of properties that simplify the description of asynchronous systems, a dictionary of properties, concepts and design alternatives started to grow by itself. In the following, are discussed several of the concepts related with asynchronous data flow, namely:

2.2.1 Synchronous versus Asynchronous

Before explaining more terms related with asynchronous data flow, it's important to clarify what is synchronous and asynchronous in programming.

Asynchronous in programming, is a call to a function or routine that returns immediately, not blocking the caller until the operation is finished. The operation processing, will be completely independent from the caller execution process and can even be done in another machine. This way, the caller is freed to do more work, even to start `N` more operations in parallel.

Meanwhile, a call to a synchronous function or routine, blocks the caller until the operation finishes. In this case, the caller has to wait for the completion of the synchronous operation before going forward, which limits the program efficiency if parallelism is applicable. To better visualize what was explained, we have the following examples:

```
HttpClient client = HttpClient.
    newHttpClient();

HttpRequest request = HttpRequest
    .newBuilder(URI.create("SOME MOVIE API
    ")))...

Task futureResponse = client
    .sendAsync(request, new
    JsonBodyHandler<>(DTO.class))
    .thenAccept(res -> {
        Console.WriteLine( res.title);
);

Console.WriteLine("prints something")

client.Wait();

//OUTPUT:
//prints something
//movie title

```

Listing 2.1: Asynchronous call example

```
HttpClient client = HttpClient.
    newHttpClient();

HttpRequest request = HttpRequest.
    newBuilder(
    URI.create("SOME URL"))
    .header("accept", "application/json")
    .build();

HttpResponse<Supplier<DTO>> res =
    client.send(request, new
    JsonBodyHandler<>(DTO.class))//
    synchronous

Console.WriteLine(res.title);

Console.WriteLine("prints something")

//OUTPUT:
//movie title
//prints something

```

Listing 2.2: Synchronous call example

As we can see, in the synchronous call example, the operation return only happens after the whole subsequent remote operation is finished, consequently, the caller operation is dependent from several variables to go forward e.g. : HTTP messaging latency, remote server operation speed or bandwidth issues. This causes that the processing of the next code statements to happen only after the synchronous IO call.

Meanwhile, in the asynchronous operation call, the return happens immediately after the call, however, the processing inherent with that operation will start just when the subsystem that handles the asynchronous function is ready to process that work. For example, when the OS is ready to process the received responses from a remote server that handled the operation. In this case, the statement right after the asynchronous call is processed before the asynchronous operation. Allowing the IO asynchronous operation to processed outside the program scope and avoiding any block of the main program.

2.2.2 Push vs Pull

Another concept important to understand how asynchronous data flow is handled in programming, is the *Pull* and *Push* processing patterns. In *Pull* pattern,

usually, exists a source of data and the program iterates over that source to operate over each item.

On the other hand, in the *Push* pattern, the items of the data source are "Pushed" to a routine that will operate over that item. To help to assimilate what was just explained, we have the following example:

```
Flowable<Long> flow = Flowable
    .interval(1, TimeUnit.SECONDS);

Iterator<Long> iterator =
    flow.blockingIterable().iterator();

while (iterator.hasNext())
    System.out.println(iterator.next());


```

Listing 2.3: Example of Pull data handling patterns

```
Flowable<Long> flow = Flowable
    .interval(1, TimeUnit.SECONDS);

flow.blockingSubscribe(System.out::println
    );


```

Listing 2.4: Example of Pull data handling patterns

As we can see, in the pull pattern, the items are "pulled" from a data source through an iteration mechanism.

In contrast with that, in the *Push* pattern, items are pushed to a consumer through a supplier.

2.2.3 Hot versus Cold

Another property that must be taken in account when handling with *Reactive Streams* or asynchronous data processing in general, is the nature of the data flow. There are two main adjectives to name a data flow, `Hot` or `Cold`.

A `Cold` data flow, is a flow of information that is produced just when the stream pipeline in subscribed by an observer. In this case, the producer only starts sending/producing data when someone is interested in the data from that source. For example, when program uses a IO mechanism to lazily retrieve a sequence of words from a database, the IO mechanism will only start sending information just when a consumer subscribes that data flow. Usually, the data is sent to the consumer in unicast.

On the other hand, in a `Hot` data flow, the data is produced independently of existing any observer to that information. This mechanism usually work in broadcast and the data is continuously produced and sent to possible observers. In this

case, when an observer subscribes to a publisher, exists the possibility of data items being already lost to that publisher while in the `Cold` flow, the consumer usually receives all items that were produced by the source. In the following examples, a number is produced each 100 milliseconds:

```
ConnectableFlowable<Long> hot = Flowable
    .intervalRange(0, 100, 0, 100,
    TimeUnit.MILLISECONDS)
    .publish();
flux.connect();
Thread.sleep(1000);
flux.blockingSubscribe(System.out::
    println

//output:
//2
//3
//4
//

```

Listing 2.5: Cold flow example

```
Flowable<Long> cold = Flowable.interval
    (100, TimeUnit.MILLISECONDS);
Thread.sleep(1000);
cold.blockingSubscribe(System.out::println
    );
//Output:
//1
//2
//3

```

Listing 2.6: Hot flow example

As we can see, in the `Hot` data flow example, the items are emitted from the moment the producer is created, independently of existing any subscriber or observer attached to that publisher. Notice that when a consumer is subscribed to the publisher, 1 seconds after the emition started, the numbers from 0 to 10 were not printed.

In the `Cold` example, the producer only emits data when a subscription is done, and because of that, all the produced numbers were printed, in contrast with what happen in `Hot` stream, where data loss are almost certain.

2.2.4 Cancelable

As already stated above, asynchronous operations may run outside the main program scope. This implies, that the main program loses visibility and control on what happens in the asynchronous operation contexts. Because of that, exists the need to put in place mechanisms of control that allow the main program to maintain control over an asynchronous operation to, for example, cancel the operation or put in place finishing logic that allow, for example: resource disposing, logging, decisions etc...

These mechanisms, are many times done through the concept of *cancelables*. Usually, a cancelable, is an entity that represents an operation that can canceled from an external entity, or, an entity that allows to set logic when an operation finishes by any reason.

In C#, a cancelable is an interface implemented by objects that represent asynchronous operations and provides the means to cancel asynchronous operations, on the fly. This is achieved through a mechanism named: `CancellationToken`, that is used to pass information through different execution threads.

In RxJava, a `Cancelable`, is a functional interface with the method `cancel()`. Then, the `Cancelable` can be associated to a data source representation, the `Observable`, by calling the method `Observable.setCancelable(Cancelable)`. When the `Observable` finishes or is canceled for any reason, the method `Cancelable.cance` will be called. This way, proper logic is put in place to handle an asynchronous operation cancelation.

As we saw, these two concepts of cancelable diverge. One, provides the means to cancel an operation on the fly and gives some control over the operation cancelation; the other, provides the means to control an operation cancelation independently of how it was cancelled.

2.2.5 Error Handling

In synchronous environments, usually, when something goes wrong, the way to handle an error in the majority of cases its by throwing an exception and propagate it until the proper code handles it, usually in a try/catch block. However, in cases when exists an asynchronous operation or when a continuous stream of data items are being received, that way of dealing with an error can imply several issues, e.g: exceptions not reaching main program, log losing or asynchronous flow blockage.

Since log losing or blocking a whole operation because of a badly handled error is unacceptable, the best way to deal with errors in asynchronous data flow it is to isolate the error. This way, the flow processing may continue in parallel while the error is properly handled.

The best way to handle this kind of errors, it is to have proper callbacks that are called when an error occurs on the stream item. This way, a function can handle the error properly, without the necessity to blocking any data stream processing, if avoidable and the proper logging and any additional measure to handle it can be put in place.

2.2.6 Intrinsic Keywords

As already stated, asynchronous code is tendentiously harder to understand because, in opposition with what happens in synchronous environments, the operations inherent with the sequence of programming statements operations may not happen chronologically ordered. Because of that, many times it is difficult read, debug and sustain asynchronous software.

For that reason, many languages started to add syntax techniques that allow the programmer to build asynchronous code that resembles the synchronous syntax. Under the hoods, the virtual machines that sustain these syntax mechanisms, handle the code bounded with that 'intrinsic words' and builds asynchronous routines that the programmer will not be aware of; being this a way to abstract the programmer from the complexity of handling and sustaining complex asynchronous code.

One example of *intrinsic words* mechanisms, is the `async...await` keywords implemented in *Microsoft's .NET C#* and in *javascript*.

In the next example we can see an example of these keywords being used:

```
1  static async Task Main(string[] args)
2  {
3      IAsyncEnumerable<int> enumerable = FetchItems(1000);
4      int i = 0;
5      await foreach (var item in enumerable)
6      {
7          if (i++ == 10){ break;}
8      }
9  }
10
11 static async IAsyncEnumerable<int> FetchItems(int delay)
12 {
13     int count = 0;
14     while(true)
15     {
16         await Task.Delay(delay);
17         yield return count++;
18     }
19 }
20
```

Listing 2.7: Intrisic words .NET C#

Where, for example, we can observe in the line 16, a call to an asynchronous operation, and, by adding the keyworkd `await`, the next statement although

being a call to an asynchronous operation, the code statement order looks like it is from synchronous set of instructions.

Additionally, the use of `async...await` in .NET, for example, simplifies error handling in asynchronous code. Instead of use a callback to handle an error, by using `async...await` the error can be handled by just using a simple try/catch block.

To better visualize the advantage of using intrinsic words in asynchronous code, on the next example, we can see a code comparison in ECM6 Javascript, with and without the use of intrinsic words in asynchronous code. In the next example, it is possible to observe a decrease of code complexity and increment of readability where is used the "syntax sugar" provided by the 'yield' return.

The example using promises is purposefully made with a "Pyramid of Doom" code to accentuate a difficulty of reading asynchronous if is made without any concern with readability.

```
1  function ourImportantFunction(callback)
       {
2      task1(function(val1) {
3          task2(val1, function(val2) {
4              task3(val2, callback);
5          });
6      });
7  }
8
```

Listing 2.8: Javascript Example with promises

```
1   function ourImportantFunction() {
2
3       var val1 = yield task1();
4
5       var val2 = yield task2(val1);
6
7       var val3 = yield task3(val2);
8
9       return val3;
10  });
11
12
```

Listing 2.9: Javascript syntax sugar

As we can see, with the use of `yield` keyword, the code that uses a result of several asynchronous operation, instead of being used in a "Matrioska Dool" type of code, with a code made with a chain of callback results; the simple use of a intrisic keyword like `yield` simplifies the code a lot. Making the code previously hard to read in a easier code to understand and maintain.

2.2.7 Back-pressure

When the *pull* method is used to retrieve items from a source, the producer retrieves only the items it can process in the given time.

However, when the *push* approach is used as data retrieval method from asynchronous flows, the producers have the initiative to push items to its consumers. This can originate situations, where the producer emits items faster than the producers can handle, which can create problems like: unwanted loss of data, lack of responsiveness from consumers, etc...

To resolve these issues, were created strategies and design patterns that are commonly referred as *Backpressure*. There are four main approaches which the majority of `Backpressure` strategies are designed from and can be resumed as:

1. **DROP:** Producer drops items after a retrieving buffer gets full.
2. **Buffer everything:** A buffer, keeps all unprocessed items that are received. Usually, this strategy is used when all received items are critical for the business development and memory managment has flexibility to handle the increase of storage needs.
3. **Error:** An error is thrown when the buffer threshold is reached, usually all items received after the threshold is reached are discarded.
4. **Lastest:** Only the last received item in the given moment is kept.
5. **Missing:** No back-pressure strategy it is in place, all items that ca not be processed on arrival, are discarded.

2.3 State of the Art

In this section, we will delve into various state-of-the-art frameworks designed for asynchronous IO pipeline processing across multiple programming languages.

Initially, we explore the options available in the Java landscape in Section **??**. Here, we begin with an explanation of the multi-language project *ReactiveX.io* and its Java implementation, *RxJava*. Subsequently, we will discuss the Reactor Project, followed by an examination of non-blocking processing in the JVM, specifically through Kotlin's native `Flow` implementation.

Following Java, we shift our focus to Microsoft's .NET C# in Section **??**. A brief mention of ReactiveX's c# implementation precedes a deep dive into how C# natively addresses this challenge, particularly through the implementation of `AsyncEnumerables`.

Next, we explore JavaScript's native solution to this problem, highlighting the use of asynchronous Iterables through the *intrinsic keywords* `FOR AWAIT...OF` and Promises. Additionally, we refer to ReactiveX's implementation for JavaScript, RXJS.

Finally, in Section **??**, we provide a comprehensive overview and draw conclusions about the various technologies discussed, comparing how each approach can be utilized for different problems and objectives. We also make theoretical predictions about the potential performance of each technology under several known circumstances.

2.3.1 Java

In the context of Java, we will explore several libraries and frameworks aimed at simplifying asynchronous data flow handling.

The first is *RxJava*, which is a Java implementation of the *ReactiveX.io* project. This library uses the `Observer pattern` to handle real-time asynchronous processing with and without back-pressure. It is important to note that the ReactiveX.io project is a multi-language project, and its Java implementation, RxJava, is among the most widely used.

Next, we delve into *Project Reactor*, an integral part of Spring WebFlux's non-blocking web stack. Although it uses a different approach from *ReactiveX*, it still provides the same benefits, like allowing developers to work with a composable API for declarative, event-driven programming.

Lastly, we will discuss the *Kotlin Flow* strategy, which is utilized in the Kotlin language but interoperable with Java. This will provide us with a perspective on how coroutine-based asynchronous data processing is implemented in the JVM environment, contrasting with the Observer pattern used by ReactiveX and Project Reactor.

2.3.1.1 RxJava Library

Before we delve into the specifics of RxJava, it's important to discuss ReactiveX (Reactive Extensions) — the project that gave birth to it. ReactiveX is a multi-language project focusing on combining the observer pattern, iterator pattern, and functional programming techniques to make the handling of asynchronous streams of data manageable and efficient. ReactiveX libraries exist for a variety of programming languages, including JavaScript, Java, C#, and others.

ReactiveX adopts a declarative approach to concurrency, abstracting away the complexities associated with low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O. Instead, it encourages developers to focus on the composition of asynchronous data streams.

An *observable* sequence in ReactiveX can emit three types of items: "next" notifications (carrying items to observers), "error" notifications (carrying error information), and "complete" notifications (signaling the end of the sequence). Observers subscribe to these observable sequences and react to whatever item they emit. The sequences are lazy; items are not pushed to observers until an observer subscribes.

Now, let's turn our attention to RxJava, the Java implementation of ReactiveX.

RxJava, an open-source project, encapsulates the *Observer pattern*, the *Iterator pattern*, and functional programming techniques to manage asynchronous data flow and control event sequences.

The data source/publisher of an asynchronous event stream in RxJava is represented by the `Observable` and `Flowable` classes. The key distinction is that `Flowable` supports back-pressure through various buffering strategies.

In RxJava, `Observable` and `Flowable` represent a stream of N events, while a single event is encapsulated by the `Single` class.

RxJava offers fluent APIs in `Observable` and `Flowable`, enabling operation chaining in pipelines — a feature also found in synchronous environments like

Java's *Stream* fluent API or .NET's Linq framework. Developers can perform stream processing operations like `filter`, `flatMap`, and `distinct` on these asynchronous event sequences, much as they would on synchronous streams in fluent APIs.

Observers/subscribers in RxJava are consumers subscribing to `Observable` through the `Observable.subscribe()` method. These consumers can be implementations of the functional interface `Consumer<T>` or the *Observer* interface. The latter provides enhanced control over stream processing through error handling capabilities, which `Consumer<T>` implementations lack.

The *Observer* can be regarded as an asynchronous counterpart to the Java util interface `Iterator`, as evidenced by the similarity in their interface methods.

1. `onSubscription()`: This method is called immediately after a subscription is made with an `Observable`.
2. `onNext(T item)`: This method is called when an item is emitted by the asynchronous data source.
3. `onError()`: Contrary to the Iterator, the RxJava *Observer* is equipped to support error handling. This callback is invoked when an error occurs.
4. `onComplete()`: This method is called when the data source closes or the subscription finishes.

Given the possible relationship between synchronous and asynchronous programming, Table **??** was developed to better visualize the correlation.

	Single Item	Multiple Items
Java	`T getData()`	`Iterable<T> getData()`
RxJava	`Single<T> getData()`	`Observable<T> getData()`

Table 2.1: Your caption

2.3.1.2 Reactor Library

The *Project Reactor* is another part of the Spring portfolio of projects. It is designed to be a fully non-blocking foundation for Java, compliant with the Reactive Streams specification. It offers efficient demand management (back-pressure) capabilities, making it an ideal choice for scenarios involving live streams of data.

The design of Project Reactor is also based on the *Publisher/Subscriber pattern*. However, instead of using `Observable`, `Flowable`, and `Single`, Project Reactor uses `Flux` and `Mono` to represent asynchronous data streams. `Flux` represents a stream of 0 to N items, while `Mono` represents a stream of 0 or 1 item.

Similar to RxJava, Project Reactor provides a variety of operators that can be used to transform, filter, combine, and manipulate data streams. This allows developers to construct intuitive instruction pipelines.

Just like in RxJava, observers/subscribers in Project Reactor are represented by consumers that are attached to `Flux` and `Mono` via the `subscribe()` method. These consumers can either be implementations of the `Consumer<T>` functional interface or the `Subscription` interface. Here's a brief comparison of how Reactor relates to the conventional synchronous counterparts in Java:

	Single Item	Multiple Items
Java	`T getData()`	`Iterable<T> getData()`
Reactor	`Mono<T> getData()`	`Flux<T> getData()`

By adhering to the Reactive Streams specification and offering a wide array of operators to handle data, Project Reactor serves as a powerful tool for developing reactive, non-blocking applications in Java.

2.3.1.3 Kotlin Flow

Kotlin, despite being a JVM-based language, differs significantly from Java when it comes to handling asynchronous data flows. Kotlin uses coroutines, a feature natively supported in the language, to simplify asynchronous programming. This feature is used in the implementation of `Flow<T>`, Kotlin's main interface for handling asynchronous data flows.

In comparison to `Observable` and `Flux`, which are based on the Observer pattern, Kotlin's `Flow<T>` is more aligned with the principles of the *Publisher/Subscriber pattern*. This is evident in how the `Flow<T>` interface is implemented. The data source implementation for a `Flow<T>` is done using a builder, and the initiation of the flow sequence is triggered by the `Flow.collect()` method.

Hot flows in Kotlin are represented by the `SharedFlow<out T> : Flow<T>` interface. Unlike `Flow<T>`, which initiates a flow every time `Flow.collect()` is called, `SharedFlow.collect()` emits an unpredictable set of items from an external stream of events initiated before the call to `SharedFlow.collect()`.

Kotlin's `Flow<T>` provides a fluent API of intermediate operators that allow data transformation through operation pipelines, similar to what we have already seen in RxJava and Project Reactor.

A `Flow<T>` data source implementation is done through a builder, like we can see in the next example:

```
fun simple(): Flow<Int> = flow { // flow builder
for (i in 1..3) {
    delay(100) // pretend we are doing something useful here
    emit(i) // emit next value
}
}

fun main() = runBlocking<Unit> {
launch {
    for (k in 1..3) {
        println("I'm not blocked $k")
        delay(100)
    }
}
simple().collect { value -> println(value) }
}

//output:
//I'm not blocked 1
//1
//I'm not blocked 2
//2
//I'm not blocked 3
//3


```

Listing 2.10: Flow builder

A consumer, to start listening a particular data flow haves to call the method `Flow.collect()`
. Since `Flow` provides support only to cold flows, calling `collect()` has the particularity of initiating flow the sequence. On the other side, Hot flows in Kotlin are represented by the interface `SharedFlow<out T> : Flow<T>`.

The main difference between the implementation of these two interfaces, is at the result of `collect()` call. While the `Flow.collect()` starts the flow every

time its called, resulting in the limited emission of the same set of items per call; the `SharedFlow.collect()` emits an unpredicted set of items from external stream of events initiated before the call to `SharedFlow.collect()`. On the other side, while the `Flow.collect()` call context is private to the caller, the `ShareFlow.collect()` is shareable by N subscribers, which makes this solution ideal for broadcast mechanisms shared by many users. On the next example, we can see several calls to the `Flow.collect()` that results in retrieving the same set of items; as explained, the call starts a cold flow every time its called:

```
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}

//Output:
//Calling simple function...
//Calling collect...
//Flow started
//1
//2
//3
//Calling collect again...
//Flow started
//1
//2
//3


```

Listing 2.11: Kotlin collect multicall example

As we can see, with the use of the keyword *emit*, it is achieved the same of what we saw in C# with the use of the keyword `yield return`. In this case, the same way the *yield return* returned an item that was part of an asynchronous enumeration of events through `IAsyncEnumerable`, the use of the keyword *emit* will lazily emit data, as it becomes available to be set as event of the Flow.

Likewise what happens in RxJava, `Flow<T>` provides a fluent API of intermediate operators that allow data transformation through the use of operation pipelines, where is received an upstream flow and the operators return a transformed downstream flow through the traditional push methods like: filter, map(), zip(),take() etc... On the next example, we can see an example of an asynchronous data pipeline operation from Kotlin `Flow<T>`, taking advantage of the Fluent API provided by its platform:

```
suspend fun performRequest(request: Int): String {
    delay(1000) // imitate long-running asynchronous work
    return "response $request"
}

fun main() = runBlocking<Unit> {
    (1..3).asFlow() // a flow of requests
        .map { request -> performRequest(request) }
        .collect { response -> println(response) }
}

//Output:
//response 1
//response 2
//response 3


```

Listing 2.12: Kotlin collect multicall example

2.3.2 .NET C sharp

2.3.2.1 RxNet

RxNet is the .NET implementation of the ReactiveX project, providing the same powerful programming paradigm from ReactiveX to the .NET ecosystem. This implementation allows developers in the .NET framework to effectively manage asynchronous data flow using the Observer pattern and functional programming techniques, as explained in the ReactiveX and RxJava sections.

In the context of C#, the concepts of Observables and Observers (or Subscribers) are represented by the `IObservable<T>` and `IObserver<T>` interfaces. These are analogous to the `IEnumerable<T>` and `IEnumerator<T>` interfaces in the synchronous realm, and closely resemble the implementations seen in RxJava.

Therefore, the understanding and application of RxNet would follow the same principles and design patterns as observed in RxJava. The key advantage of

RxNet is that it provides .NET developers with an abstracted and simplified approach to asynchronous programming, similar to what the ReactiveX project offers in other languages.

Importantly, RxNet is fully integrated with other .NET asynchronous programming constructs such as Tasks and the async/await pattern, offering a robust and comprehensive toolset for addressing various asynchronous and event-based programming scenarios in the .NET framework.

2.3.2.2 Asynchronous Enumerables

In the .NET framework, the concept of an enumerable is represented through the `IEnumerable<T>` interface, which defines a method `GetEnumerator()`. This method returns an `IEnumerator<T>`, enabling iteration over a collection. To extend this concept to the asynchronous world, .NET introduces the `IAsyncEnumerable<T>` interface.

Similar to its synchronous counterpart, `IAsyncEnumerable<T>` returns an `IAsyncEnumerat` but with an asynchronous `MoveNextAsync()` method. This minor but significant modification lets us deal with data sources where data availability is asynchronous, such as real-time feeds, network streams, etc.

Here's a simple example of how asynchronous enumerables can be used in C#:

```
1  static async Task Main(string[] args)
2  {
3  IAsyncEnumerable<int> enumerable = FetchItems(1000);
4  int i = 0;
5  await foreach (int item in enumerable)
6  {
7      if (i++ == 10){ break;}
8      Console.WriteLine(item);
9  }
10 }
11
12 static async IAsyncEnumerable<int> FetchItems(int delay)
13 {
14 int count = 0;
15 while(true)
16 {
17     await Task.Delay(delay);
18     yield return count++;
19 }
20 }
21 //
22 //1
23 //1 sec delay
24 //2
```

```
25 //1 sec delay
26 //3
27 //....
28
```

Listing 2.13: Asynchronous Enumerable, C# example

This example demonstrates how asynchronous data sources can be worked with in a similar way as synchronous collections, thanks to the use of `IAsyncEnumerable<T>` and the `await foreach` construct.

2.3.3 Javascript

As a functional and dynamically-typed language, JavaScript provides a distinctive approach to managing asynchronous data flow. Enabled by its asynchronous runtime environment, Node.js, JavaScript employs specific intrinsic keywords and libraries to facilitate asynchronous operations. JavaScript's native constructs, namely the `async...await` and `for await...of` keywords, significantly ease the handling of asynchronous tasks. Beyond these built-in facilities, libraries like the Reactive Extensions for JavaScript (RxJS) enhance these capabilities further, enabling more sophisticated operations on asynchronous data streams. This section explores both JavaScript's intrinsic keywords and the RxJS library, emphasizing their respective roles in handling asynchronous flows in JavaScript.

2.3.3.1 Javascript Intrinsic Words: async...await and for await...of

The JavaScript runtime environment, Node.js, adopts a functional and weakly-typed approach to asynchronous flow processing. The mechanism to process asynchronous streams is somewhat similar to what we've seen in C#, but instead uses intrinsic keywords `async...await` and `for await...of`.

The keyword pair `async...await` provides a syntax that closely resembles synchronous code while making asynchronous calls. The `async` keyword marks a function as asynchronous and enables the use of the `await` keyword within it. The `await` keyword is then used before calling an asynchronous function, indicating that the function should pause and wait for the Promise to resolve or reject.

On the other hand, the keywords `for await...of` provide support for iterating over asynchronous enumerables, as demonstrated in the following example:

```
async function* streamAsyncIterablestream {
const reader = stream.getReader;
try {
while true {
const { done, value } = await reader.read;
if done {
return;
}
yield value;
}
} finally {
reader.releaseLock;
}
}

async function getResponseSizeurl {
const response = await fetchurl;
let responseSize = 0;

    const iterable = streamAsyncIterableresponse.body;

    for await const chunk of iterable {
        responseSize += chunk.length;
    }
    return responseSize;
}

```

Listing 2.14: Mozilla's Javascript Asynchronous Enumerables example

This example demonstrates how JavaScript's `async function*` construct can be used to define asynchronous enumerables. These can then be conveniently iterated over using the `for await...of` construct, just like you would with regular collections.

2.3.3.2 RxJS

RxJS represents the JavaScript adaptation of the ReactiveX project, analogous to RxNet in .NET and RxJava in Java. This library furnishes JavaScript developers with the same robust mechanisms and abstractions for handling asynchronous data streams.

While RxJS operates under similar principles as its Java and .NET counterparts, it introduces unique approaches that cater specifically to JavaScript's dynamic and functional nature. As such, the core philosophy remains consistent across these libraries: simplifying the management of asynchronous data streams, with specific implementations nuanced to suit the distinct characteristics of their respective languages.

2.3.3.3 IxJS - Interactive Extensions for JavaScript

IxJS, also known as Interactive Extensions for JavaScript, is an integral part of the ReactiveX project and aims primary to enable the manipulation of data sequences, similarly to RxJS.

Similarly with another technologies already studied in this work, IxJS focuses on providing powerful abstractions for managing both synchronous and asynchronous sequences of data, while also aiming to be approachable and understandable. It has a robust suite of operators that you can use to write expressive, declarative code. Developers can also create custom operators very easily, extending the core functionality of IxJS to suit specific needs.

2.3.4 Technologies comparison

As we saw, each technology have a set of properties that help to characterize the solution. To help the characterization of each technology documented, we have a relation between the characteristics saw in the chapter **??**.

	Rx(JAVA/.NET)	**FLUX**	**FLOW**	**C# async enums**	**IxJS**
Pull		x	x	x	x
Push	x	x			
Cancelable	x	x	x	x	x
Error Handling	x	x	x	x	x
Backpressure	x	x	x		
Intrinsic words			x	x	x

3

Benchmarking of Non-blocking IO Processing Algorithms

In this chapter, we delve into the benchmark results of two key algorithms applied using various strategies and implementations. These algorithms can be understood as abstract representations of real-world tasks that involve processing large amounts of data asynchronously. The two main algorithms under consideration are:

The main tasks can be broadly divided into two operations:

1. Finding the largest word in a set of files
2. Grouping of words based on their sizes, known as the "group words" operation

The "group word" operation identifies words within a specified size range in a dataset. The algorithm then returns a collection of these words, along with a count of their occurrences, and is particularly designed to output the most recurring words within the specified range. This operation presents an intriguing computational challenge as it requires efficient data retrieval, processing, and frequency analysis.

Conversely, finding the largest word in a set of files, although seemingly simple, becomes a non-trivial task when considering vast amounts of data.

The data used in these tests is a collection of text files from Project Gutenberg, a large digital library of thousands of free eBooks. Project Gutenberg offers a wide variety of books in different languages, and for this experiment, we used a random sample of hundreds of books, providing a diverse and challenging dataset for our implementations.

To give a better idea of the operations, we present the pseudocode for each operation:

For the "group words" operation:

```
FUNCTION GroupWords(folder, minLength, maxLength)
    Create an empty map 'wordMap'
    FOR each file in 'folder' DO
        Skip the first 14 lines of the file (These are typically metadata
        FOR each remaining line in 'file' DO
            IF the line contains "*** END OF" THEN
                Break (This is the end of the actual content in Gutenberg
            END IF
            FOR each word in 'line' DO
                IF length of 'word' is between 'minLength' and 'maxLength
                    Increment the count of 'word' in 'wordMap'
                END IF
            END FOR
        END FOR
    END FOR
    RETURN 'wordMap'
END FUNCTION
```

For finding the largest word:

```
FUNCTION FindLargestWord(folder)
    Set 'largestWord' as an empty string
    FOR each file in 'folder' DO
        Skip the first 14 lines of the file (These are typically metadata
        FOR each remaining line in 'file' DO
            IF the line contains "*** END OF" THEN
                Break (This is the end of the actual content in Gutenberg
```

```
            END IF
            FOR each word in 'line' DO
                IF length of 'word' is greater than length of 'larges
                    Set 'largestWord' as 'word'
                END IF
            END FOR
        END FOR
    END FOR
    RETURN 'largestWord'
END FUNCTION
```

During the implementations, a conscious effort was made to keep the operation pipelines as similar as possible across the different technologies for each algorithm. This endeavor aimed to create a fair and representative evaluation of the behaviors of each technology. By maintaining consistency in pipeline operations, we can more accurately attribute performance differences to the underlying technology, rather than variations in the implemented code. This approach brings us closer to a true comparison of how each technology handles the challenges of asynchronous I/O data retrieval and processing.

In certain instances, particularly with Java, there was a need to incorporate external libraries for non-blocking asynchronous file retrieval. This requirement highlights the varying degrees of native support for these operations across programming environments. Some environments have native functions, while others rely heavily on third-party solutions. This aspect of the project mirrors the challenges often faced in real-world environments, where the necessity to adapt and find suitable solutions is a common part of the development process.

The implementations differ in the specific programming languages, libraries, and technologies they use. This allows us to evaluate the relative strengths and weaknesses of each approach and provides valuable insights into how these factors can affect performance.

After presenting and discussing each implementation and its results, we will be able to directly compare them. This will enable us to draw meaningful conclusions about the performance of the strategies and technologies when applied to the same task under the same conditions. Specifically, we are interested in how the performance of a given strategy or technology can vary across different programming frameworks when tasked with finding the largest word and executing the group word operation.

3.1 .NET Benchmarking

In this subsection, we focus on the benchmark using different strategies in .NET programming environment.

3.1.1 Find biggest word algorithm results

For the find biggest word algorithm, used i .NET implementations are the following strategies:

- **Baseline**: This strategy serves as the basic approach for finding the biggest word and acts as a baseline for comparison.
- **Async Baseline** : This strategy uses a single asynchronous task to find the biggest word.
- **Parallel** : An approach that makes the use of parallelization.
- **Linq using blocking IO**: This strategy uses Language Integrated Query (LINQ) in a synchronous manner.
- **Asynchronous Enumerables on Non-blocking IO**: This approach uses asynchronous programming with enumerable collections.
- **RxNet blocking IO**: The Reactive Extensions (Rx) library is used to handle data sequences asynchronously and event-based.
- **RxNet** : This strategy uses the Reactive Extensions (Rx) library with asynchronous file reading operations.

In the following graphic, we have the results in seconds for each strategie:

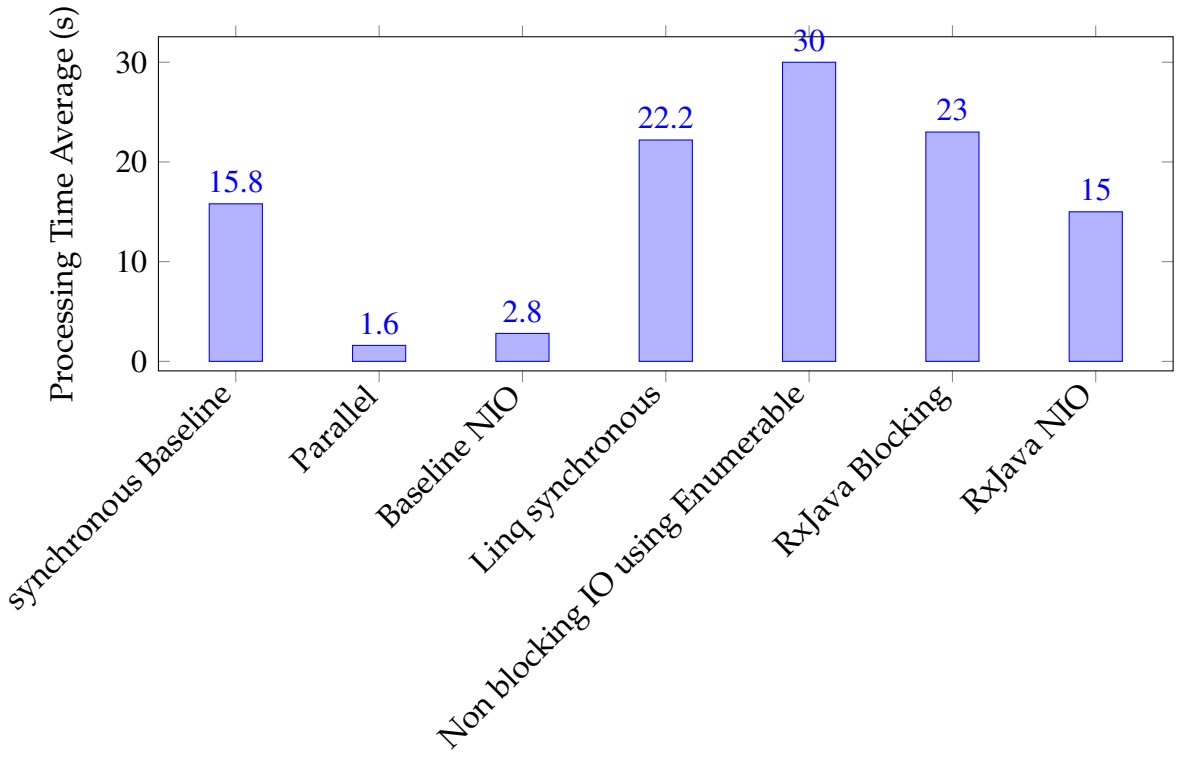

Figure 3.1: Processing times for different strategies for "Find the biggest word".

As we can see from these results, it is evident that the algorithm for "Finding the largest word" in a large file dataset performs worse with blocking IO solutions compared to non-blocking approaches. Furthermore, when considering the various solutions, it becomes apparent that the performance deteriorates as the API complexity increases. An exception to this trend is that, despite leveraging non-blocking IO, AsyncEnumerable exhibits the poorest overall performance.

3.1.2 Group Word Results

In this subsection, we concentrate on the task of grouping words from a file using different strategies. The strategies that we evaluate here include:

- **Baseline**: This strategy serves as the basic approach for word grouping and acts as a baseline for comparison.
- **Linq**: Like in the find word algorithm, this strategy uses Language Integrated Query (LINQ) in a synchronous manner.
- **AsyncEnumerable**: This strategy uses .NET async enumerables to process the asynchronous data .
- **RxNet**: Here, the Reactive Extensions (Rx) library is used to handle data sequences asynchronously and event-based.

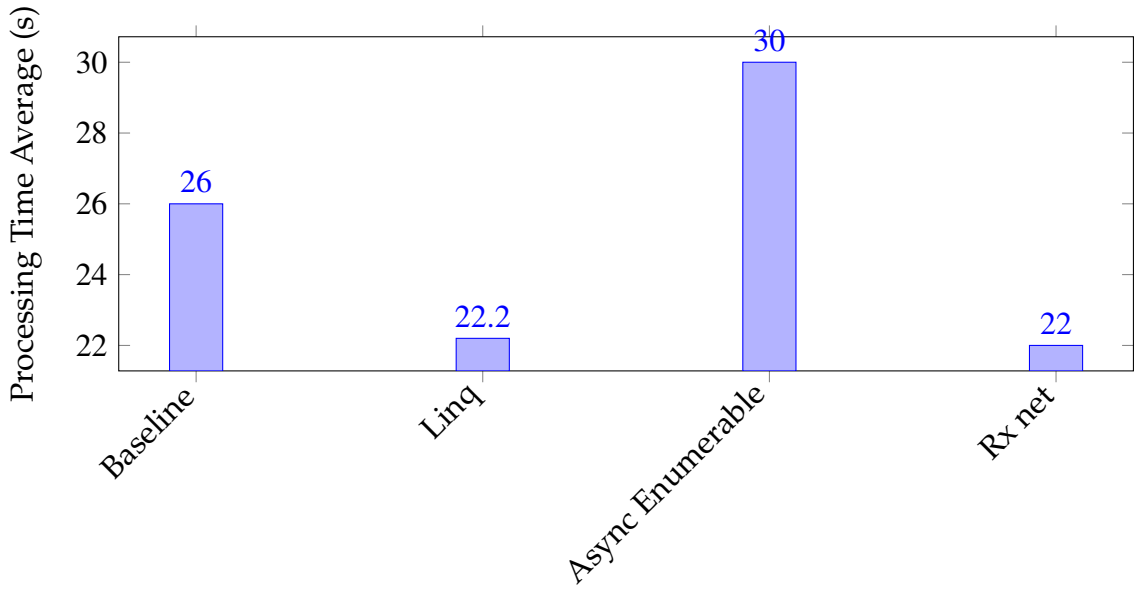

Figure 3.2: Processing times for different strategies for "Count Words".

In the case of memory-intensive algorithms like "Group words," which heavily rely on dictionaries, for example, pipeline libraries outperform baseline approaches. However, whether one uses blocking or non-blocking IO operations becomes irrelevant in such scenarios, as memory-intensive operations overshadow any potential gains from using non-blocking IO operations.

3.2 Java/Kotlin Benchmarking

In this section, we explore and assess diverse strategies applied in Java and Kotlin to process files, and we scrutinize their performances solving the "Find Word" an "Group Word algorithms"

3.2.1 Biggest Word Results

The strategies used in JAVA are:

- **Baseline**: This strategy illustrates a basic non-blocking I/O operation, serving as a comparison baseline.

- **Flux**: These strategies leverage the Reactor Flux model from Java's Project Reactor library. The former follows a standard non-concurrent processing model, while the latter introduces parallelization for improved performance.

- **RXJava**: This strategy employ the RXJava library. They replace the Reactor Flux with Observables, with the distinction being made between non-concurrent and concurrent processing.

- **Streams and parallelization**: Implementation of three strategies that use Java's Streams API and explore handling of blocking operations under three different conditions: standard usage, raw multithreading using threadpools and using parallel method in the streams API.

In the following graphic, we have the results in seconds for each strategy:

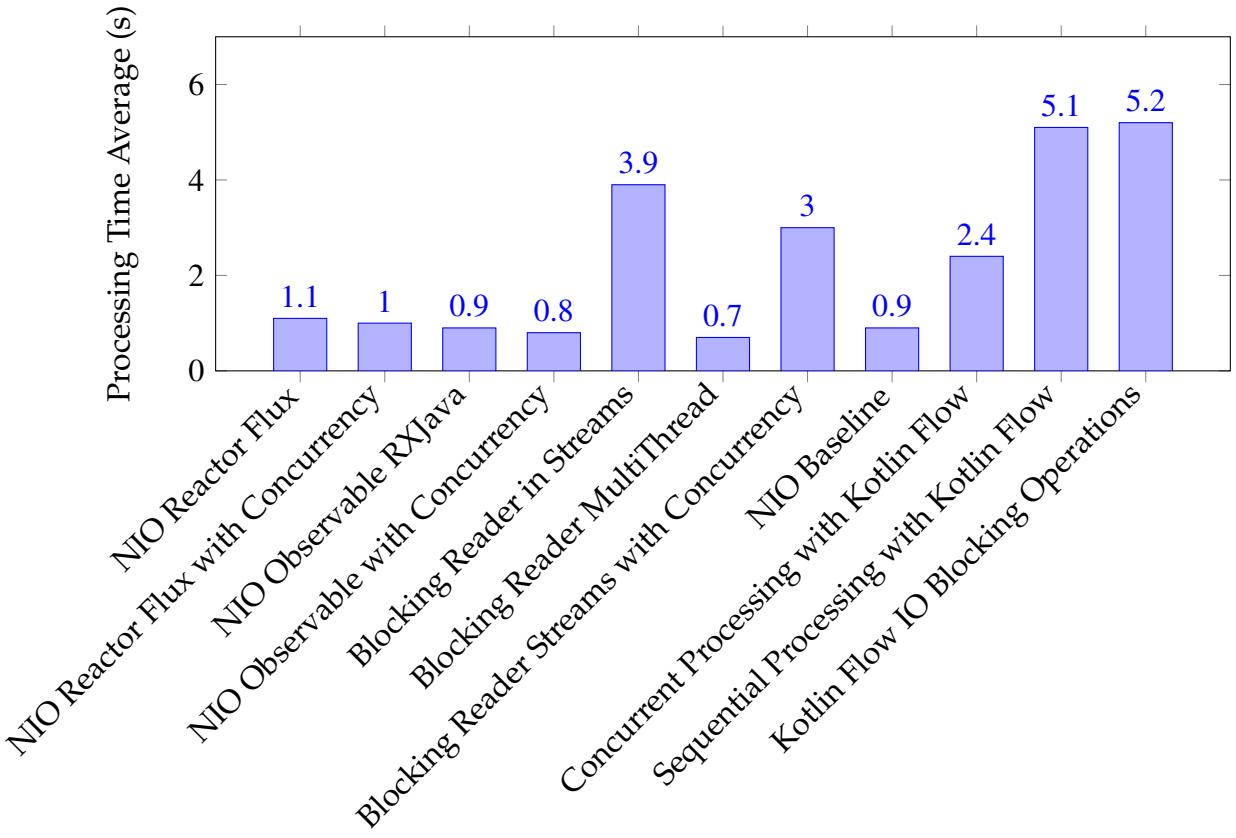

Figure 3.3: Processing times for different Java/Kotlin strategies for "Biggest Word".

3.2.2 Group Word Results

The strategies discussed here include:

- **Baseline**: This strategy serves as a baseline for comparison. It illustrates a basic non-blocking I/O operation without the use of any high-level constructs like Reactor Flux or Observable.

- **RXJava**: This strategy employs the RXJava library, popular for building asynchronous and event-driven applications.

- **Flux**: This strategy leverages the Reactor Flux model available in Java's Project Reactor library, providing an efficient approach to handling asynchronous data sequences.

- **Parallelization**: This strategy uses Java's Streams API and explores handling of blocking operations with the help of threadpools.

- **Streams**: Similar to the previous strategy, this one uses Java's Streams API, but handles blocking operations within streams.

- **Flow (Kotlin)**: This strategy utilizes Kotlin's Coroutines and Flow API, which are particularly well-suited for handling multiple values that are emitted sequentially.

In the following table and graphic, we have the results in seconds for each strategy:

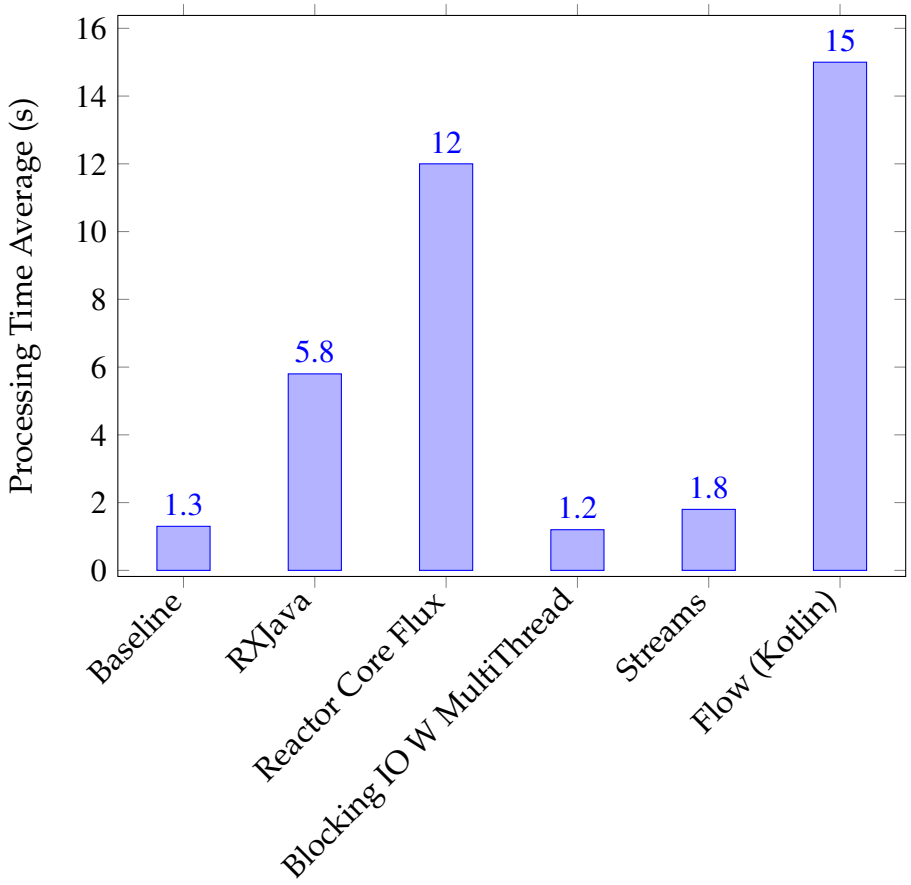

Figure 3.4: Processing times for different Java/Kotlin strategies for "Group Words".

...

3.3 JavaScript Benchmarking

3.3.1 Finding the Biggest Word

Here, we investigate the following three strategies:

- **Baseline for await..of without pipeline**: This strategy serves as the basic JavaScript approach for finding the biggest word, acting as a baseline for comparison.
- **Baseline for await..of with pipeline**: This strategy uses JavaScript streams, which provide a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient manner.
- **RxJS**: This strategy leverages the Reactive Extensions for JavaScript (RxJS) library, which offers a set of methods for dealing with asynchronous data sequences in an effective way.

In the following graphic, we have the results in seconds for each strategy:

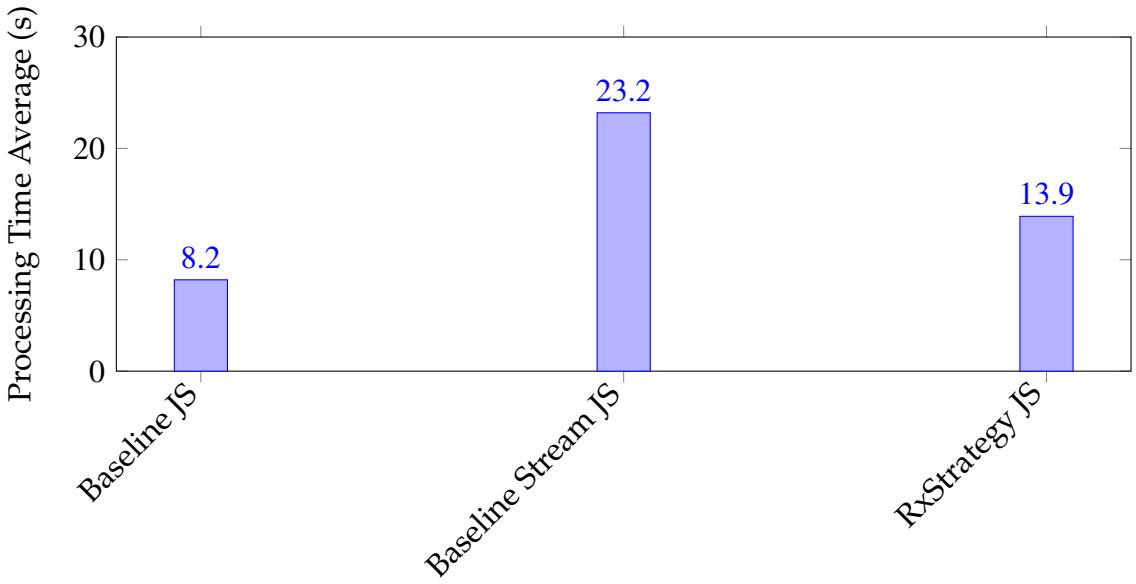

Figure 3.5: Processing times for different JavaScript strategies for "Biggest Word" Graphic

As we can see in this javascript implementation, the performance for find biggest word algorithm of the baseline behaves as expected, having the best performance among its alternatives. On the other side, the algorithm implemented using RXJS library behaves slightly better than the alternative that makes use of JS streams API.

3.3.2 Grouping Words

In this subsection, we evaluate the following strategies for grouping words:

- **Baseline JS**: This strategy serves as the basic JavaScript approach for grouping words, acting as a baseline for comparison.
- **Baseline Stream JS**: This strategy uses JavaScript streams, which provide a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient manner.
- **RxStrategy JS**: This strategy leverages the Reactive Extensions for JavaScript (RxJS) library, which offers a set of methods for dealing with asynchronous data sequences in an effective way.

In the following graphic, we present the results in seconds for each strategy:

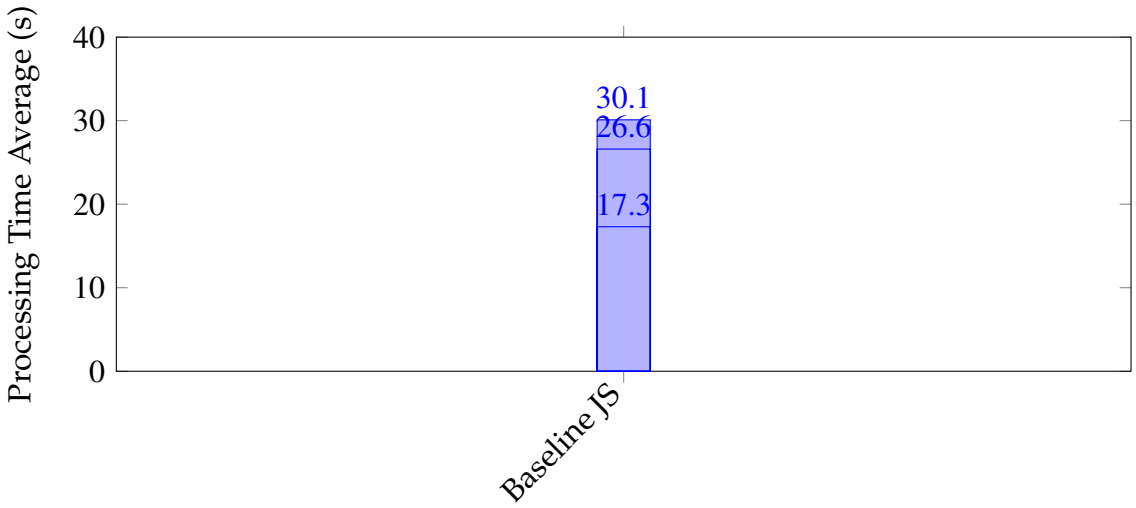

Figure 3.6: Processing times for different JavaScript strategies for "Grouping Words" Graphic

In this case, the performance of the algorithms change radically from what we saw in the "Find the biggest word" algorithm. In this case, the baseline haves the worst performance among the three strategies, while the RxStrategy haves the best. One explanation for these results is that the pipeline instructions of the pipeline instructions are optimized while the baseline is not.