INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

From asynchronous IO to reactive stream pipelines

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

List of Figures

List of Listings

1

Context and Progress status

This report has the objective of informing the reader, about the current status of the development of the dissertation named *From asynchronous IO to reactive stream pipelines*, that is being written by the student *Diogo Paulo de Oliveira Rodrigues*.

Firstly, to understand the current state of development, it is important to make clear what will be the objective and structure of the document that is being written.

This dissertation aims, in the first place, to make an overview on what tools were historically available to deal with non-blocking IO operations and asynchronous programming in general, and then, indentifying what are the state-of-the-art solutions available today to deal with this problem that are usually supported through asynchronous data pipeline architectures. After presenting state-of-the-art concepts and technologies, some of these technologies will be then compared in terms of performance through the development of practical benchmark test cases, written in several programming languages widely used today.

Taking in account what was explaining above, the dissertation will aim to have the following main structure:

1. Introduction and motivation
2. Concepts and State-of-the-art
3. Test case development and Benchmark results

4. Conclusions

Since this report is preliminary, this structure can be slightly changed in the course of the development.

To fulfill the objectives in a desirable timeline, were created several tasks scheduled to be done in a pre-determined time interval. This task can be viewed in the next Gant diagram, where it is possible to see the completion status of each task as they were in 15 of March:

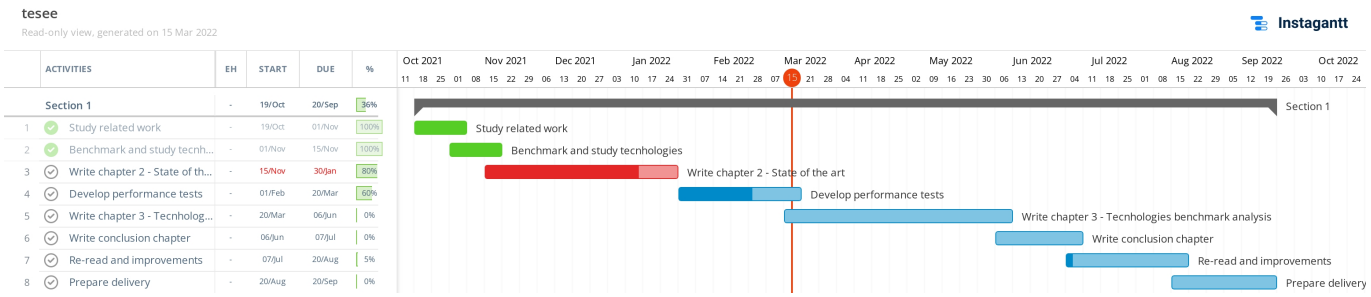

Figure 1.1: Planning Gant diagram

As we can see in the Gant diagram, 80% of the chapter 2 is completed, however, are remaining some adjustments in some examples and to be more clear in the explanation of others, these adjustments and a complete review will be completed until the final of March.

On the other side, the development of the testing solutions to benchmark the different asynchronous pipeline technologies are under course. The finish date to this task is expected to be 20 March, where is expected to begin the writing of the chapter 3 of the thesis.

On the next chapter of this report, is written the chapter 2 of the thesis "*State of the Art*" as it was on the day of 15 of March 2022.

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 data operations involving asynchronous data flows. On the section 2.1, will be made an overview on previously developed work made on this subject, then, on section 2.2, will be made a characterization of the key concepts related to it. By last, on section 2.3, are presented and explained several technologies representative of the state of the Art on how asynchronous data flows is handled 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 [1], where 23 patterns were compiled to deal with object-oriented problems; a group of researchers wrote a paper [3], 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. [5] or SOAP.

To achieve some of these properties, the authors propose the *Proactor Pattern* [3]. 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 [2], 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 be explaining the different frameworks that are the *State of the Art* in asynchronous IO pipeline processing in several programming languages.

First, in the section 2.3.1, will be presented the multi-language `reactivex.io` project, and how the `Observer Pattern` is used in this approach to implement real time asynchronous processing with and without *back-pressure*.

Secondly, in the section 2.3.2, will be presented the *.NET Async Enumerables* and how this approach diverges from approaches made in *reactivex.io*, *Javascript*, and in Kotlin with *Kotlin Flow*. In the sections 2.3.3 and 2.3.4, will be respectively presented the JavaScript and the Kotlin Flow strategies for asynchronous data processing.

Finally, in the section 2.3.5, will be made an overview and taken conclusions about the different technologies presented, and how each approach can be used for different problems and objectives. Then, will be made a theoretical prediction on how each technology behaves in several know circumstances.

2.3.1 ReactiveX.io Project

After the historical context explained in the chapter 2.1, many groups of developers started to implement several solutions to handle asynchronous data flows. One of the most ambitious results of this proactivity is the *reactiveX* project.

This project gathers several ideas and known design patterns to implement an asynchronous data flow mechanism. The most positive points about this project implementation is that the mechanism is implemented in several programming languages using an open source approach. Above that, the solution design is done in a way, that all the complexity inherent with asynchronous programming is hidden from the programmer, making the code very intuitive and iterative an easily understandable for programmer that are not used to handle with asynchronous enviroments. In this dissertation, we will analyze the implementation of reactive X made for Java, which is named: RXJava

2.3.1.1 RxJava architecture

To implement an asynchronous mechanism that can handle `Hot` and `Cold` data flows with and without back-pressure, the project adopted the *Observer pattern* allied with the concepts described in the *Publishes/subscriber pattern*. The designers justify this choice by saying that this model allows treating streams of asynchronous events with the same simplicity made in the synchronous counterparts e.g. in `Collection` and `Iterator`. The designer complement the justification by saying that, using this approach, the code complexity inherent with low-level threading, non-blocking IO, thread-safety etc... are completely abstracted from the programmer.

In the implementation, the data source/publisher of an asynchronous stream of events is represented by the classes `Observable` and `Flowable`. In this case, the only difference between `Observable` and `Flowable`, is that the class `Flowable` supports back-pressure through different bufferization strategies.

On a side note, it is important to refer that in RxJava, the same way `Observable` and `Flowable` represent a stream of N events, a single event is represented by a class named `Single`.

One of the great advantages of using this framework, is that the `Observable` and `Flowable`, is that these types offer fluent API's that allows chaining operation fluently in operation pipelines, like its provided on synchronous environments like java `Stream` fluent API or with .NET Linq framework. In these fluent API's, stream processing operations like: `filter`, `flatMap` or `distinct` are available, allowing to treat these asynchronous sequence of events like data item are treated synchronous streams in fluent API´s like . The fluent API's, can then be used to create a pipeline of operations like we see in synchronous pieces of codes, but in this case, the pipelines deal with asynchronous data, improving a lot the code readability in contrast with we would see if callbacks and another non suitable ways of deal with asynchronous operations would used.

On the other hand, the observers/subscribers are represented by consumers that are attached to `Observable` through the method `Observable.subscribe()`. The consumers can be either implementations of the functional interface `Consumer<T>` or implementations of the interface `Observer`.

The `Observer` interface is built with the intent to offer an extra control over the stream processing through error handling, which a `Consumer<T>` implementation lacks. The `Observer` can be viewed as the asynchronous representation of the java util interface `Iterator`, the resemblance can be seen by

seeing the interface `Observer` methods:

1. `onSubscription()`: Method called right after the subscription is made with an `Observable`
2. `onNext(T item)`: Method called when an item is emitted by the asynchronous data source.
3. `onError()`: In contrast with Iterator, the RxJava `Observer` is equipped to support error handling. This callback will be called when an error occurs.
4. `onComplete()`: Method called when the data source closes or if the subscription finishes.

Since its possible to make a relation between synchronous and its asynchronous logic counterparts in many languages, on next, we will take as example the relation between `Java` library and how these objects can be directly related to their asynchronous counterparts in `RXJava`. On the next table, it is possible to see how each object type can be mapped:

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

2.3.2 Asynchronous Enumerables

To better understand what an asynchronous enumerable is, firstly, we must define what Enumerables are.

An enumerable, is the logical representation of a collection that can be iterated. In synchronous programming, this is usually made by creating a Collection class, that implements a mechanism that enables iteration over its items.

On the other side, in the asynchronous world, many times exists situations where a stream of data items are constantly arriving from a remote source, the commonly named *asynchronous Flows* or *asynchronous streams*.

The software designers saw, that did not exist many logical differences between a collection and real time data sources; the only difference was that a collection items could be retrieved on demand from memory, while the data flows are iterated over time when items become available for consumption.

Taking all this in account, has born the idea of creating the notion of the asynchronous enumerable, where the asynchronous enumerable is the representation of a constant source of readable items. With the use of this concept, were implemented several mechanisms, e.g. the Asynchronous Enumerables in C#, that simplify the iteration over asynchronous data sources. Making the code similar to synchronous statements, easier to maintain, and, once again, abstracting the programmer from many complex details inherent with asynchronous processing.

2.3.2.1 .NET Asynchronous Enumerables

`IEnumerable<T>` in C#, is an interface that implements a method `GetEnumerator()` that returns a `IEnumerator<T>`. In its turn, `IEnumerator<T>`, is an interface that allows to iterate over items of a collection by offering the method `T getCurrent()`, that allows to retrieve the current item and a method `bool MoveNext()`, that allows to iterate to the next item of the sequence and check if the iteration has finished. `IAsyncEnumerable<T>`, keeps the same logic of `IEnumerable<T>`, but instead of returning a synchronous `IEnumerator<T>` with a synchronous `MoveNext()` method; it returns a `IAsyncEnumerator<T>` where its method `bool MoveNextAsync()` is asynchronous.

On the other side, the keyword `yield`, is a syntax sugar introduced in **C# 2.0** that allows to avoid the implementation of `IEnumerable<T>` and `IEnumerator<T>` interfaces by the programmer, allowing iteration and the lazy return of items. All the work of implementing iteration interfaces is handled by `.NET` virtual machine in background. On top of that, the use of `yield` is compatible with asynchronous iteration involving `IAsyncEnumerable<T>`, which simplifies the iteration of asynchronous flows in C#, as we can see in the next example:

```
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 2.10: Asynchronous Enumerable, C# example

As we can see in the example, although the operations and the emission of items being asynchronous, thanks to the IAsynEnumerable and the use of the intrinsic keywords, the code seems similar to a synchronous code. With the *await foreach* statment, the item emission is constant and non-blocking although existing blockage with the call to `Task.Delay` statment.

2.3.2.2 Javascript asynchronous enumerables

Node.js tries to provide a functional and weakly typed approach to asynchronous flow processing, opposed to what we saw in the C# solution.

The mechanism to provide the means to iterate over asynchronous streams, is relativity similar to what we saw in C#, with the use of the intrinsic keywords `async...await` and `for await...of`.

With the keyword pair `async...await`, its provided the syntax "sugar" that enables to make asynchronous calls and allowing the syntax to resemble synchronous code. The `async` mark is used in an asynchronous function and enables the use of the keyword `await`. The `await` keyword is then used in an asynchronous call to another asynchronous function

On the other side, the keywords `for await...of`, provides support for iteration over asynchronous enumerables, as we can see in the next example:

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

async function getResponseSize(url) {
    const response = await fetch(url);
    let responseSize = 0;

    const iterable = streamAsyncIterable(response.body);

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

```

Listing 2.11: Mozilla's Javascript Asynchronous Enumerables example

In this example, the function `streamAsyncIterable`, provides a data stream that emits data items as they become available.

On the other side, on the function `getResponseSize()`, the keyword `await` its followed by an asynchronous HTTP request and the mechanism `for wait...of` it is used to iterate over an asynchronous iterable stream. As we can observe, the code syntax is very similar to a synchronous iteration.

With the implementation of this mechanism, JavaScript simplifies in a very subtle way, how asynchronous flows are handled, allowing the code to be easily readable and maintainable and abstracting the programming from, for example: callbacks, tasks or completion tokens.

This solution is usually used in front-end environments, for example, to load several items from remote sources while an HTML web page loads.

2.3.3 Kotlin Flow

Kotlin asynchronous flow implementation is a [4] compliant solution, inspired in RxReactor and another projects, made with the intent to be simple as possible. Similarly to RxJava, Kotlin strongly typifies a stream of N events, which, in this case, is done through the implementation of the interface `Flow<T>`.
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 2.12: 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 2.13: 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 2.14: Kotlin collect multicall example

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 2.2.

	RxJava	Kolin Flow	Javascript	C#
Pull			x	x
Push	x	x		
Cancelable	x	x	x	x
Error Handling	x	x	x	x
Backpressure	x	x		
Intrisic words		x	x	x

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, ISBN: 0-201-63361-2.

[2] Oleg Kiselyov, Simon Peyton Jones, and Amr Sabry, "Lazy v. yield: Incremental, linear pretty-printing", vol. 7705, Dec. 2012, ISBN: 978-3-642-35181-5. DOI: 10.1007/978-3-642-35182-2_14.

[3] Douglas C. Schmidt Tim Harrison Irfan Pyarali and Thomas D.Jordan, *Proactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events*. 4th conference of Pattern Languages of Programming, 1997.

[4] Reactive Streams initiative. "Reactive stream". (), [Online]. Available: http://www.reactive-streams.org/.

[5] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. University of California, 2000.