Performance Comparison of Reactive and Object-Oriented Design In Event Processing Applications

Hessam Emami
hemami@smu.edu

Abstract

High volume data stream processing became a universal challenge for software engineers. The majority of emerging technologies consume and process data streams to generate knowledge, and software architects and developers propose new design paradigms to enhance these data-driven applications' performance. Reactive programming is not a new concept, but we did not have the desire or tools to consider it in our past design patterns. By having more powerful processing units and faster network connections, we have the opportunity to apply it in today's system design. This paper compares the reactive and Object-Oriented Programming performance differences in event-based applications.
Keywords— Event Processing, Reactive programming, Object Oriented Design, Performance.

1. Introduction

User expectations on program performance have been increasing compared to what engineers had to support a decade ago. Studies show that about 47 percent of users expect website loads in less than 2 seconds [1]. Whether the application uses REST API, events, SOAP APIs, or any other protocols or technologies, we cannot afford to lose our users due to high latency or slow response time. Modern competitive software must fulfill the following requirements [2]:

- Modular/dynamic
- Scalable
- Resilient
- Fault-tolerant
- Responsive

Reactive programming has recently been considered a popular paradigm for developing high volume data processing applications that realize these requirements. The reactive programming paradigm is based on the synchronous dataflow programming paradigm [Lee and Messerschmitt 1987] but with relaxed real-time constraints [3]. In procedural or Object-Oriented programming, iteration is the main block to process event streams. The application consumer thread is responsible for checking the upcoming events and read them in the loop. Each iteration in the loop can consume one or a collection of events; however, a dedicated thread will be assigned to each event, and until the process of it is not complete, it will occupy the resources. One solution to increase the performance is to run a multi-thread application to handle multiple messages simultaneously. This method is a conventional approach that has been used in Object-Oriented programming but imposes multi-threaded programming difficulties, which quickly can get out of control in large applications.

This article focuses on application responsiveness through message throughput, message latency, and CPU usage differences between reactive and Object-Oriented design in event-based applications. We also discuss the obstacles of using reactive programming in a large-scale application and make recommendations to couple these two methods in large applications.

2. Background and related work

As we discussed, In the previous section, software engineers' design should have extraordinary attention to five attributes in high user volume applications. Designing and implementing such applications needs a higher level of abstraction. In Reactive programming, instead of having the call stack, the application reacts to events. This paradigm does not keep the states and can respond to a stream of events asynchronously. Glenn Wadden introduced this concept in 1986, and since then, many user interface developers have been employing this idea. For example, Front-End web developers bind event handlers to HTML elements and the UI responses to events generated by users' actions accordingly. Due to the tremendous growth of the internet and new concepts such as big data and the increase in computational power, asynchronous data processing has

been studying for Back-End systems. Reactive programming is one of the paradigms that can benefit developers to deliver a high-performance application.

When using reactive programming, data streams are going to be the spine of your application [4]. Events, messages, calls, and even failures are going to be conveyed by a data stream [4]. Therefore, everything, including error messages, is considered data—also, the non-blocking feature helps to optimize the usage of resources.

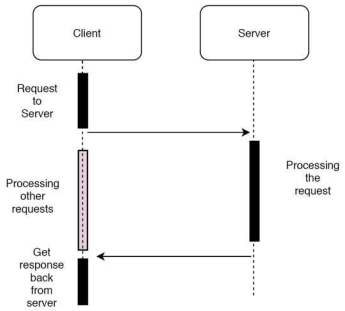

Figure 1 - Reactive design

Shape [1] shows that in this design, the client application can resolve a promise, call a callback function or notify an observer when it receives the server's response while using its resources to process other client's requests. In Object-Oriented programming with iterations, the client must wait until the server sends the response to the client applications.

We should also keep in mind that a reactive system is different from reactive programming. Reactive System builds on top of reactive programming. A reactive system is an architectural style that allows multiple individual applications to coalesce as a single unit, reacting to its surroundings, while remaining aware of each other—this could manifest as being able to scale up/down, load balancing, and even taking some of these steps proactively [6]. Scalability and resilience are specific attributes of a reactive system that reactive programming does not necessarily bring to the system. Shape [2] demonstrates the reactive system manifesto and the relationship between these attributes:

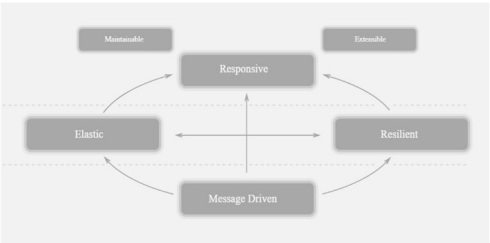

Figure 2 - Reactive Manifesto – reactivemanifesto.org

As shown in shape [2], the reactive system uses the message term instead of the event. Message-driven systems most often correspond to distributed processes communicating over a network, maybe as cooperating microservices [7]. Event-driven applications are local [7]. This difference can impact some attributes such as scalability, but it is not a standard definition, so in this paper, we will use the event and message terms interchangeably.

3. Reactive in event processing

Apache Kafka is a community distributed event streaming platform capable of handling trillions of events a day [8]. Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit log [8]. Apache Kafka supports poll-based Java APIs as well as Reactive APIs through the Scala library that wraps Java APIs. One of the common programming styles to use Kafka poll-based APIs is to create an infinite loop in a thread either for consumer or producer and process the new events or generate new ones in the loop. Apache Kafka's broker is responsible for managing topics, partitions, and portion's offsets in each partition. Adding extra broker can increase the system's reliability as each broker can add redundancy to the system and increase the throughput; however, to run more parallel processes concurrently, the number of partitions in each broker can be increased. Each client application has a unique id, and each partition can only be consumed by on thread, but multiple threads from one client Id can be connected to one topic. On the other hand, producer's threads can produce events in each topic independently. Different acknowledgment strategies in Kafka impact the performance or reliability of the applications. However, most of these methods require a wait/resume logic in producer and consumer threads, which reduces the application's performance as some resources need to wait until they receive an acknowledgment from the consumer or the prouder. The Reactive paradigm aligned with functional programming concepts can dedicate the resources to another task and respond to these triggers when they are ready.

This paper evaluates consumer and producer Kafka applications' performance for the Object-Oriented poll-based and functional reactive designs in consumer and producer applications. For this purpose, we consider four scenarios:

- Generate events without remote dependencies.
- Generate events with remote dependencies.
- Process events without remote dependencies.
- Process events with remote dependencies.

In the first and third scenarios, we evaluate the raw performance of traditional REST blocking and Reactive APIs. As there is no latency caused by external resources, this scenario can help us to understand the impact of using these APIs in local applications or use cases that are not heavily dependent on I/O. In the second and fourth scenarios, messages cannot be processed or produced without interruptions, which is a more realistic scenario in most distributed applications. Each scenario has one consumer and one producer applications, and these applications are developed in Java. Kafka environment is set up on the Confluent platform, and performance data will be gathered through Confluent tools.

In our multi-threaded consumer applications, we can see the following loop that can be run in an independent thread:

Pseudo 1 Traditional API logic

```
WHILE True:

    events  = POLL (topic)
    FOR All EVENTS:
            PROCESS (event)
            COMMIT (offset)
    ENDFOR

ENDWHILE
```

Each thread will create one of these loops, and Apache Kafka automatically assigns a partition to it if the number of partitions > = threads. We do not have an infinite loop in traditional producers, but each producer thread can independently write to the same topic. In our application, the Kafka broker will manage which partition will receive the message.

Everything in reactive programming is data, even errors. The application creates observables to listen to upcoming events and respond to changes accordingly. In both consumer and producer applications, the following logic exists:

Pseudo 2 - Reactive Design

```
SENDER/RECEIVER.ONDATA(callback(){

        PORCESS(event)

}).ONERROR(callback(){

        PROCESS(event) // error is also data

})
```

No matter if the application receives desired data or an error, no exception will be thrown. This type of programming facilitates to skip creating another branch in the application flow to handle the exception. Inside of the producer call back function, we can receive the acknowledgment to confirm the next message can be sent. The consumer callback function can commit the offset to comminute with Kafka; the message is consumed. In both implementations, we do not need to manage shared data related to events in parallel processes. Kafka has a mechanism that guarantees only one consumer at the same time can be connected to the one partition. Therefore, the offset in each partition can be only set by one thread or process. If the order of event matters, it will be challenging to manage the state with reactive design. Each consumer in Kafka has a client id, and the broker knows how many processes at the same time are connected to each partition. If a new partition is connected to the topic, the broker will divide all partitions among available consumer processes. We should also mention that because we use observable, reactive programming cannot be assumed the same as an observable pattern. The observable pattern in Object-Oriented programming is used to change a component without impacting the others, but it is not threaded safe and does not support non-blocking asynchronous processes. Reactive programming is a paradigm that can combine multiple patterns to fix the above issues.

3.1. Generate events without remote dependencies

Poll-based consumer application uses a commit sync strategy which commits the offset after processing of each event. The consumer fetches 10 events from the sample topic each 50ms. On the producer side, acknowledgment is set to all, a common strategy in applications with critical data. In this scenario and the others, one broker server manages the topic. In the reactive version, the same configurations are used, and the only difference is in the approach that consumer and producer threads will be run. Poll-based consumer and producer classes implemented runnable interface, and new threads can be created by initializing these classes and use the run methods. On the other hand, in the reactive design, functional programming concepts are followed, and instead of creating new consumer or producer threads, we can call "sendMessage" and "consumeMessage" multiple times. In each scenario number of partitions are equal to the number of threads. We should emphasize that still reactive functions can be executed in multiple threads, but no waiting strategy is required.

The figure [3] shows the throughput differences between the two designs in the producer. The reactive shows slightly better throughput up to 10 parallel tasks.

In contrast, traditional REST APIs can produce about 30 percent more KB/S data when the number of partitions increased. When there is a delay in receiving acknowledgment, the application context switch in reactive design causes the lower throughput. As the CPU is not required to wait for other resources to be released, no significant waiting time is observed during the process. Diagram [4] shows a significant CPU usage in reactive design, which without considering the throughput value could lead us to a wrong conclusion that this design can utilize the CPU more efficiently; however, it is clear that the computation resources are wasted to manage task context switch.

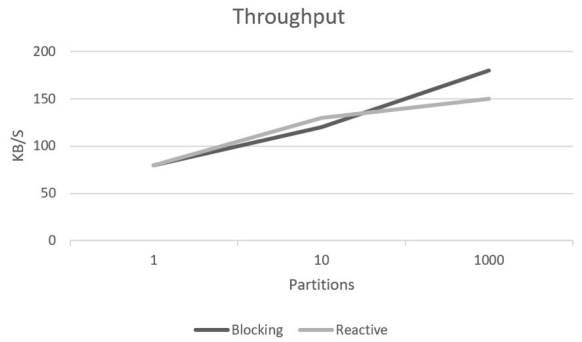

Figure 3 – Producer – Throughput – no delay

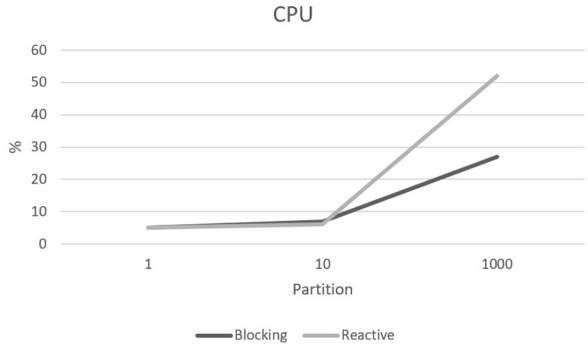

Figure 4 - Producer - CPU Usage - no delay

3.2. Generate events with remote dependencies

In this scenario, a 50ms delay was added before the producer can generate the next message, simulating the disturbed system's real scenario. As the diagram shows, the overall throughput is reduced in both scenarios; however, a reactive producer shows significantly better performance than a multi-threaded one. This diagram also shows, more concurrent tasks reduced the throughput in poll-based applications.

Although the CPU usage almost stays the same as the previous scenario, the reactive producer can utilize CPU to process other messages while other functions are waiting for the broker's response to confirm the events are stored correctly. In addition to this, some REST API threads were interrupted during this process, and they had to be restarted. This problem did not happen to reactive producers.

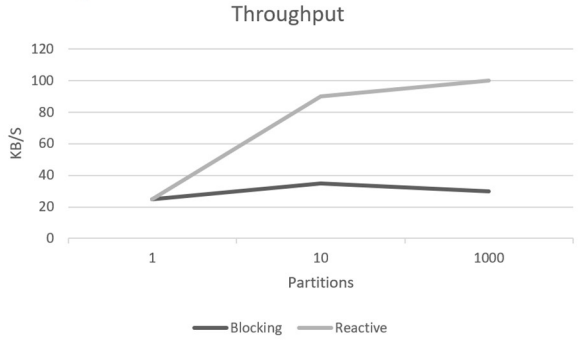

Figure 5 - Producer - Throughput - delay

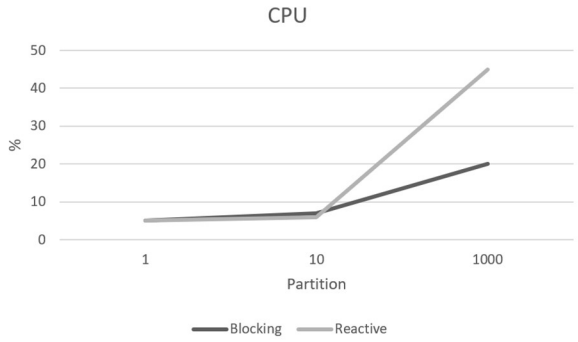

Figure 6 - Producer - CPU Usage - delay

3.3. Process events without remote dependencies

Kafka allows to consume messages from multiple partitions in one thread, but each partition can be only connected to one thread. Therefore, to maximize the concurrency, for each partition, one dedicated thread consumes the events. Consumers can fetch more than one message on each trip, reducing the number of requests to the broker. The third scenario almost shows a similar result to the first scenario. There is a minor advantage in using the blocking design, but both applications performed almost the same.

The CPU usage is less than the producer case, but reactive shows higher CPU utilization. Considering the throughput attribute, we can conclude that higher CPU usage cannot favor the reactive design. The consumer context switch between functions could be the reason for observing higher numbers.

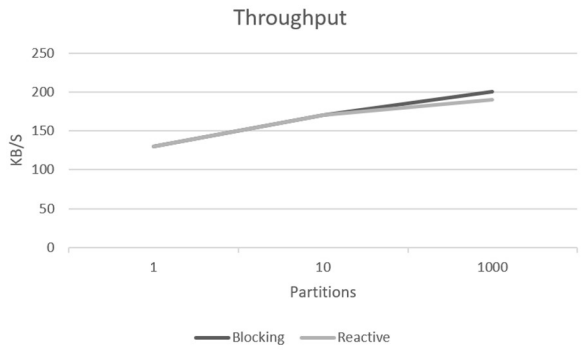

Figure 7 - Consumer – Throughput – no delay

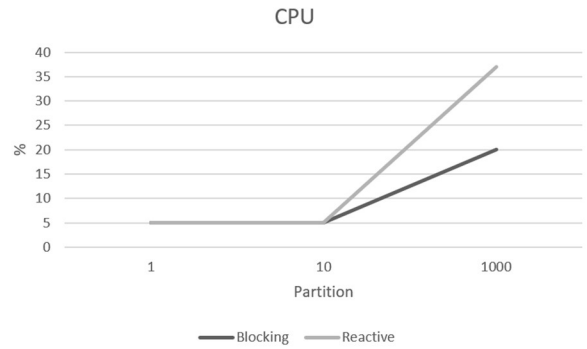

Figure 8 - Consumer – CPU Usage – no delay

3.4. Generate events with remote dependencies

In most cases, the consumer cannot commit the offset immediately. The events must be stored in the database, or extra information should be loaded before the processing is complete. In addition to the REST blocking APIs that we have been studying in this research, Kafka also provides stream APIs that can be used for data stream manipulation. Other tools, such as KSQL in Confluent, can filter, modify and store the data directly to the database outside of the application. Therefore, there are some options to improve the performance in this type of scenario, such as using the stream API directly rather than reactive APIs built on top of it, but if the business logic is complicated, developers prefer to have more control over events inside the application. This scenario added 50ms delay before allowing consumers to commit the offset to create a similar condition to real-world applications.

The CPU usage is higher in reactive design in topics supporting higher concurrency. This result is aligned with the better throughput reactive application shows in the diagram [10]. There is a notable difference in the throughput results, as shown in the diagram [9]; with a higher number of partitions, we can observe a decline in throughput in the poll-based consumer. However, the reactive design demonstrates the same behavior that we see in the producer application. One can argue that the consumer application's commit strategy harms the consumer application as it has to wait until the message processing is done before retrieving the next set of messages. This argument cannot be valid because of two reasons. First, the auto-commit strategy also is evaluated, and the result is very close to the commit sync approach. Second, for the first round of polling messages, the auto-commit might show a better performance, but since we consider applications with high volume, the application continually fetches the new events. Therefore, it has to stop for external resources after the first iteration before consuming more messages. The other question could be what would happen if the number of messages that are consumed increased. In other words, expand the size of batch size. This change cannot also impact the result because it can only reduce the number of trips to the broker, and unless communication with the broker is not efficient, it cannot be an issue for both solutions.

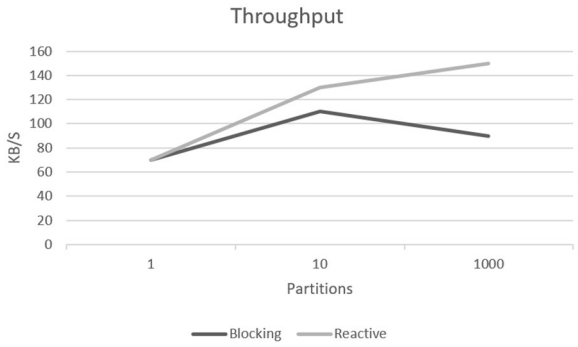

Figure 9 - Consumer – Throughput – delay

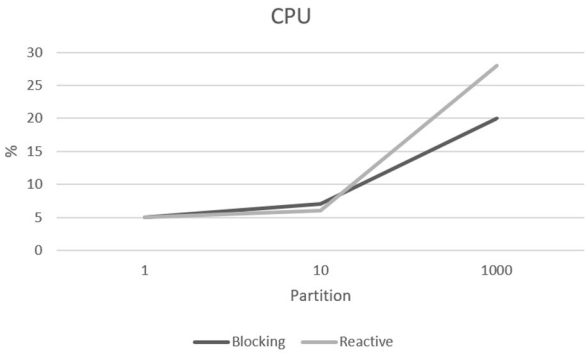

Figure 10 - Consumer- CPU Usage – delay

3.5. Compare solutions Latency

End-to-end latency is the time between when the application logic produces a record via “KafkaProducer .send()” to when the record can be consumed by the application logic via “KafkaConsumer.poll()”[9]. The latency was not discussed so far as it has a dependency on both producer and consumer. Four possible system designs can be created by combining blocking and Reactive designs for consumers and producers. As in the majority of applications, system architects have control over both producer and consumer design; for evaluating latency, both consumer and producer follow the same design pattern.

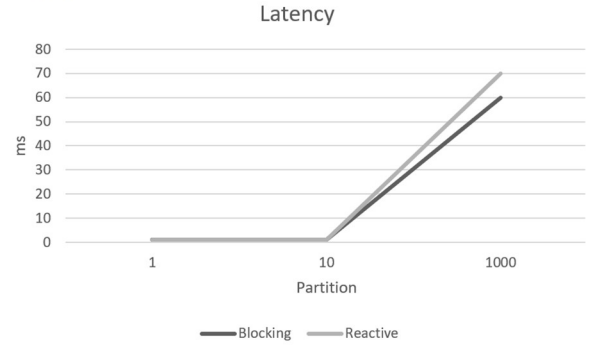

Figure 11 - E2E - Latency - no delay

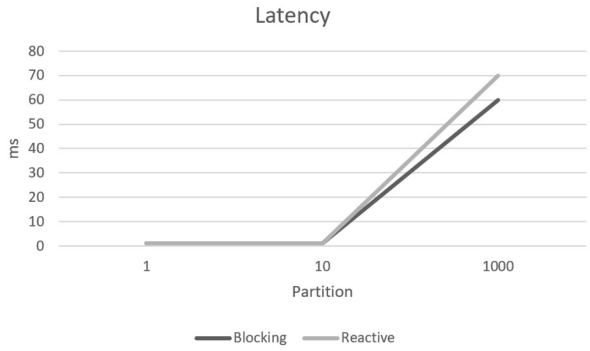

Figure 12 - E2E - Latency - delay

To calculate the latency instead of measuring the latency in the applications, broker data is used. Kafka broker monitors both consumer and producers in each topic, and it can calculate the latency. Figure [11] demonstrates the broker's latency when there is no delay either in generating or consuming the message. The reactive design has a higher latency, explaining that both consumer and producer in this design without intermediate delays performed slower than the blocking design. Also, increasing the number of partitions increases the latency as it puts more pressure on the broker to manage the topic.

The broker can manage multiple topics; therefore, adding more topics can impact the latency. Adding extra broker cannot increase the concurrency; however, as each broker can be responsible for some partitions, latency and throughput can be improved.

4. Reactive challenges

Reactive can mitigate performance issues caused by managing multiple threads. It is best practice to dedicate each thread to a single request, but this approach will reduce the application performance. Reactive programming uses functions, so context switch is lighter than threads if multiple processes run in one thread; however, it requires more knowledge and experience to implement the solutions. Therefore, testing, debugging, and resolving the defects are more time-consuming. One common problem with only leveraging reactive programming is that its tight coupling between computation stages in an event-driven callback-based or declarative program makes resilience harder to achieve as its transformation chains are often ephemeral and its stages the callbacks or combinators are anonymous, i.e. not addressable [10]. Consequently, processing the data and errors can be handled in silos, and applications cannot share a state if we run the functions in different threads.

Another challenge is observable, or callback concepts make understanding of the code logic more difficult. The Callback functions are triggered by external events, and developers could struggle to determine how a specific function is called. Besides, as shown in our scenarios, a single application was used for either consumer or producer. Once there are multiple nodes, there is a need to start thinking hard about things like data consistency, cross-node communication, coordination, versioning, orchestration, failure management, separation of concerns and responsibilities etc. i.e. system architecture [10]. Orchestration and failure management can be handled by container orchestration systems such as Kubernetes and can be mostly considered a DevOps operation, but developers should manage other requirements.

5. Using Reactive design in enterprise applications

In Java 1.1, the entire threading model was implemented as so-called "green threads", i.e. the entire Java VM was running in just one operating system process [11]. Java threads were implemented within the VM with their own scheduling algorithm [11]. This makes context switch between threads quick with less memory overhead, but the disadvantage of this

implementation is that Java with Green Threads could only use one core or processor in multi-core or multi-processor systems [11]. Some libraries and projects are fostering this idea to return greed threads to Java, such as project loom. Project Loom's mission is to make it easier to write, debug, profile, and maintain concurrent applications meeting today's requirements. Therefore, reactive programming might not be the only solution to optimize the application's performance when dealing with a high volume of data. As reactive programming in enterprise applications causes complexity, system architects could hesitate to use it in their design. Besides, reactive programming sometimes is confused with reactive systems, and understanding the difference among them could be another challenge for software engineers.

We observed that reactive programming improved the application performance, but considering the above challenges, should be recommended in enterprise applications? There is no decisive answer to this question. One of the popular proposed approaches is to use reactive microservices. Isolation is a major advantage of microservice. Each component in microservice can be developed and deployed independently; therefore, it is possible to have a heterogeneous component in this design. Reactive microservice are reactive systems that have both reactive and microservice system attributes. Reactive microservices can keep the state in the memory, so no trip is required to the database, and they can asynchronously communicate with other components. By adding an extra layer to a reactive system, developers only need to focus on implementing the business logic layer and not being concerned about its complexity.

Article [12] suggests reactive microservice architects in an event stream that can react independently. This article shows that an increasing number of microservices or increasing the participating microservices on the same event stream has a low impact on the individual microservice's performance. When an increased number of microservices begin causing a significant impact on an individual microservice's performance, then microservices can easily be scaled horizontally or workflows can be broken down [12]. They also show the event count, memory, and CPU usage in their proposal, suggesting that reactive microservice could be considered as a solution in enterprise applications. We need to emphasize that reactive programming is different from reactive systems and reactive microservices, but reactive microservices is built on top of reactive system manifesto, and reactive systems are built on top of reactive programming.

6. Future Study

Reactive programming can be used in distributed systems, and like other distributed solutions, it needs communication and synchronization. Article [13], studies the cost of consistency. This motivates a framework that enables the developers to select the best trade-off between consistency and overhead for the problem at hand [13]. In this paper, we study the event processing applications that are built on top of Kafka. Kafka can process each event as soon as it arrives, but other streaming platforms can process events in batches. Article [14] studies Apache Storm, Apache Spark, and Apache Flink event processing platforms. The result in this paper could be changed if another type of platform is used. [15] suggests a performance model that predicts the performance of Apache Kafka. They consider the disk retention parameter that can impact the performance of Kafka infrastructure. In our study, the default message retention is set to 7 days, and we did not consider this factor in the performance study. The storage hardware in this study has an average read speed of 3500 MB/s and a write speed of 2700 MB/s. This I/O speed is not realistic in distributed systems as not only the messages could be stored in a different machine, but servers do not use M.2 drives for their operations.

Our producer and consumer applications are not deployed in a distributed system. Big data applications cannot be run in a centralized system, but it is not easy to create test scenarios and create an environment to accurately measure the distributed system's performance. CloudSim [16] is a tool that can simulate a cloud platform. [16] explains the different layers of CloudSim architects, and as their future work, they will help developers simulate the cloud environment using CloudSim. This tool can be used for the scenarios that we did not evaluate in this study.

Reactive programming also is used in the front-end. However, some argue that front-end applications do not implement all reactive programming attributes, but frameworks like RxJS use observable to create asynchronous code. RxJS is a library for composing asynchronous and event-based programs by using observable sequences. [17] It provides one core type, the Observable, satellite types (Observer, Schedulers, Subjects) and operators inspired by Array#extras (map, filter, reduce, every, etc.) to allow handling asynchronous events as collections [17].

We also discussed reactive applications debugging could be challenging. [18] propose a method to debug reactive programming effectively. They developed an Eclipse plugin that can help developers to debug their application. Each time application receives a signal, the plugin creates a dependency graph and allows developers to see a visualized version of these

dependencies and create a breakpoint on them and step into the functions to inspect the object structure. They compare data at breakpoints, inspect memory, navigate object structure, and performance in their method to traditional Object-Oriented debuggers.

Apache Kafka's performance has a significant dependency on its configurations. Also, the environment that runs the infrastructure affects the test results. In our paper, we only used one Broker Server to manage the topic. Increasing the number of Brokers cannot improve the concurrency, but it can reduce the computing pressure from one broker. Running 1K partitions on one server can create a bottleneck and throttle communication speed. For the future study, we can increase the number of broker servers and evaluate the applications' performance with less pressure on one server. Using extra brokers scan give us the ability to increase the number of partitions as well. Almost without any exception, increasing the number of parallel processes at some point will have a negative impact on the performance so that we can measure this threshold in our system and developer a model for it.

This paper also discussed real-time event processing and the impact of reactive programming in this design. As another study opportunely, we can measure the performance when events are consumed in a batch process.

7. Conclusion

Reactive programming is not just another trending term. This article created a comprehensive review of this paradigm's impact on event processing applications and compared the result to a poll-based application that uses blocking methods to handle concurrency. We observed that events could not be processed in traditional APIs due to data dependencies, but reactive programming can release the computing units so other tasks can utilize them. CPU usage and events throughput in these applications in some scenarios showed a higher value, which can encourage software engineers to employ it in their design. However, there is no silver bullet in software design, and we showed that reactive programming could reduce the performance if there is no waiting time between events to process them. The complexity of these applications can grow if no framework is built around them.

8. References

[1] Anderson, S. (2020, September 28). How Fast Should A Website Load & How To Speed It Up. Retrieved November 06, 2020, from https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/

[2] Maglie, A. (2016). Reactive Java Programming. doi:10.1007/978-1-4842-1428-2.

[3] Bainomugisha, E., Carreton, A. L., Cutsem, T. V., Mostinckx, S., & Meuter, W. D. (2013). A survey on reactive programming. ACM Computing Surveys, 45(4), 1-34. doi:10.1145/2501654.2501666.

[4] 30, C. (2020, August 06). 5 Things to Know About Reactive Programming. Retrieved November 06, 2020, from https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/

[5] The Reactive Manifesto. (n.d.). Retrieved November 06, 2020, from https://www.reactivemanifesto.org/

[6] Jonas Bonér, V. (2016, December 02). Reactive programming vs. Reactive systems. Retrieved November 06, 2020, from https://www.oreilly.com/radar/reactive-programming-vs-reactive-systems/

[7] Waldman, M. (2018, January 16). Reactive Programming is event-driven as opposed to Reactive Systems that are generally... Retrieved November 06, 2020, from https://medium.com/@marilyn_2414/reactive-programming-is-event-driven-as-opposed-to-reactive-systems-that-are-generally-message-ecc079a9b8f4

[8] What is Apache Kafka? (n.d.). Retrieved November 06, 2020, from https://www.confluent.io/what-is-apache-kafka/?utm_medium=sem

[9] Tail Latency at Scale with Apache Kafka. (n.d.). Retrieved November 06, 2020, from https://www.confluent.io/blog/configure-kafka-to-minimize-latency/

[10] Jonas Bonér, V. (2016, December 02). Reactive programming vs. Reactive systems. Retrieved November 06, 2020, from https://www.oreilly.com/radar/reactive-programming-vs-reactive-systems/

[11] Limburg, A., Author Arne Limburg Arne Limburg a is software architect at open knowledge GmbH in Oldenburg. He has many years of experience as a developer, Arne Limburg a is software architect at open knowledge GmbH in Oldenburg. He has many years of experience as a developer, & Christopher. (2019, April 01). The fight for performance – Is reactive programming the right approach? Retrieved November 06, 2020, from https://jaxenter.com/the-fight-for-performance-157515.html

[12] Goel, D., & Nayak, A. (2019). Reactive Microservices in Commodity Resources. 2019 IEEE International Conference on Big Data (Big Data). doi:10.1109/bigdata47090.2019.9006584

[13] Margara, A., & Salvaneschi, G. (2018). On the Semantics of Distributed Reactive Programming: The Cost of Consistency. IEEE Transactions on Software Engineering, 44(7), 689-711. doi:10.1109/tse.2018.2833109

[14] Dayarathna, M., & Perera, S. (2018). Recent Advancements in Event Processing. ACM Computing Surveys, 51(2), 1-36. doi:10.1145/3170432

[15] Varga, E. (2016). Apache Kafka as a Messaging Hub. Creating Maintainable APIs, 187-201. doi:10.1007/978-1-4842-2196-9_12

[16] Long, S., & Zhao, Y. (2012). A Toolkit for Modeling and Simulating Cloud Data Storage: An Extension to CloudSim. 2012 International Conference on Control Engineering and Communication Technology. doi:10.1109/iccect.2012.160

[17] (n.d.). Retrieved November 06, 2020, from https://rxjs.dev/guide/overview
[18] Salvaneschi, G., & Mezini, M. (2016). Debugging reactive programming with reactive inspector. Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16. doi:10.1145/2889160.2893174