Reliability enhancements for high-availability systems using distributed event streaming platforms

Ana-Maria Dincă
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
ana_maria.dinca@stud.etti.upb.ro

Sabina-Daniela Axinte
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
axinte_sabina@yahoo.com

Ioan C. Bacivarov
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
ibacivarov@yahoo.com

Gabriel Petrică
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
gabriel.petrica@upb.ro

***Abstract*—The effectiveness of a performant high-availability system can be measured by the transaction operations' execution speed and reliability metrics. The performance and reliability of traditional database interactions can be improved through multiple techniques, including incorporating emerging technologies, such as distributed event streaming platforms, in the new or existent software infrastructure. This study is underlining the improvements that were obtained through Kafka integration with existing database operations by running a series of performance tests on two separate endpoints, that communicated with an independent software module. One endpoint used an asynchronous communication system, through event streams, and the other one used a traditional synchronous communication, through REST calls. Performance indicators such as throughput, error rate and execution time were extracted and analyzed, and it resulted that the data sent through Kafka had a 0% loss rate. Also, due to the numerous configuration options available, the reached throughput, using the asynchronous communication system, was more than 10 times higher. The study continued with an analysis of other event streaming platform vendors, with their particular features, disadvantages and other companies that use that specific technology. The final section is dedicated to a guide of best practices that ease the transition to an asynchronous messaging system between independent software modules, or just the addition of a new event streaming pipeline, and how to achieve performant and resilient communication, with configurable back-up mechanisms.**

***Keywords*— database reliability metrics, distributed event streaming platforms, Kafka, performance testing, high throughput processing pipeline**

I. Introduction

The microservices architecture became a popular alternative, being favorized over the traditional and, oftentimes, obsolete monolithic architecture. Its primary goal is to decouple the interdependent software modules found in a web application, and thus, whenever a service becomes unavailable, the other ones continue to function as expected, increasing the overall availability and resilience.

The microservices theory demands the modules to be completely independent, although most of the time, in a day-to-day implementation, there are numerous scenarios when they need to communicate with each other, for example, whenever there is a change in the state of a record that needs to be propagated in multiple business flows.

The communication between microservices is most often done *synchronously*, through traditional REST APIs, but with a rise in the amount of information sent for processing and Big Data, the fragility of this type of communication was revealed. An unsuccessful REST call will, by default, result in data loss, due to various reasons: the information is not backed-up, or the failed call is not further processed and the retry mechanisms are not implemented. The synchronous communication requires an additional fail-safe mechanism to be implemented in the software application, in order to increase the resilience of the transferred information.

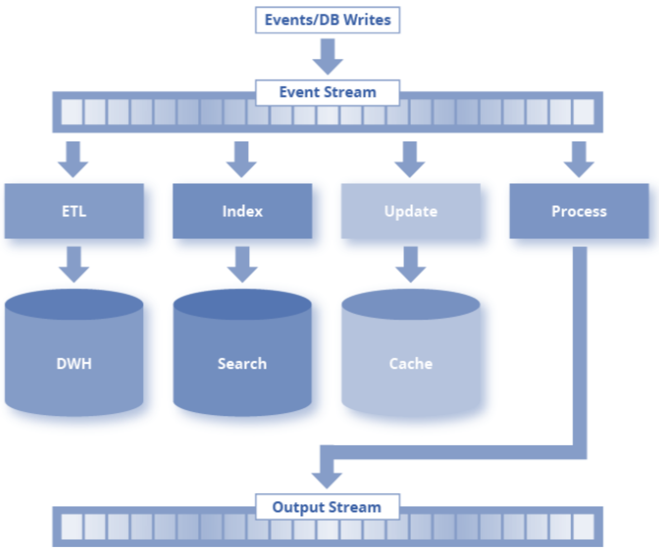


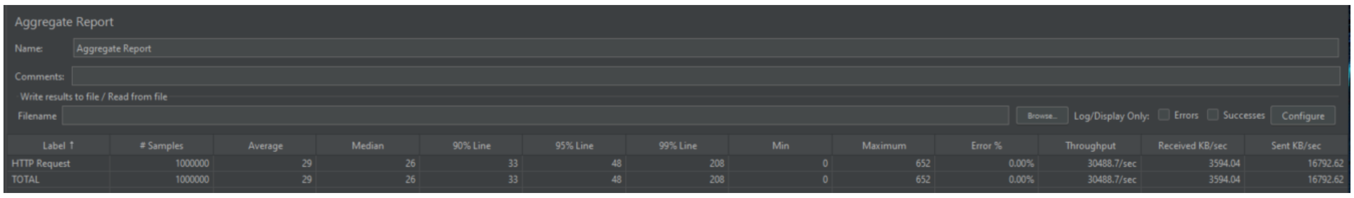
Fig. 1. Example of processing real-time data through event streaming [1]

The asynchronous communication is highly recommended for independent software modules (Fig. 1), and it can be achieved using event streaming platforms. Once they are integrated with a software application, the enhancements they provide are irrefutable, as they ensure data resilience in the mishap of an unresponsive information system. This is why the demand is rapidly growing, with more and more companies that develop software products choosing to migrate to an asynchronous communication and use an event streaming platform for their platforms based on a microservices or a modular architecture.

II. Analysis of Event Streaming Platforms and the Utility of Queuing Mechanisms

The usage of event streaming platforms is the emerging solution to an ever-growing flux of data that needs to be processed, visualized, and updated constantly. Depending on the target availability and resilience of the software application in question, the data loss acceptance level can vary from none, to a lower or medium percentage.

Investing in an architecture similar to the one presented in Fig. 1 also ensures an increase of the overall resilience achieved by the software application. In this scenario, the events are stored in the pipeline, and they are available to be pulled out by the platforms asynchronously, unlike a traditional architecture, where the data is sent through a REST API call, and depends on the platform's status and availability at that exact moment, whether it is capable to process the incoming request or not, possibly resulting in data loss.


Aggregate Report

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename

Label ↑	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/sec	Sent KB/sec
HTTP Request	1000000	29	26	33	48	208	0	652	0.00%	30488.7/sec	3594.04	16792.62
TOTAL	1000000	29	26	33	48	208	0	652	0.00%	30488.7/sec	3594.04	16792.62

Fig. 2. Example of aggregated test result using JMeter

This new mechanism of data transfer allows real-time information to be passed to multiple software platforms, through resilient and scalable pipelines. The two essential components of the previously mentioned processing mechanisms are the *events* and the *event streams*. The events represent a change of state, such as a customer finalizing an order, or an account deactivation. Any number of authorized and interested parties can *subscribe* to receive these events, in order to process them however they consider necessary. The *event stream* is a series of events that are processed and fed to a pipeline, where they are persisted and ordered by the timestamp when the event occurred [2].

An example of a standard architecture that has integrated an event streaming platform is depicted in Fig. 1. The events are pushed on the event stream, which stores them locally and feeds them continuously to any platform subscribed to it. The events are ingested by the modules, processed, and the result can be either stored directly, or used to change the state of an existing record, or pushed on another event stream.

The indispensability of the input data is determined by the business requirements and the target availability of a software application. For example, building an eHealth application, that monitors patients' vital signs information, requires data stream processing [3], because any lost event/information can result in a misguided health diagnostic, and therefore an erroneous treatment.

III. Performance Review on Back-up Mechanisms Implemented Using Event Streaming Platforms

This section is focused on measuring the degree of performance reached by a software application in two main scenarios: with and without an integration of an event streaming platform.

The performance metrics used in the analysis below were gathered using *JMeter* (Fig. 2), which is a load testing tool, able to simulate multiple users that are sending concurrent requests, for a pre-defined number of times, or continuously,

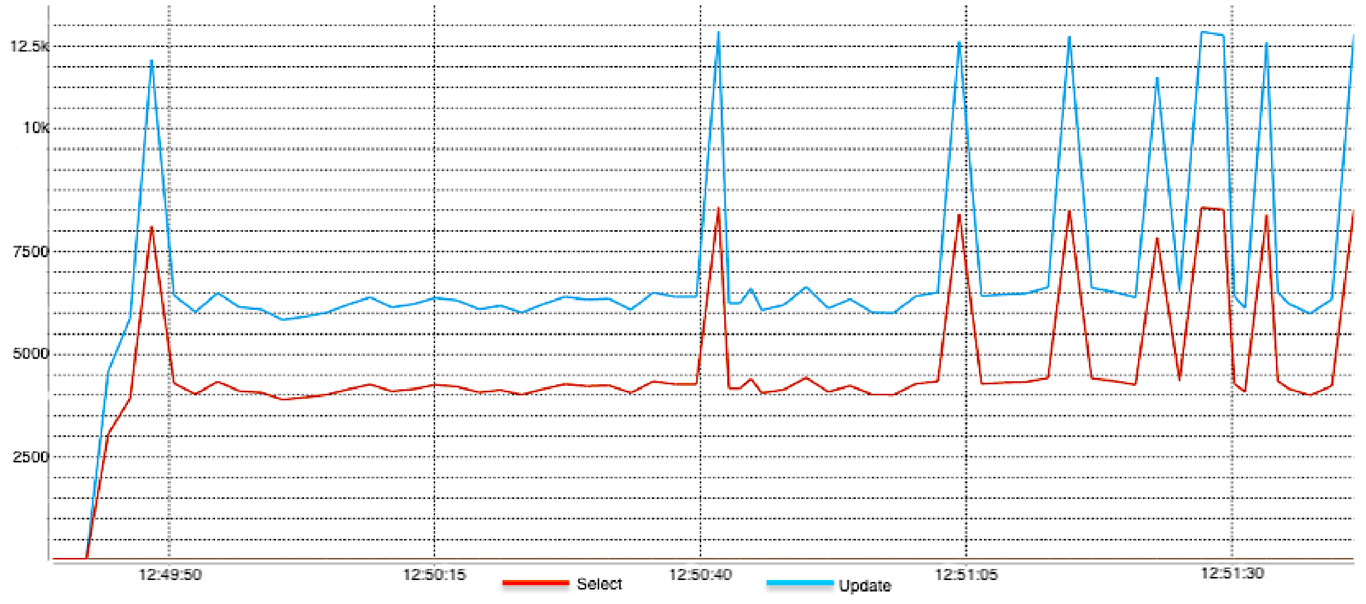


Fig. 3. Requests processed by the database whithout Kafka integration

The integration of event streaming platforms in a *microservices architecture* is essential, as it eliminates the unavoidable coupling between separated services, due to their need of communication among each other of certain changes.

until stopped, and measure the error rate, the throughput, and many other performance indicators.

The software application used for this research has been developed using Java 17 and Spring Boot 3 (Fig. 5). The chosen event streaming platform was Kafka, one of the most popular solutions in use at the moment, especially with platforms built using the Spring Boot framework. This demo application scenarios, as the performed operation had a rather low complexity, consisting only of a couple of data processing operations, prior to an insert query.

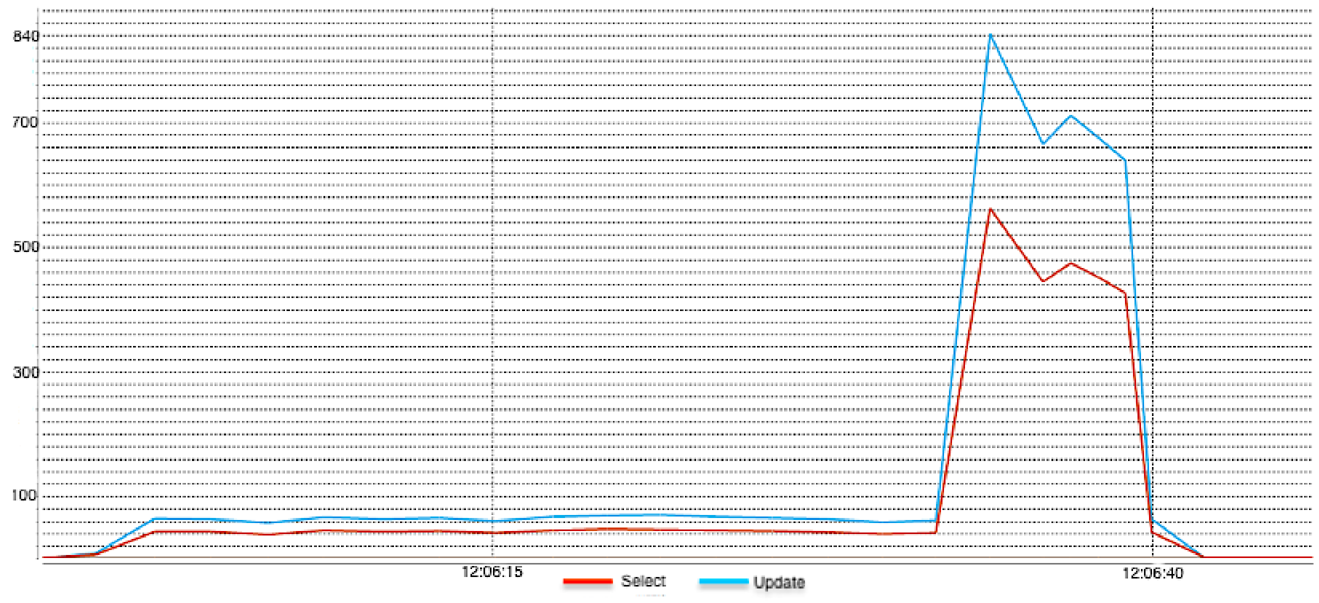


Fig. 4. Requests processed by the database with Kafka integration

consists of two separate modules that handles user activation operations, using multiple implementations and communication options. When a new user is created, then the client side sends a JSON object to the back-end component, containing general data about the user, an address, and its contact details.

The Kafka configuration implemented for this application has both a consumer and a producer, each one in a separate module. The producer uses a minimum of 15 threads for processing, with the possibility to increase to 250 threads, in case of a heavy load. The buffer memory that persists records, until they can be sent on the designated topic is 67,108,864 Bytes, which is double of the default value. The buffers used for receiving and sending data are also twice or thrice higher than the default values, in order to increase the maximum throughput produced by the software infrastructure.

There are two separate APIs used for the user activation. One receives the user data and sends it to the activation module through a REST call, where it is directly saved in the database, and the other one processes the data and then it writes it on a Kafka topic, from where it is then read asynchronously by the activation module, using multiple concurrent consumer threads.

TABLE I. PERFORMANCE INDICATORS GATHERED USING JMETER

Type	Number of samples	Error %	Throughput	Sent kB/sec	Execution time
No Kafka	1 Mil	0.00%	2062.9/sec	1124.15	08'26.21"
Using Kafka	1 Mil	0.00%	30488.7/sec	16792.62	00'33.62"

Two simulations were run, each one calling one of the APIs mentioned previously. The testing scenario involved 10,000 users accessing the back-end server at the same time, and each one sending exactly 100 activation requests, adding up to a total of 1,000,000 hits. The error rate in 0% in both

As presented in TABLE I, the API that sent the activation requests to another module via a Kafka topic, for further processing and persistence, was 15 times faster than the API using a traditional persistence mechanism. The obtained result is directly determined by the usage of an event streaming platform, which allowed the application to reach a throughput more than 10 times higher, that sent all the data load to multiple asynchronous processing mechanisms, instead of processing the requests sequentially.

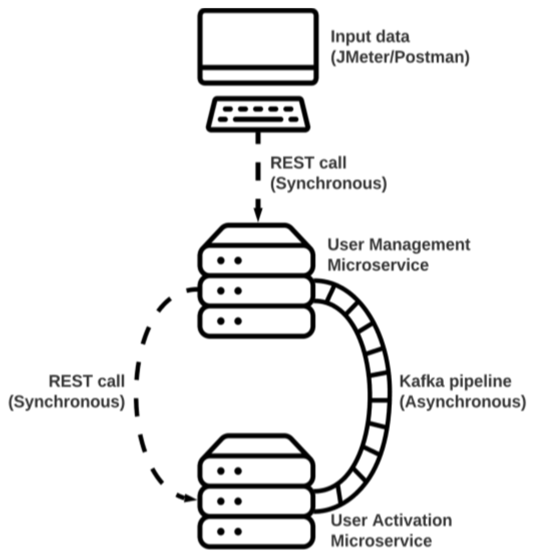


Fig. 5. The architecture chosen for the demo software application

Two more simulations were run using *Postman*, a tool created for testing REST APIs, automating test scenarios and performing load tests. 10,000 POST requests were sent to the previously mentioned software application, and further to each activation endpoint, the one saving new users directly in the database, and the one reading the activation requests asynchronously from the Kafka topic. It was performed a simulation of an unresponsive microservice by configuring the activation module to restart periodically, using random time intervals. Through this configuration, it was possible to measure the success rate achieved when the software application was using a Kafka integration versus the traditional implementation, while the activation service uptime was slightly intermittent.

The simulation results are presented in TABLE II, and according to them, the success rate was 100% when using a Kafka integration with a Spring Boot microservice. There was a loss of 1372 requests, that were sent directly to the activation module while it was unresponsive, and there was no other back-up mechanism in place. The requests sent using Kafka were processed and persisted on the designated topic, and they were pulled out, read and processed by the activation service while it was up and running, and if the module stopped while processing a record, then the acknowledgement of the successful processing event was not sent, the operation was resumed when the application started again.

TABLE II. PERFORMANCE INDICATORS GATHERED USING POSTMAN

Type	Number of samples	Tests passed	Tests failed	Users saved in DB	Error %
No Kafka	10,000	8,628	1,372	8,628	13.72%
Using Kafka	10,000	10,000	0	10,000	0.00%

The current implementation was optimized with bigger buffers for sending data and for temporary persistence of information, in order to accommodate a bigger Kafka request, that contained details about the user, its address and contact details. As the software application evolves and the business flows are separated in different modules, the received information for a user activation will be sent to each corresponding module. For example, the address will be sent separately to a distinct module focused on address operations, such as handling and tracking deliveries. This will result in using smaller buffers for the Kafka configurations, and therefore freeing the system resources for other processes that might make use of them.

IV. EVENT STREAMING PLATFORMS VENDORS COMPARISON

Whilst synchronous communication between internal microservices became increasingly obsolete, the research started focusing on sending information *asynchronously*. At first, this was achieved through reactive programming, and, for the last couple of years, it shifted to event streaming. Therefore, multiple vendors came forth with their own event streaming platforms, some of the most popular solutions being Apache Spark, Apache Kafka, Apache Flink, Spring Cloud Data Flow, Amazon Kinesis, Cloud Dataflow and IBM Streams.

Apache Spark is, at the moment, the biggest open-source big-data processing project, with a massive supporting community of developers. It uses in-memory computing mechanisms, which positions its processing speed remarkably high above its competitors' [4]. Another advantage is that it is also multilingual, as it supports some of the most popular programming languages such as Python, Scala, Java and SQL. One of its disadvantages is that, although it allows batch processing, it is limited to dividing batches according to predefined time window, thus it doesn't support record-based window criteria, generating very irregularly sized batches of data. Some of the companies that have integrated this technology are Uber, Slack and Shopify [5].

Apache Kafka is an open-source data streaming platform that handles data published by multiple sources, stores it, processes it, and, finally, delivers it to multiple consumers [6]. It also provides a Java API that facilitate aggregating, filtering, grouping, and joining data, without writing any additional code. The complexity of this system is one of its main disadvantages, as it has numerous components that must be integrated, such as brokers, producers, consumers, and Zookeeper, and it can lead to a difficult learning curve [7]. Some of its advantages are that it is easy to integrate with existing applications, and it has a low latency, of up to 10 milliseconds. Apache Kafka is integrated in software platforms such as Zalando or Pinterest [8].

Amazon Kinesis is a service that can collect, process and analyze streams of data in real time. The input data can come not only from event streams, but also from social media feeds or application logs. Other advantages are that it is easy to set up and maintain, and that it integrates with processing and analysis solutions for Big Data, developed by Amazon. Some drawbacks are its steep price, the service being billed per hour and per traffic, and its complicated documentation. Some examples of companies that use Amazon Kinesis are Figma, Deliveroo and Lyft [8].

Cloud Dataflow is an event streaming platform developed by Google, and its main goal is to execute data processing pipelines. This platform has a serverless approach, capitalizing on Google's best cloud network topology and performance. It provides real-time AI-powered processing patterns, that can be used for anomaly detections. Its biggest disadvantage is the restriction to Cloud Datastore services, and the expensive bill that comes with it. Spotify and The New York Times use Cloud Dataflow for event streaming [8].

V. BEST PRACTICES IN ASYNCRHONOUS COMMUNICATION IMPLEMENTATION USING EVENT STREAMING PLATFORMS

The event streaming platform is a must in order to build performant and resilient software applications, especially the ones based on microservices architecture, as they require asynchronous communication between modules, potentially enhanced by a retry-on-fail back-up mechanism [9].

There is no fixed number of topics that should be defined for a software application, but in order to reach performance, the *topics related to metric streams should be separated from the ones used for events* that represent a change of state in a business entity. Also, busy streaming topics, that have a lot of input data, should take advantage of the parallelism features provided by event streaming platforms, and use a *bigger number of partitions*, for *increasing the throughput*.

When a change occurs in the current state of a software module, and the information has to be propagated to other modules, a new event is generated and broadcasted on the pre-defined topics. Such *events should have a light structure*, containing *only the indispensable information*, as any supplementary and irrelevant sent data results in a waste of resources for the information system.

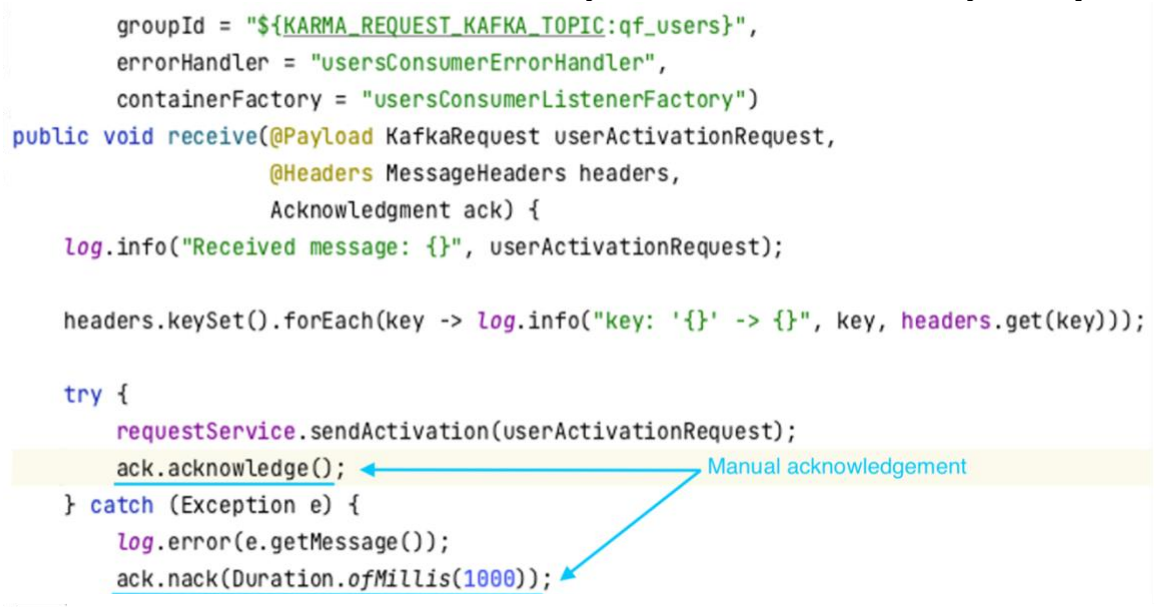
Event streaming platforms generate a considerable amount of logs, which, if unmanaged, can outgrow the existing disk size, leading to data loss and possible left undocumented incidents due to a lack of memory. This can be prevented by *controlling the topic retention*. The retention period can be calculated using the data rate and the available disk space.

An event streaming platform such as Kafka has a customizable downloading speed, and by default, depending on the used version, it varies between 64kb and 100 kb. These speeds are considered relatively slow, especially in a high throughput environment, therefore it is strongly recommended to override the default value of the *receiver buffer size* with one better suited for the business requirements, *between 1 and 16 MB*.

Writing an event in a topic is not always successful, therefore, to avoid data loss and to increase the fault tolerance, there are internal retrial mechanisms built in. The recommended *number of retries* depends on how vital is the information sent on the topic, but for an application with no tolerance for data loss, the recommended number of retries is "Integer.MAX_VALUE", which is equal to $2^{31} - 1$.

There are built in mechanisms that can confirm whether a request was sent successfully or not, with multiple customization options. The *acknowledgment* could be completely disabled, the retry mechanisms will never be used, and the record will be considered as sent as soon as it is added to the socket buffer. The acknowledgment can be partially activated, and if a failure occurs after the record is written on the buffer by the leader node, but before it is in fact sent by the replicas, the data can be lost, as the event was already considered to be sent, and the retrial will not be attempted. Lastly, the most reliable setting is to wait for all the replicas to acknowledge the record, and therefore, the information will not be lost as long as at least one replica remains alive.

A scenario with an increased complexity, that handles vital information, and whose event acknowledgement is of upmost importance, would benefit from manually acknowledging a successfully processed record, as presented in Fig. 6. The Kafka consumer receives the Acknowledgement object, and, if the save operation is completed, then it calls the *acknowledgement()* method. Otherwise, the *nack()* method will be called, the remaining unprocessed records will be discarded, and the consumer will re-seek the partition, until the record is redelivered [10].



```
        groupId = "${KARMA_REQUEST_KAFKA_TOPIC:qf_users}",
        errorHandler = "usersConsumerErrorHandler",
        containerFactory = "usersConsumerListenerFactory")
public void receive(@Payload KafkaRequest userActivationRequest,
                    @Headers MessageHeaders headers,
                    Acknowledgment ack) {
    log.info("Received message: {}", userActivationRequest);

    headers.keySet().forEach(key -> log.info("key: '{}' -> {}", key, headers.get(key)));

    try {
        requestService.sendActivation(userActivationRequest);
        ack.acknowledge();
    } catch (Exception e) {
        log.error(e.getMessage());
        ack.nack(Duration.ofMillis(1000));
```

Fig. 6. Kafka Processor using manual acknowledgement

Conclusions

This paper analyzed the novel event streaming platforms and how they can be used in order to achieve asynchronous communication between independent software components, in order to ensure reliability and increase performance indicators such as throughput, success rate, and response time.

The *second section* was solely dedicated to detailing how an event streaming platform works, how the events are published on streams, and how any authorized platform can subscribe to any number of topics and receive records. This section also emphasizes on how event streams are used for asynchronous communication between software services, especially in microservices architecture, and how this type of messaging can significantly increase the overall data resilience.

The majority of published research related to event streaming platforms, up to this date, is mostly focused on the configuration options available for the producing or processing components [11], or mathematical formulae, calculations and predictions on how the performance of the platform varies while using different configurations [12]. The comprehensive performance results and the testing scenarios are seldom presented in the research papers.

Thus, the *third section* presents the exhaustive performance results obtained while performing the necessary research for this paper. The software application used for the testing scenarios is based on a microservices architecture, and was built using Java, Spring Boot, and Kafka, as the event streaming platform. The independent modules communicate using both synchronous and asynchronous communication, through REST calls and, respectively, Kafka.

For the first two scenarios, JMeter was used for load testing the software application, in order to evaluate the impact of heavy traffic and how modules handle 1,000,000 requests, sent by 10,000 concurrent users. The data was sent for processing through a POST call and through Kafka, and the results are presented in TABLE I. The success rate was 100% for both test cases, but the asynchronous communication, through Kafka, had a *throughput more than 10 times higher*, and the *processing was 15 times faster*.

The next two testing scenarios were ran using *Postman* and 10,000 sequential requests, using both methods of communication, REST and Kafka. The software application was configured to restart at random time intervals, in order to simulate an intermittent behavior, and measure the success rate for each testing case. The results revealed that data sent synchronously, while the application was unresponsive, was irreversibly lost, especially because there was no back-up mechanism in place, and there was an error rate of over 10%. On the other hand, the data sent through Kafka was processed asynchronously, whenever the service was available and responsive, and the success rate was 100%.

The *fourth section* of the paper provides a short analysis on the most popular vendors of event streaming platforms on the market, available at this moment. For each one of them, there were presented their advantages and disadvantages, and other notable software applications that is currently using the mentioned vendor.

The fifth and *final section* is comprised of several guidelines that can be followed during an integration with an event streaming platform, or while reconfiguring an existing one, in order to achieve a performant and resilient asynchronous communication, therefore increasing the achieved survivability of the information system. There is a focus on the type of acknowledgement that should be decided on a case-by-case analysis, relying on the business requirements and the acceptable level of losing data during event processing. Additionally, streaming topics should be specific, dedicated, and separated according to the business requirements. Topics that handle high volumes of data should use an increased number of partitions, in order to maximize the data throughput. It is recommended for the events to have a light structure, containing only the minimum level of information. If data traffic and storage volume is a concern for the project under discussion, then the topic retention should be configured according to the required retention period and the available disk space.

References

[1] Hazelcast, "Event Stream Processing," https://hazelcast.com/glossary/event-stream-processing/. [Accessed 21 08 2023].

[2] D. Dhanushka, "Anatomy of an Event Streaming Platform," 08 04 2021. https://medium.com/event-driven-utopia/anatomy-of-an-event-streaming-platform-part-1-dc58eb9b2412. [Accessed 22 08 2023].

[3] P. Schneider and F. Xhafa, "Chapter 2 - Data stream processing: models and methods: The complexity of data stream processing," in *Anomaly Detection and Complex Event Processing Over IoT Data Streams With Application to eHealth and Patient Data Monitoring*, Elsevier, 2022, pp. 29-47.

[4] I. Education, "Hadoop vs. Spark: What's the Difference?," 27 05 2021. https://www.ibm.com/blog/hadoop-vs-spark/. [Accessed 29 09 2023].

[5] KnowledgeHut, "Apache Spark Pros and Cons," 14 07 2023. https://www.knowledgehut.com/blog/big-data/apache-spark-advantages-disadvantages. [Accessed 26 08 2023].

[6] Red Hat, "What is Apache Kafka?," 10 10 2022. https://www.redhat.com/en/topics/integration/what-is-apache-kafka. [Accessed 29 08 2023].

[7] "Drawbacks of Kafka," 24 04 2023. https://medium.com/@harsh.b26/d-of-kafka-fcb459a50ee5. [Accessed 29 08 2023].

[8] R. N. Dosio, "A look at 8 top stream processing platforms," 17 01 2022. https://ably.com/blog/a-look-at-8-top-stream-processing-platforms#further-reading. [Accessed 29 08 2023].

[9] B. H. Jethva, "Kafka Best Practices-Topic, Partitions, Consumers, Producers and Brokers," 15 07 2022. https://cloudinfrastructureservices.co.uk/kafka-best-practices-topic-partitions-consumers-producers-and-brokers/. [Accessed 30 08 2023]

[10] Spring , "Interface Acknowledgment," https://docs.spring.io/spring-kafka/api/org/springframework/kafka/support/Acknowledgment.html [Accessed 10 09 2023].

[11] B. R. Hiraman, C. V. M. and K. A. C., "A Study of Apache Kafka in Big Data Stream Processing," in *International Conference on Information, Communication, Engineering and Technology (ICICET)*, Pune, 2018.

[12] Z. S. K. W. Han Wu, "Learning to Reliably Deliver Streaming Data with Apache Kafka," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Valencia, Spain, 2020.