ISSN: 2836-8495

Research Article

Journal of Current Trends in Computer Science Research

Design and Optimization of a Zookeeper and Kafka-Based Messaging Broker for Asynchronous Processing in High Availability and Low Latency Applications

Sourabh Sethi[1]*, Sarah Panda[2] and Sandeep Hooda[3]

[1]Sourabh Sethi, Infosys, Jersey City, New Jersey, 07306, USA

[2]Sarah Panda, Microsoft, Seattle, Washington, 98101, USA

[3]Sandeep Hooda, Infosys, Alpharetta, GA, 30004, USA

[*]Corresponding Author
Sourabh Sethi, Design and Optimization of a Zookeeper and Kafka-Based Messaging Broker for Asynchronous Processing in High Availability and Low Latency Applications.

Submitted: 2024, Feb 13; **Accepted:** 2024, Mar 08; **Published:** 2024, Mar 12

Citation: Sethi, S., Panda, S., Hooda, S. (2024). Design and Optimization of a Zookeeper and Kafka-Based Messaging Broker for Asynchronous Processing in High Availability and Low Latency Applications. *J Curr Trends Comp Sci Res, 3*(2), 01-07.

Abstract
In modern distributed systems, achieving high availability and low latency is crucial for ensuring optimal performance and responsiveness. This paper aims to explore and enhance the capabilities of a messaging broker system by leveraging Apache Zookeeper and Apache Kafka for asynchronous processing. The focus is on designing a robust architecture that ensures high availability while minimizing processing delays, thus meeting the stringent requirements of contemporary applications.

Keywords: Apache Zookeeper, Apache Kafka, Messaging, Low Latency Applications

1. Introduction

Let's consider an example involving a messaging system, whenever a message is sent from one user to another, such as user1 sending a message to user2, the message is stored in user2's database. Following this action, several tasks need to be performed: Notify user2. Send an email to user2 if they haven't read messages in the last 24 hours. Update relevant metrics in analytics. However, we don't want the sender of the message to wait for these tasks to be completed. Additionally, if any of the above tasks fail, it should not imply that the message itself failed to be sent successfully. How can we achieve immediate success? To address these requirements, we utilize a Persistent Queue. A Persistent Queue is durable, meaning that data is written to a hard disk to ensure it is not lost. Persistent Queues work on a model called pub-sub (Publish Subscribe) such as apache Kafka. Kafka uses Zookeeper internally, Zookeeper serves as a dis-tributed coordination service for Kafka, providing essential functionalities such as cluster coordination, metadata management, leader election, and configuration management, which are crucial for Kafka's re-liability, scalability, and fault tolerance. Why opt for Zookeeper? In a Master-Slave architecture, the mas-ter is the central point for all write operations, excluding the slave machines. Consequently, all clients (app servers) need to be aware of the current master. While this is manageable when the master remains consistent, the challenge arises if the master becomes unavailable. In such cases, a new master needs to be selected, and all machines must synchronize and be aware of the change. If posed as a problem, how would one address this issue? A simplistic approach involves having a dedicated machine assigned the sole task of tracking the master. Whenever an app server needs information about the current master, it would query this designated machine.

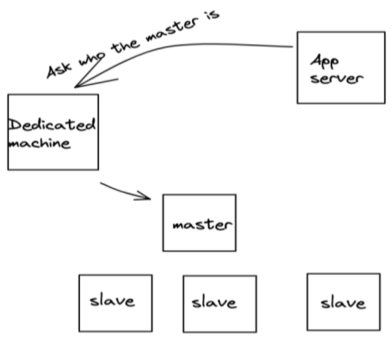


Figure 1: Who is the Master?

Nevertheless, there are two concerns associated with this strategy.
1.) The designated machine becomes a sole point of failure; if it experiences downtime, write operations cease, even if the master remains operational.
2.) Introducing an extra hop for every request is a drawback, as it necessitates determining the master. To address concern 1, an alternative could be employing not just one machine but rather a group of machines or clusters.

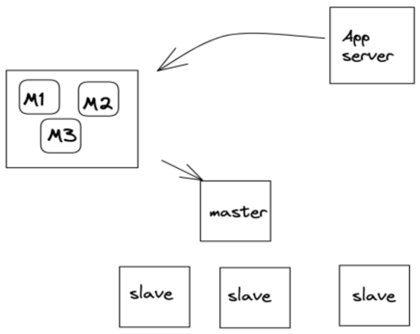


Figure 2: Cluster of Machine

How do these machines ascertain the master's identity? How can we guarantee consistent information about the master across all machines? How do we enable direct access for app servers to the master with-out ad-dictional hops? The solution lies in a cluster management system like Zookeeper.

2. Zookeeper

2.1. Strongly Consistent Consensus System

Zookeeper is a versatile system designed to maintain data in a strongly consistent format, with further de-tails to be discussed later in this paper. The storage structure in Zookeeper closely resembles that of a file system. For instance, there is a root folder containing a collection of files or directories.

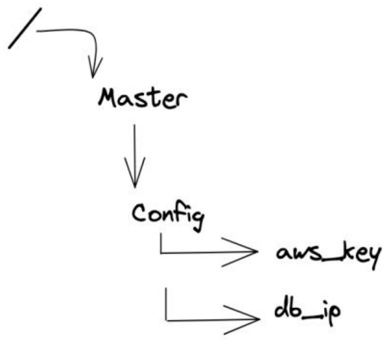


Figure 3: ZK nodes

All these files are known as ZK nodes in zookeeper.

2.2. ZK Nodes

Every file in Zookeeper falls into one of two categories Ephemeral & Persistent Ephemeral: Ephemeral nodes (not to be confused with machines; nodes represent files in the context of Zookeeper) are files where the data written is only valid as long as the machine/session that wrote the data remains active. In simpler terms, the data on these nodes remains valid only if the machine continuously sends heartbeats to ensure its presence. Once an ephemeral node is created, other machines/sessions cannot write any data to it. An ephemeral node is exclusively owned by exactly one session/machine, al-lowing only the owner to modify the data. If the owner fails to send a heartbeat, the session terminates, and the ephemeral node is automatically deleted. Subsequently, any other machine can create the same node/file with different data. These nodes are commonly employed for tracking machine status, deter-mining the master of a cluster, implementing distributed locks, etc. Further details will be provided later.

Persistent: Persistent nodes are nodes that persist unless explicitly deleted. They are typically utilized for storing configuration variables.

2.3. ZK Node for consistency / Master Election

For the sake of simplicity, let's envision Zookeeper as a solitary machine initially (we'll address multiple machines later on). Consider a cluster named X consisting of several storage machines.

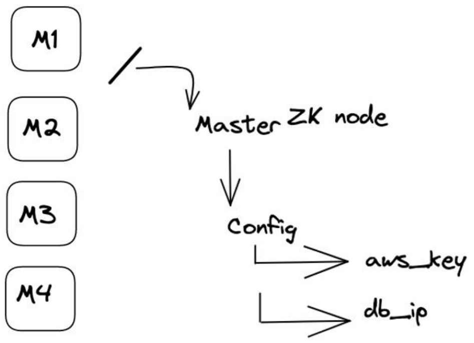


Figure 4: ZK Nodes Consistency

All of them aspire to become the master. However, only one can assume this role. Thus, we face the challenge of determining (who will become the master). To address this, it instructs all machines to at-tempt writing their IP addresses as data to the same ephemeral Zookeeper node (let's designate it as /clusterx/master_ip). It's important to note that only one machine will successfully write to this ephemeral node; all other attempts will fail. Suppose M2 successfully writes its IP address to /clusterx/master_ip. Now, as long as M2 remains operational and continues sending heartbeats, /clusterx/master_ip will retain M2's IP address. Consequently, any machine attempting to read data from /clusterx/master_ip will receive M2's IP address in response.

2.4 ZK: Setting A Watch

The issue of additional hops persists. If every app server and machine must communicate with Zookeeper for every request to determine the master, it not only burdens Zookeeper with excessive load but also in-creases the number of hops for each request. How can we tackle this challenge? Upon reflection, it's ap-parent that the data on ephemeral nodes changes infrequently, perhaps only once a day or even less fre-quently. It seems impractical for every client to repeatedly query Zookeeper for the master value when it remains static most of the time. Therefore, why not invert the process? Instead of clients continually que-rying for updates, we can provide them with the current value, instructing them to use it without the need for repeated inquiries. We assure them that whenever this value is updated, they will receive notification. Zookeeper employs a similar approach to resolve this issue through a "subscribe to updates on this ZK node" feature. On any Zookeeper node, you can establish a watch (subscribe to updates). In Zookeeper, all read operations offer the option of setting a watch as a secondary action. For instance, if I'm an app server and I set a watch on /clusterx/master_ip, I, along with all other clients who have set a watch on that node, will be notified when the node's data changes or when it is deleted. This implies that when clients set a watch, Zookeeper maintains a list of subscribers for each node/file.

2.5. ZK: Architecture

Zookeeper cannot function on a single machine alone. How does this functionality extend across multiple machines?
The current issue lies in the fact that Zookeeper operates on a single machine, which consequently creates a single point of failure. Therefore, Zookeeper is designed to function across multiple machines, typically

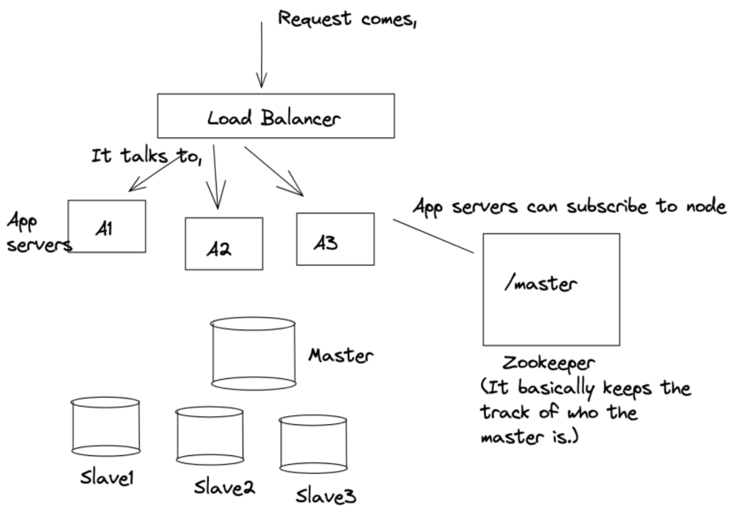


Figure 5: ZK Architecture

an odd number of them. Within the Zookeeper cluster, machines collectively elect a leader or master from among themselves. When setting up the cluster or in the event of the existing leader's failure, the first task is to elect a new leader. Suppose Z3 is elected as the leader; any write operation, such as modifying the /master IP address to a new value, is initially directed to the leader. The leader then broadcasts this change to all other machines. For a write operation to be considered successful, at least the majority of machines, including the leader, must acknowledge the change; otherwise, it is rolled back. For instance, in a cluster of 5 machines, 3 machines need to acknowledge for the write to succeed.

Even if a machine in the cluster fails, the total number of machines remains unchanged, and therefore, even in such cases, 3 machines need to acknowledge. Thus, if 10 machines were vying to become the master and simultaneously sent requests to write to /clusterx/master, all these requests would initially be directed to a single machine—the leader. The leader can employ a lock mechanism to ensure that only one of these requests proceeds initially. The data is written only if the majority of Zookeeper machines acknowledge the request. If not, the data is rolled back, the lock is released, and the next request is pro-cessed. But why rely on the majority of machines? Consider if we allow the write operation to succeed if it is acknowledged by at least X/2 number of machines (where X represents the total number of ma-chines). For instance, let's envision a scenario where we have 5 Zookeeper machines. Due to a network partition, machines z1 and z2 become disconnected from the other 3 machines. Let's consider the scenario where write1 (/clusterx/master_ip = ip1) occurs on machines z1 and z2, while another writes, write2 (/clusterx/master_ip = ip2), happens for the same Zookeeper (ZK) node on machines z4 and z5. When attempting to read (/clusterx/master_ip), half of the machines would indicate that ip1 is the master, while the other half would suggest ip2 as the master. This situation is commonly referred to as split brain. Thus, the implementation of Quorum or Majority is essential to avoid the occurrence of two separate sets of machines asserting different values as the answer. Consistency is crucial in such cases. Therefore, until the write operation is successful on the majority of the machines, we cannot confirm success. In the described scenario, both ip1 and ip2 attempt to write to Z3, and whichever operation succeeds will determine the master address, while the other operation will fail.

2.6. Master Dies

Consider a scenario where the master has written its IP address to /clusterx/master_ip. All app servers and slaves have set a watch on the same node to track the current master IP address. Now, let's envision what happens if the master dies: The master machine ceases sending heartbeats to Zookeeper for the ephemeral node /clusterx/master_ip. Consequently, the ephemeral node /clusterx/master_ip is deleted.

All subscribers are notified of this change. Slaves, upon receiving this update, initiate attempts to become masters again. The first to successfully write to Zookeeper assumes the role of the new master.

App servers delete the local value of master_ip. They must then read from Zookeeper, set a new watch, and update the local master_ip value whenever a new write request occurs. If they receive a null value, the request fails, indicating that a new master has not yet been selected. When the old master comes back online, it reads from the same Zookeeper node to determine the new master machine and assumes the role of a slave.

Unless it returns quickly and finds the Zookeeper node to be null, it joins other slaves in attempting to become the new master.

3. Async Tasks

To address such scenarios where asynchronous tasks, need to be handled, we utilize a concept called Per-sistent Queue. A Persistent Queue ensures durability, implying that data is written onto a hard disk to prevent any risk of data loss. Persistent Queues work on a model called pub-sub (Publish Subscribe) whereas zookeeper is used within Kafka cluster to manage Kafka cluster.

3.1. Pub Sub

Pub sub comprises two components: Publish: In this part, all events of interest that necessitate subsequent actions are identified. For instance, sending a message constitutes an event, as does a customer purchasing an item on Flipkart. These events are published onto a persistent queue.

Subscriber: Various events may attract different types of subscribers interested in those events. Subscrib-ers consume the events they have subscribed to from the queue. For example, in the scenario of a message notification system, a message email system, and a message analytics system would subscribe to the event of "a message sent" on the queue. Similarly, an invoice generation system might subscribe to the event of "a purchase made on Flip Kart." There can be multiple types of events being published, and each event may have multiple types of subscribers consuming these events. Topics: Within a queue, segregation of topics is necessary because the system does not want to subscribe to the entire queue; rather, it needs to subscribe to specific types of events. Each of these specific events is referred to as a topic.

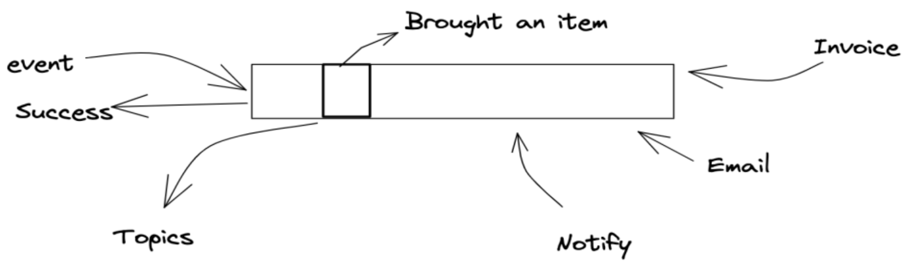


Figure 5: Topic

Consider an example involving Flipkart that Flipkart incorporates an integrated messaging service, al-lowing customers to communicate with vendors regarding product quality and feedback.

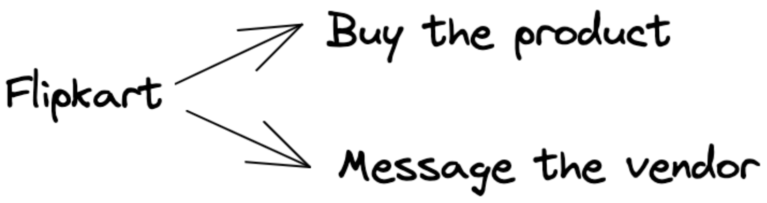
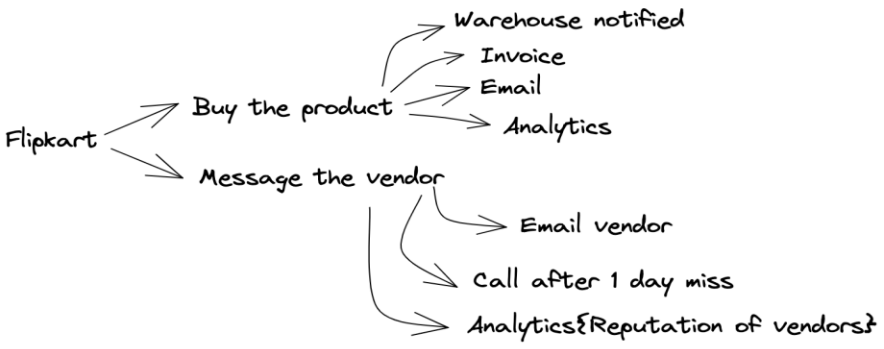
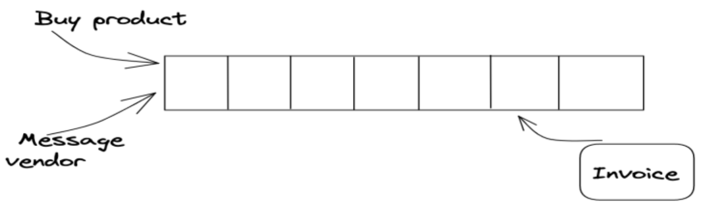


Figure 6: Publish Event

These are the two events, and subsequently, we desire specific actions to occur.



Here, both events are distinct from each other. If we were to publish both events in a single persistent queue, and suppose the invoice generation system has subscribed to the queue, it would receive a consid-erable amount of irrelevant information.



Therefore, we assert that not all events are equal, leading us to classify them into different topics.

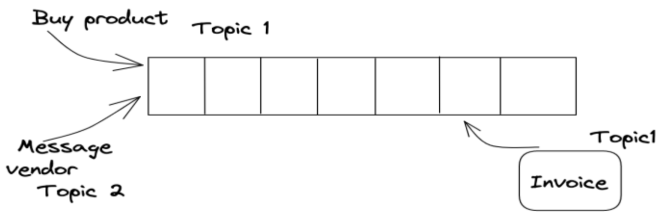


Figure 7: Publish Event on Different Topics

Now, the invoice generation system has exclusively subscribed to Topic1, ensuring it only receives mes-sages relevant to that topic. One prominent high-throughput system that incorporates persistent queues and supports topics is Kafka. In essence, persistent queues assist in managing systems where producers and consumers operate at varying rates, asynchronously. These queues provide assurance against event loss within a specified retention period and enable consumers to work asynchronously without impeding the producers' primary tasks.

3.2. Apache Kafka

Terminologies: Publisher is the systems responsible for publishing events to a topic are referred to as publishers. There may be multiple publishers. Subscriber Systems that consume events from subscribed topics (/topics) are known as subscribers. Every machine within the Kafka cluster is termed a broker. This term is simply a sophisticated designation for machines storing published events for a topic. Partition: Within a single topic, it is possible to configure multiple partitions.

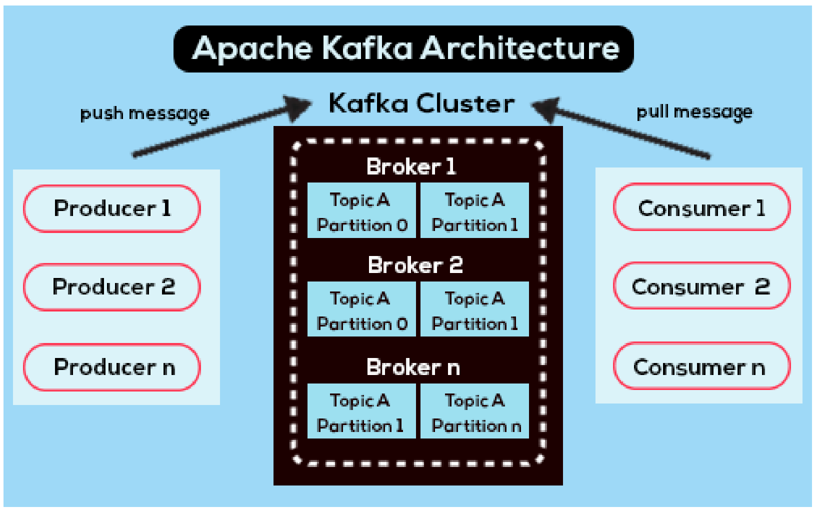


Figure 8: Apache kafka Architecture

The utilization of multiple partitions allows Kafka to effectively shard or distribute load internally. Additionally, it aids consumers in achieving faster consumption rates.

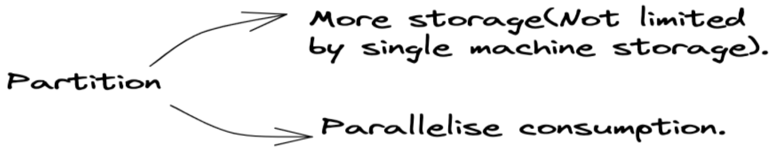


Figure 9: Partition

Event Retention Period: Kafka, like any persistent queue, is designed to store events temporarily rather than indefinitely. Therefore, you define an event retention period, which denotes the duration for which an event is retained. Events older than the retention period are periodically cleaned up. What if a topic becomes so large, with numerous producers contributing to it, that the entire topic, even within the reten-tion period, cannot fit in a single machine? How can your shard it? Kafka enables you to specify the number of partitions for each topic. While a single partition cannot be divided across machines, different partitions can reside on different machines. Increasing the number of partitions allows Kafka to distribute topic + partition assignments across various machines internally. With multiple partitions, the structure no longer resembles a simple queue. Ensuring the ordering of messages between partitions becomes chal-lenging.

For instance, in a topic where messages m1, m2, m3, m4, and m5 are assigned to different partitions, consumers lack a mechanism to determine the next most recent message across partitions. Introducing mechanisms to maintain message ordering between partitions incurs additional overhead and may not be conducive to high throughput. In many cases, strict message ordering may not even be necessary. Con-sider the scenario of a messaging system like Flipkart, where messages from different users to vendors are published on a topic named Messages. While the order of messages between different users may not be crucial, maintaining the order of messages from the same user is essential for coherence. Kafka offers a solution to ensure that all messages from the same user are routed to the same partition. Producers can optionally specify a key along with the message, and Kafka employs a hash function (hash(key) % num_partitions) to determine the partition to which the message should be sent. Messages with the same key are directed to the same partition, ensuring that messages from the same sender are grouped together. This simplifies the task of maintaining message ordering within the same partition.

3.3. Consumer Group

What if a topic becomes exceedingly large, making it impractical for a single consumer to process all events efficiently? What is the solution in such a scenario? In such cases, the only viable approach is to employ multiple consumers operating in parallel, each handling a different subset of events. Kafka facili-tates this through consumer groups. A consumer group comprises a set of consumer machines, all-consuming from the same topic. Internally, each consumer within the consumer group is assigned ex-clusively to one or more partitions (rendering it redundant to have more consumer machines than parti-tions). Consequently, each consumer exclusively receives messages from the partitions it is tagged to. This enables the concurrent processing of events from topics across multiple consumers within the same consumer group.

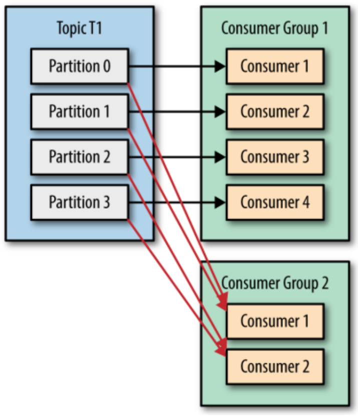


Figure 10: Consumer Group

In the event of one or more machines (brokers) failing within Kafka, how can it guarantee that events are never lost? The solution remains consistent with other scenarios is replication. Kafka allows you to con-figure the desired number of replicas. Subsequently, for each partition, primary and secondary replicas are allocated across machines or brokers.

Acknowledgements

We have partnered with strategic allies whose invaluable contributions have played a crucial role in iden-tifying highly available and low-latency solutions within the market, including Zookeeper and Kafka. Their substantial involvement extends to the implementation of various asynchronous task solutions across a spectrum of enterprise applications throughout the years [1-9].

References

1. Garg, N. (2013). *Apache kafka* (pp. 30-31). Birmingham, UK: Packt Publishing.
2. Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., ... & Stein, J. (2015). Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment, 8*(12), 1654-1655.
3. D'silva, G. M., Khan, A., & Bari, S. (2017, May). Real-time processing of IoT events with historic data using Apache Kafka and Apache Spark with dashing framework. In *2017 2nd IEEE International conference on recent trends in electronics, information & communication technology (RTEICT)* (pp. 1804-1809). IEEE.
4. Narkhede, N., Shapira, G., & Palino, T. (2017). *Kafka: the definitive guide: real-time data and stream processing at scale.* " O'Reilly Media, Inc.".
5. Le Noac'H, P., Costan, A., & Bougé, L. (2017, December). A performance evaluation of Apache Kafka in support of big data streaming applications. In *2017 IEEE International Conference on Big Data (Big Data)* (pp. 4803-4806). IEEE.
6. Hiraman, B. R. (2018, August). A study of apache kafka in big data stream processing. In *2018 International Con-ference on Information, Communication, Engineering and Technology (ICICET)* (pp. 1-3). IEEE.
7. Thein, K. M. M. (2014). Apache kafka: Next generation distributed messaging system. *International Journal of Sci-entific Engineering and Technology Research, 3*(47), 9478-9483.
8. Shivakumar, S. K., & Sethii, S. (2019). *Building digital experience platforms: A guide to developing next-generation enterprise applications*. APress.
9. Sethi, S. (2023). Platforms Based Approach and Strategy for Fintech applications. *Authorea Preprints.*