An Analysis of the Implementation of Kafka in High-Frequency Electronic Trading Environments

Vlad Bucur, Ovidiu Stan, Liviu Miclea

***Abstract*—Electronic trading amounts for the vast majority of all financial transactions with bonds and equities in the world. This type of trading is based largely on the brokering of messages from the buy side, brokers or financial institutions, to a sell side (usually an exchange). Since not all trading is done at the same time these messaging systems need to account for server down times, sequencing, high throughput and performance requirements. Therefore, most fintech companies employ queuing mechanism to manage the message flow. This paper analyzes the way in which Kafka, one of the premiere messaging systems currently in use, can simplify various systems in use such as message brokering, persistence or fault tolerance. The authors of this paper hope to demonstrate the use of Kafka as a messaging system, backup solution and alert broker for software operators and developers with low fault-tolerance and rigid up-time requirements.**

***Index Terms*—Kafka, message queue, electronic trading, message broker.**

I. Introduction

The rise of electronic trading in stock markets is believed to have started in 1971, when NASDAQ, the largest stock exchange in the world, was founded. NASDAQ was founded as a computerized trading platform and thus was moving away from other markets where stocks were traded manually through a stockbroker via an open outcry [1]. This is not to say that there weren't other attempts before 1971 to automate stock trading but it is largely believed that the market truly reached maturity in 1971. At first, progress was slow for electronic trading as it took another 14 years before the next major step was reached, the introduction, in 1985, of a retail trading platform by Trade*Plus on AOL and Compuserve terminals [1]. Another 9 years passed before the next major step in the evolution of stock markets happened, when "K. Aufhause & Co. became the first brokerage firm to offer online trading via its WealthWeb" [1]. Despite its slow start, from the moment when brokers realized the power of the internet and interconnected computers the writing was quickly on the wall for traditional trading solutions. In only a few years multiple US companies started offering networked services for small traders and, by the late 1990s, the popularity of electronic trading had already become a global phenomenon, not only a US one.

Currently, in a report published by the Bank of International Settlements as early as 2016, electronic trading has become the dominant method for several fixed income markets [2]. According to the study, which reflected numbers from 2016, 90% of all futures trading was done electronically. In foreign currency exchanges (FX) and equities 80% of trading was done electronically. Even government and other covered bonds, a safer investment instrument with a lower risk that is advertised using traditional media such as TV or radio, 50% of all trading was done electronically. These numbers reflected the global market as they were the result of polling over 30 trading platform providers [2]. Not only is trading done electronically, but it is overwhelmingly done automatically. In a paper published in 2013, which later became the go-to-reference for studies on algorithmic trading for multiple publications, Morton Glantz & Robert Kissell concluded that, in 2012, 85% of volume of trades in global markets was done by algorithmic trading [3].

This flurry of market activity is enabled by message brokering systems that can handle throughput of millions of messages every minute and tens or hundreds of millions of messages daily. It's important to note that, unlike the popular belief that all trading is done nearly instantaneously, most messages can persist in a broker's, bank's and exchange's databases for days, weeks or even months. Trades aren't solved instantaneously, and multiple paper trails must be kept for audit reasons, or to accommodate multiple brokers, working from different countries on different timelines with the same order. This is one of the reasons why the number of messages send and received on a daily basis reaches such staggering values. And as a result of these needs, most traders have resorted to using different types of messaging systems and queues to keep order in their books. Stock exchanges allow customers to interact via XML, JSON or FIX Messaging standards and therefore might need to accommodate various possible responses to an execution report, a type of acknowledgment that an order or trade has been processed. Companies themselves need to be able to queue messages overnight, from brokers that might need to make changes to an order that will only go into effect the next morning and they need to ensure that the flow of messages to the exchange is not interrupted or lost lest they risk losing money.

As a result, many companies have chosen to implement third party messaging services such as ActiveMQ, RabbitMQ, JMS and, for the purpose of this paper, Kafka. Kafka started development in 2008 at LinkedIN, where three programmers wanted to solve problems when dealing with streams of data [4]. It later became part of the Apache project and was open sourced becoming available to developers worldwide. Kafka's vision is to allow streaming from multiple connected platforms, all at once, including databases [4]. As a result, developers are able to connect and integrate streams and APIs, process them and build applications on top of them [4]. This has become particularly useful to companies that need to

Manuscript received October 9, 2019; revised March 10, 2020.

Vlad Bucur, Ovidiu Stan, and Liviu Miclea are with theDepartment of Automation, Technical University of Cluj Napoca, Cluj, Romania (e-mail: Ovidiu.stan@aut.utcluj.ro).

DOI: 10.7763/IJMO.2020.V10.746

communicate between applications in several different ways, both internally and externally.

This paper will focus on how integrating Kafka can help with managing time and data-loss sensitive streams from a financial markets' perspective. It will be structured in four sections, beyond the introductory part: a brief explanation of how Kafka works, both in general terms and in the context of this paper, a section analyzing and describing the implementation of Kafka for multiple tasks in message-driven electronic trading systems, a succinct comparison between Kafka and other messaging queues and finally a conclusion which will also highlight some research challenges with Kafka implementations and general message management systems.

II. The Inner Workings of a Kafka Cluster

Kafka primarily is a "distributed streaming platform" [5]. It allows developers to subscribe and publish streams, store them and process them. In essence, when initially developed, Kafka was engineered as a pure messaging queue system, but was later modified to be based on an abstraction of a commit log [6]. Kafka still works as a cluster of servers, despite this not being a requirement to using it as Kafka can delegate this clustering duty to Kubernetes. However, when run by itself, the Kafka cluster is formed out of at least one broker, or a "physical" instance of Kafka, at least one topic and at least one partition. It's important to note that topics and partitions do not need to be created explicitly by the developer, because if they do not exist, Kafka will take care of their creation automatically.

Kafka uses topics, as showed in Fig.1, to manage the messages it receives from producer threads. A topic is a basic messaging queue that is, however, split up in at least one partition. Messages split up in multiple partitions are not in order, however, due to load balancing concerns. In order for messages to be in order developers need to generate a specific key for the message, using an internal serializer tool provided by Kafka. When messages are assigned to partitions, they are also assigned and index number, known as an offset in Kafka terminology. The offset is the position of a message in a partition from which a consumer thread will begin reading a topic. Developers can determine whether the consumer threads read topics from the beginning, or index 0, or from another offset number, based on what their needs are. Deleting messages from Kafka topics is a complicated affair and is arguably one of the weak points of working with Kafka, which is why it's important to note that the messaging system uses offset numbers to purge topics on a user determined schedule (by default: one week).

Beyond simply acting as a messaging system Kafka offers load balancing and failover solutions for managing cases of force majeure. If multiple brokers are available in a cluster the system will distribute partitions separately to each broker. If, for example, a cluster is formed of three brokers and a topic is formed of three partitions, each broker in the cluster will host one partition for load balancing purposes. Kafka also enables users to replicate data once or twice. The official documentation recommends that data be replicated twice, and in such a scenario each of the three brokers mentioned above would hold a copy of a distinct partition from their original load-balancing assigned partition. Should one broker encounter issue the consumer threads could reorient themselves to one of the remaining two brokers which would have an in-sync copy (ISC) of the partition they are reading from.

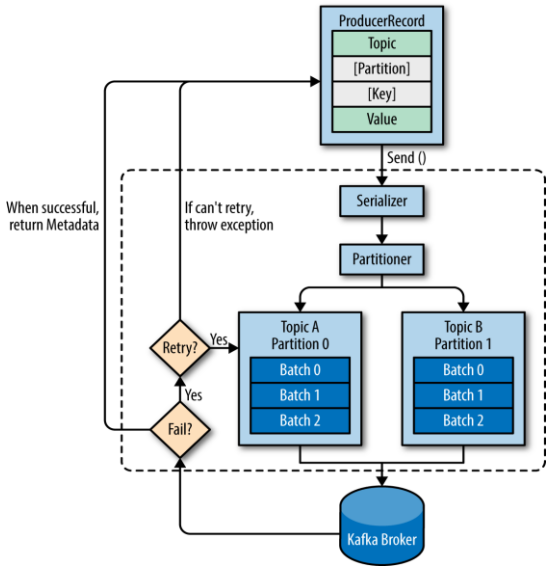


Fig. 1. Kafka standalone cluster.

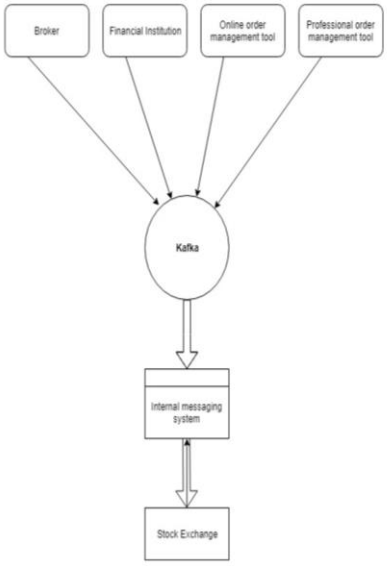


Fig. 2. Basic architecture of an electronic trading system using Kafka.

III. Implementation of Kafka in Electronic Trading Systems

One of the most common ways to use Kafka in any type of server application is to synchronize and normalize input from various sources. In the case of electronic trading mechanisms in particular this means that several buy-side (brokerage firms, financial institutions, etc.) terminals are linked to a Kafka cluster or topic and their messages are threaded through to a sell-side (stock exchange).

In Fig. 2 a basic software architecture of an electronic trading system using Kafka is shown. The architecture follows the basic design patterns of a messaging queue cluster with a few notable exceptions. In the above image the processing and storage of data is left to the internal messaging system entirely. Here, instead of directly

connecting to a database using DB connectors provided by Kafka the tasks of storing messages, sequence numbers and replication are left to the internal messaging system.

This helps companies to improve performance and cyber-security. Firstly, by letting the internal messaging system handle all data storage the data can be stored in flat file databases on the client's side. This results in greatly improved performance as the part of the architecture that handles the processing of messages does not need to connect to a remote server but rather does all the processing on a local hard drive. Secondly, the data replication can be done through an internal network, ensuring that, in addition to the data replication in Kafka topics, a hard copy of all transactions is kept on a local hard drive in a different part of the network without requiring access to the internet to achieve, only a functioning local network. Thirdly, this method ensures that business and customer sensitive data is kept only on company computers until it is disseminated back to the broker or financial institution that started the order.

Additionally, when using an internal messaging system to manage Kafka topics, the communication between counterparty, Kafka and end customer is easier to achieve. The internal messaging system will convert all messages sent by Kafka to a common format agreed upon by the stock exchange. This means both outbound and inbound messages are handled in the same language and conversion only needs to be done one way, from Kafka to the messaging standard and back. Perhaps the question then arises, where in all of this does Kafka help simplify the functionality of an electronic trading system if the internal messaging system handles the conversion of messages? In fact, in this particular case, before the advent of Kafka, the messaging system would've been required to handle either multiple types of strings (JSON, plain String, char arrays, POJOs, etc.) whereas now, while using Kafka, the conversion can be done with the provided Kafka classes. In the code snippet below (Fig. 3), Kafka is able to use its consumer functionality to receive all types of messages from multiple sources, convert the JSON to a specific type of object intended for use with the internal messaging system and then send these messages for processing to the messaging system itself.

It's extremely important to underline that the ability to handle all types of input at a cluster level, before the message reaches and internal messaging system is a massive advantage offered by Kafka to electronic trading systems that other messaging queue simply do not offer. Since the processing of messages can be done independently, by the API, there is no need to accommodate for a special case for each queue that is employed. Before the advent of Kafka most internal messaging systems used by banks and financial institutions (for example FIX messaging systems) had a separate setting that needed to be enabled for each type of queue. Queues were not compatible with each other all the time and, if a new queue was set up, the trading cluster needed to be taken down, the new queue added to the configuration file, tested and only then the server could be booted back up. Kafka solves this issue because it processes all types of messages through its topics and the entire topic can be read by a middleware that stands between the internal messaging system and the Kafka cluster. The problem of who is sending what type of message is a non-issue at this point and the developer only needs to focus on interpreting the strings and sending them to the FIX engine. No matter how many types of messages in different formats changes are made only in code and they do not affect the immediate functionality of the trading cluster.

```
Public void run(){
    while (RUN) {
        try {
            ConsumerRecords<String, Supplier<String>>
                records = consumer.poll(100);

            for (ConsumerRecords<String, Supplier<String>>
                record : records){

                    String strValue = String.valueOf(record.value());

                    Action action = (Action) Util.convertJSON2Obj (strValue,
                        Action.class);

                    if ("send-message".equalsIgnoreCase (action.getType())){
                        String[] cmds = action.getCmd().split(">");
                        send2IMS(cmds[0], cmds[1], cmds[2]);
                    }
                }

        }catch(Exception ex){
            ex.printStackTrace();
        }finally{
            consumer.commitSync();
        }
    }

    consumenr.close();
}
```

Fig. 3. Implementation of a Kafka consumer thread with JSON-based streams.

Beyond just managing the flow of inbound and outbound messages one of the most important aspects of a messaging queue system, especially from a high-frequency trading perspective, is to ensure persistence. Persistence refers to the process of storing an object and its characteristics after the process which created that object has been completed [7]. While persistence often means redundant data [8], and this is certainly the case in electronic trading systems, huge amounts of data are persisted daily in trading systems for various reasons. One of the most important reasons is due to performance as persisted data is used to throttle trading systems before the start of the trading day. Additionally, smaller persistence methods are required to carry specific information, or tags, from one different message to another, since not all brokers and financial institutions send the same amount of information in their messages.

Then, persistence is of three broad types: at startup, scheduled and always. To implement a persistence of this type using Kafka one would need to create three different consumer threads at least, one to handle each of the types of persistence. To know which messages, need to be persisted, Kafka uses the offsets of each queue to determine from where it needs to start sending messages to the cache. For data that is needed at startup Kafka would send all messages from the earliest offset to an internal cache and store them there for when the internal messaging systems needs to reference them. When data is required to be accessed on a schedule, for example every five minutes, Kafka would set the offset to the number that was last sent and then remove the remaining objects for the active map. Finally, when data needs to be persisted constantly, the latest offset would simply be sent to the active cache.

This functionality basically bypasses the traditional database approach somewhat, as Kafka uses its topics to store data and then send it directly to a cache instead of a database which is then, usually, loaded into memory and is only then cached. On an implementation level, topics are created and

deleted as needed and associated to a producer thread. Once the topic has been consumed it is deleted. When involved with a physical database, methods are created separately for handling all table management and concurrency situations such as in a shared file scenario, if using a flat-file database. The cache itself can be implemented in Java as an object using streams to receive and distribute data.

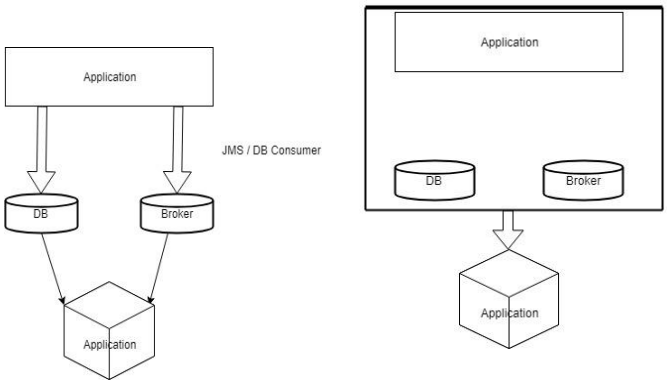
Finally, one of the last aspects which needs to be considered when managing a security and performance critical messaging system is high availability, load and replication of messages. Kafka offers a guarantee that, "a topic with a replication factor N, will tolerate N-1 server failures without losing any records committed to log" [5]. This is the basic replication mechanism of a Kafka cluster and it can be enabled remotely in the cloud every time Kafka is deployed. As a result, one of the most common implementations of Kafka in a high-availability low fault-tolerance system is to deploy multiple instance of the internal messaging system and the Kafka cluster with Docker and Kubernetes. An even easier solution involves a third-party tool, Strimzi, which provides developers a way of running a Kafka cluster on Kubernetes in various deployment configurations [9].

However, the automated deployment approach is not the only way in which Kafka can be used for load balancing and high-availability. A bespoke solution using middleware between the cluster and the internal messaging system can also be employed. In this case messages from multiple messaging systems are sent to one topic and then from that topic they are distributed to at least two physical machines, using in-sync replication for each machine, so that, if one machine fails the other one can take up the status of leader and continue sending messages to the counterparty.

IV. A Comparison of Kafka and other Messaging Systems

Before attempting to compare Kafka to other messaging systems it's important to note that Kafka isn't only a messaging system. In fact, according to the developer's website, messaging is only one of the features of Kafka, along with storing data and processing streams [5].

One of the ways in which it's immediately apparent that Kafka differs from other messaging systems is in the way in which it uses queues. In a traditional queueing system queues are not multiple-subscriber and once a process reads the data, the data is gone [5]. In Kafka, the consumer group process allows developers to divide up processing to a collection of processes [5], as was the case with the high-availability scenario described in section III. Furthermore, in a traditional queueing system records retain their order and if "multiple consumers consume from the queue the server will hand out records in the order in which they are stored" [5]. The partition inside Kafka topics offers load balancing and ordering guarantees by assigning each partition in a topic to a consumer in the consumer group so that each partition is used by exactly one consumer at a time [5]. Since data in partitions is stored in order, that guarantees that each consumer consumes the data in order.



(a) (b)

Fig. 4. Comparision of queue implamentation. (a) ActiveMQ implementation of a queue; (b) Kafka implementation of a queue.

Another major difference between Kafka and other messaging queues, most notably ActiveMQ (also an Apache project) is the way in which the architecture of the two systems was developed (Figure 4). In the case of ActiveMQ it uses JMS [10], the provided Java API for interfacing with messaging brokers. This obviously makes it quicker in terms of performance as the JMS API is designed to be a lightweight API that wants to offer reliable ways of brokering messages to clients. As a result, JMS type brokers, such as ActiveMQ are not really considered universal data pipelines [10]. It's a completely different model from Kafka, which is solely concerned with becoming a universal data pipeline and has followed that design philosophy throughout its development. The concept of topics in themselves was designed to handle not one type of message broker interface, but multiple. Therefore, the major difference is that, in the end, almost all types of messaging queues require an individual implementation of their message broker interface, somewhere between the application and the final output destination, whereas Kafka does not.

Finally, it's important to note that Kafka isn't just a queueing system. While this obviously would invalidate the comparison with other messaging systems, it should still be noted that Kafka can be used in multiple ways, including, as a database, for persistence purposes as a stream management tool or as a storage system. However, it should be noted that this might come to a cost in terms of performance, particularly when using Kafka as a messaging system in high-performance environments. Kafka's message acknowledgement system provides three levels of accuracy:

- **acks = 0**: in this case the producer does not wait for an acknowledgment from the server and no guarantee can be made that the server has received the message [10]. This is one of the riskiest approaches to managing a messaging queue and it doesn't carry any guarantees from Kafka
- **acks = 1**: in this case the leader writes the record to its local log but will respond without waiting for full acknowledgment [10]. This is usually the preferred method of acknowledging a message was received successfully as it doesn't involve waiting for the server to respond
- **acks = all**: the final case, in which the leader waits for **all** servers to respond that they have acknowledged

the record [10]. This includes all in-sync replicas on every Kafka broker.

None of the three solutions above are nearly as safe, quick or elegant as using a regular JMS queue which will return the **acks** from the server nearly instantaneously. And, in high-performance environments even using the **acks = 1** solution, which is the only intermediate solution that offers a guarantee that messages were received, still drags performance down considerably as Kafka waits for confirmation from the leader.

V. Conclusions

Even though Kafka could be considered an established product in most software development cycles knowledge of its inner workings and concrete implementations using it as a main message broker or storage solution are still quite sparse. In an article from February 2019, of most in demand skills for programmers Kafka came in second, only losing out to GoLang a new object-oriented language from RedHat Enterprises which also has a stake in Kafka [10].

As a result of the high demand for Kafka skills in the workplace we, the authors of the paper, postulate that one of the major reasons for this high demand is a lack of knowledge related to the way Kafka operates and what exactly it might be good for, let alone how to use it properly. The reason why Kafka became more popular was not due to, perhaps, its intended use as a message broker for big data but rather as a tool for managing disparate messaging systems in non-monolithic applications – micro-services in the cloud. We base our theory on the fact that, throughout researching this paper, most of the references to other scientific articles that we found were based on high-throughput, big data systems using Kafka as a message broker or storage solution.

However, that should not dissuade developers from using Kafka nor should it be an indictment of the product itself. We believe that Kafka is a powerful tool which has yet to be used to its full extent in non-cloud environments, especially in those environments where high-availability and low fault-tolerance are critical to success. And while, it can be argued, that performance wise JSM or API based broker interfaces are quicker than Kafka we cannot ignore that the world of software development is moving to a distributed solution where implementing a different API for each solution is simply becoming an impossible task, wasting too much time and resources which are already at a premium in the industry.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

BV conducted the research, algorithm and wrote the paper; SO analyzed the experimental stage and the English grammar; ML assured the experimental work. All authors had approved the final version.

Acknowledgment

The research presented in this paper was supported by the following projects: ROBIN (PN-III-P1-1.2-PCCDI-2017-0734) and SeMed (2933/55/GNaC 2018).

References

[1] V. *Market, Multi-asset Risk Modeling: Techniques for a Global Economy in an Electronic and Algorithmic Trading Era*, Academic Press, 2013, pp. 258.
[2] Anadiotis, Geroge. (2017). Kafka: The story so far. [Online]. Available: https://www.zdnet.com/article/kafka-the-story-so-far/
[3] A. Kafka. (2019). Introduction. [Online]. Available: https://kafka.apache.org/intro
[4] Confluent. (2019). What is apache kafka? [Online]. Available: https://www.confluent.io/what-is-apache-kafka/
[5] B. Stephanie, "Contracted persistent object programming," Swiss Federal *Institute* of Technology Zurich, 2015.
[6] W. Thorsten and K. Veit, "Persistence in data warehousing," in *Porc. 2012 Sixth International Conference on Research Challenges in Information Science (RCIS)*, 2012.
[7] Strimzi. (2019). Overview. [Online]. Available: https://strimzi.io/
[8] T. Liam. (2019). [Online]. Available: Best-paying programming languages, skills: Here are the top earners. https://www.zdnet.com/article/best-paying-programming-languages-skills-here-are-the-top-earners/
[9] A. Kafka. (2019). Documentation. [Online]. Available https://kafka.apache.org/20/documentation.html
[10] ActiveMQ. (2019). Hello World. [Online]. Available https://activemq.apache.org/hello-world

Bucur V. is a PhD student in the Automation Department at the Technical University of Cluj-Napoca. His research interests include advanced Internet technologies, dependable systems, cloud computing.

Stan O. is a lecturer in the Automation Department at the Technical University of Cluj-Napoca. His research interests include medical informatics, semantic interoperability, information management in the age of the Internet, dependability and fault-tolerant systems. Stan received a PhD in systems engineering from the Technical University of Cluj-Napoca. He is a member of IEEE.

Miclea L. is a full professor the Automation Department at the Technical University of Cluj-Napoca. He is also the dean of the same faculty. He is the author or co-author of 17 books, 40 research works and more than 180 scientific publications. His research interests include are dependability, cyber-physical-systems, agent systems. Miclea is a Senior member of IEEE and is regular the general chairman of the IEEE-CS-TTTC-AQTR conference.