TRAK: A Testing Tool for Studying the Reliability of Data Delivery in Apache Kafka

Conference Paper · October 2019

DOI: 10.1109/ISSREW.2019.00101

3 authors:

Han Wu
University of Birmingham
24 PUBLICATIONS **201** CITATIONS
SEE PROFILE

Shang Zhihao
Zhengzhou University
31 PUBLICATIONS **444** CITATIONS
SEE PROFILE

Katinka Wolter
Freie Universität Berlin
171 PUBLICATIONS **2,625** CITATIONS
SEE PROFILE

TRAK: A Testing Tool for Studying the Reliability of Data Delivery in Apache Kafka

Han Wu, Zhihao Shang, Katinka Wolter
Institut für Informatik
Freie Universität Berlin
Berlin, Germany
Email: {han.wu, zhihao.shang, katinka.wolter}@fu-berlin.de

***Abstract*—In modern applications the demand for real-time processing of high-volume data streams is growing. Common application scenarios include market feed processing and electronic trading, maintenance of IoT devices and fraud detection. In some scenarios reliability is the utmost concern while in others speed and simplicity are the top priority. Apache Kafka is a high-throughput distributed messaging system and its reliable stream delivery capability makes it an ideal source of data for stream-processing systems. With various configurable parameters Kafka is very flexible in reliable data delivery thus allowing all kinds of reliability tradeoffs. In this paper we introduce a tool for Testing the Reliability of Apache Kafka (TRAK), to study different data delivery semantics in Kafka and compare their reliability under poor network quality. We build a Kafka testbed using Docker containers and use a network emulation tool to control the network delay and loss. Two metrics, message loss rate and duplicate rate, are used in our experiments to evaluate the reliability of data delivery in Kafka. The experimental results show that under high network delay the size of messages matters. The at-least-once semantics is more reliable than at-most-once in a network with high packet loss, but can lead to duplicated messages.**

***Index Terms*—Stream processing, Reliability, Apache Kafka, Docker**

I. Introduction

Apache Kafka is a distributed messaging system which has seen wide adoption in modern stream-based applications [1]. It is often built as the reliable data source for stream-processing systems such as Apache Storm, Apache Spark Streaming and Apache Flink [2]. The use cases of these applications range from tracking clicks in a website, network and infrastructure monitoring, to electronic financial trading and customer service for online reservation. The requirements of Kafka's reliable data delivery differ among those use cases. For instance, an application that collects streams of web page logs to count views per web page can tolerate some inaccurate processing. In this situation quick response from the application is prioritized over reliability. However, for the streams of debit and credit card payments, reliability is the top priority and there is no tolerance for errors in processing. Specifically, every stream of data should be processed exactly once without exception.

To study the impact of server and network failures on the delivery probability and system performance, we create a testing tool TRAK. With this tool we can customize a Kafka cluster with deployment configurations including the number of brokers and network settings. We can create Kafka topics by user-defined parameters such as the number of partitions and replications. TRAK generates message loads with Kafka producers and uses Kafka consumers to receive messages. The size of messages is configurable and the value of each message contains a header for the result analysis. The tool provides a fault injection environment where the network delay and packet loss can be controlled. We evaluate the reliability of Kafka by sending a number of messages from the producer to the consumer. TRAK counts and analyzes the lost and duplicated messages. The outputs of the tool are two metrics, the message loss rate and the duplicate rate.

The paper is organized as follows. In Section III we introduce the mechanism of the semantics in Apache Kafka and their application scenarios. We describe the design of TRAK in Section IV. In Section V we define the fault we inject, and the metrics we evaluate. The experimental results are illustrate in Section VI.

II. Related Work

Numerous work exists on the comparison of Apache Kafka and traditional messaging systems [3]. Some results show that Kafka is more appropriate for scenarios that need to process massive amounts of messages, but if messages are important and security is a special concern, systems like AMQP are more reliable [4]. The studies on the reliability of publish/subscribe systems generally focus on fault tolerance strategies [5]. A general knowledge model is proposed in [6] for providing exactly-once message delivery in a content-based publish/subscribe system. Exactly-once delivery can also be guaranteed in another popular distributed stream-processing platform, Apache Flink [7].

We build the tool TRAK using Docker containers. Docker is a popular emerging virtualization technology on operating system level, which is similar to a lightweight virtual machine [8]. A Docker container is piece of a software that packages up all code, runtime, system library and dependencies, and runs fast and reliably when moved from one environment to another. Containers are less resource and time consuming, and rising as an important part of microservices and the cloud computing infrastructure [9]. For fault injection we use a network emulation tool for Docker, Pumba, which is used to

tune the delay and packet loss of the network between the Docker containers [10].

III. Delivery Guarantees in Kafka

As a messaging system Apache Kafka processes messages in a publish/subscribe manner. A Kafka producer publishes messages to a topic, and a Kafka consumer who is interested in this topic will subscribe to it and consume messages from it. Different categories of messages go to different topics. Every topic consists of multiple partitions, which are the physical storage of messages in the Kafka cluster. Partitions are distributed across Kafka servers, which are called brokers, and can be replicated for fault tolerance. Messages are written to partitions in an append-only manner, and within a partition a Kafka consumer will read messages in the order that they are appended. As illustrated in Fig.1, producer A sends a message x to one of the three partitions in Topic A, and consumer A reads message x from the partition by subscribing to Topic A. Generally the messages in the same topic are evenly distributed among all the partitions for load balance, therefore the producer sends messages to all partitions and the consumer consumes from all. We see that the partitions in Topic A are replicated in the Kafka cluster, e.g. Partition 1 is replicated to Broker 3 as Replica 1. For Topic B represented with a different color, the producer B and consumer B will send and receive messages in a similar FIFO way.

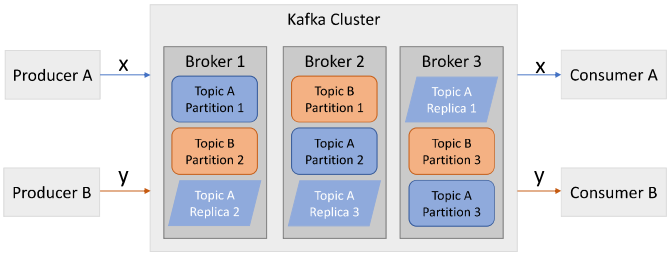

Fig. 1. Overview of Kafka

Normally the reliability of data delivery in Kafka can be guaranteed through three different delivery semantics. The at-most-once delivery semantics is the easiest to implement in Apache Kafka, because the producer does nothing other than sending each message once without waiting for any acknowledgement from the consumer. Thus there is no guarantee that a message is successfully delivered. This offers high throughput and makes good use of the available bandwidth, but there is no mechanism to avoid message loss.

We can tune the configurations to stipulate that a message is considered to be written successfully to Kafka only if the producer receives an acknowledgement (ack) from the broker. When a producer ack timeout expires or the producer receives an error, the producer can retry sending the message until the message arrives safely to the partition. This is the at-least-once delivery semantics in Kafka, which can lead to duplicated messages in the partition, as depicted in Fig. 2. The broker has successfully written message $m4$ to its partition (step 2), but failed before sending an ack back to the producer. Or just because of poor network quality the ack was not successfully delivered (step 3). Then the producer retries to send the message $m4$ (step 4) and this retry leads to the message being written twice to the partition (step 5). Thus the consumer will read the same message $m4$ twice which can cause duplicated work and incorrect results.

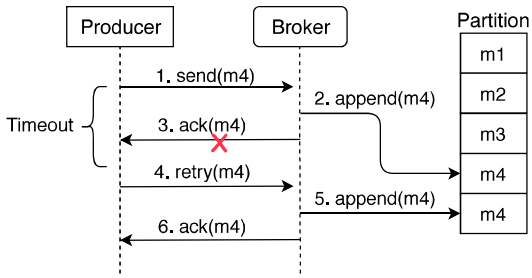

Fig. 2. At-least-once delivery in Kafka

Apache Kafka supports exactly-once delivery semantics in its latest version by introducing idempotent producer and transactional guarantees. An idempotent producer adds a sequence number to each batch of messages which will be used by the broker to detect any duplicated send. The sequence number is persisted to the replica of a partition and in case that a leader partition fails, the broker that takes over still knows if a message is duplicated. This guarantees that even though a producer retries upon failures every message will be persisted in the partition exactly once. However, the sequence number occupies extra numerical fields in each batch of messages which causes additional overhead, and the idempotence is guaranteed only within a single producer session.

Transactional messaging means that the producer sends a batch of messages as a single transaction and either all messages in the batch are visible to all consumers or none of them is. In order to achieve this Kafka introduces a transaction coordinator and a transaction log as depicted in Fig. 3. The coordinator is a module running inside every Kafka broker and it maintains the transaction log to manage the states of transactions.

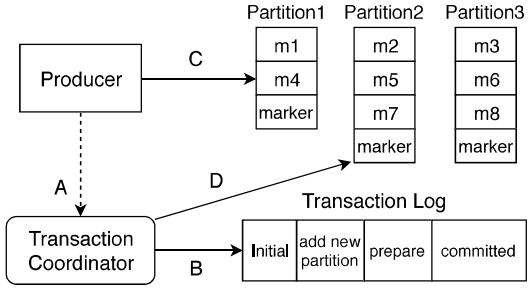

Fig. 3. Transactional messaging in Kafka

Before the producer sends messages to a partition, it will register the partition with the transaction coordinator first (step

A). The coordinator writes the state of each transaction to the transaction log and whenever the producer moves a step forward the coordinator will update the transaction's state (step B). The log stores only the states of a transaction and not the actual messages. As the messages are written to the partition (step C), the coordinator uses the two phase commit protocol to write transaction commit markers to the corresponding partition (step D). The marker is used by consumers to filter out messages from aborted transactions. Finally the coordinator updates the final state of the transaction and moves on to the next one.

IV. Testing Tool Design

After having described the delivery semantics in Apache Kafka we now need to evaluate the reliability and performance of these semantics in different application scenarios.

To study the reliability of data delivery in Apache Kafka we created the testing tool TRAK. With this tool users can create the Kafka cluster that they want to test. They can choose the number of brokers in the cluster, the number of partitions in topics and many other server properties. Each broker in the cluster is a running Docker container, which is started through the Docker compose tool. We design the Docker image of each broker and use the Docker compose tool to define and run multiple Docker containers. Users of TRAK can easily start, restart, scale up or down the Kafka cluster. In TRAK we use a bridge network to control the communications among containers as well as the DNS resolution of container names to IP addresses. Users can publish and subscribe to the topics in the Kafka cluster with producer and consumer client, which are additional containers connecting to the network. The producer client generates message loads with user-defined delivery semantics, message size and the total number of messages to be published. In the consumer client all received messages are stored in a list for reliability analysis. In our experiment we add an additional header to each message value and each header is a unique primary key of this message. Thus TRAK can generate reliability metrics by comparing the list of received messages headers and the list of sent ones. The fault injection in the testing tool includes high network delays, packet loss and broker failure. Users are able to choose the specific parameters of these faults such as packet loss rate and duration of the lossy period. In the next section we introduce the faults we inject and metrics we measure.

V. Fault Injection and Reliability Metrics

The failure scenarios can be divided to three types, the broker failure, the client failure and the network failure. The failure of a broker will trigger a new broker to take over its work including the newly elected leader partition receiving messages and the management of the transaction log. In TRAK we can reproduce this failure by aborting or restarting a running container which hosts the Kafka broker. Similarly, we can manipulate the Docker containers to reproduce the scenario of a producer or consumer client failure. Poor network is a common scenario when using Apache Kafka in the cloud. We use the open source tool Pumba to manage the emulations of network properties such as delay, packet loss and corruption. Pumba utilizes the built-in network emulation capabilities in Linux and can help simulate realistic network conditions as we build and run microservices in Docker containers. We inject two types of faults to the network of Kafka, high network delay and packet loss. Both faults can cause message loss in Kafka, and under at-least-once semantics, we can observe duplicated messages.

To evaluate the reliability of data delivery, we define two metrics, the loss rate and duplicate rate of messages, denoted α and β. In our experiments we add an additional header to each message value and each header is a unique primary key of this message. Assuming that the user intends to send N messages from the producer client, and finally the consumer client receives M messages with M' different headers. Thus we obtain the number of lost messages $L = N - M'$, as well as the number of duplicated messages $D = M - M'$. Therefore, if a message is duplicated four times, we count four. We define $\alpha = L/N$ and $\beta = D/N$.

VI. Experimental Results

We run TRAK on a PC machine equipped with Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz processor, 32 GB of RAM, and 2 TB disks. We use Docker version 18.09.7 and within the Docker image we use Kafka 2.2.1 with OpenJDK 11.0.3 on Ubuntu 18.04. In TRAK we create a Kafka cluster with three brokers. Then we run two containers as producer and consumer clients to send and receive messages, respectively. For each test we sent 500,000 messages from the producer implemented as load test which tool roughly 26 minutes. In the tests without any fault injection, the network is fast (less than 0.1 ms delay) without any packets lost, and no messages are lost or duplicated. In the test under fault injection, we inject high network delay and packet loss before we run the producer client to send messages. The network condition is unchanged until the sending process terminates. The network delay ranges from 1ms to 300ms and the packet loss from 1% to 10%. We ran each test at least 5 times to be able to compute an average result. Some abnormal results are discarded.

To implement at-most-once semantics, we set the number of acknowledgements required by the producer to zero, therefore the producer does not require any response from the server. Then we use Pumba to tune the delay of egress traffic from each container. Fig. 4a shows the impact of network delay on the message loss rates with different message sizes.

We observe that generally the loss rate is growing as the network delay increases. It is worth noting that when the network delay is less than 150 ms, messages of 100 bytes are much easier to be lost than the ones of 500 bytes. However, under higher network delay the loss rate of large messages (500 bytes) increases sharply to over 0.4%, while the loss rate of smaller messages remains almost stable around 0.1%. This result indicates that if the user of Kafka uses at-most-once delivery semantics, there is no guarantee of successful message

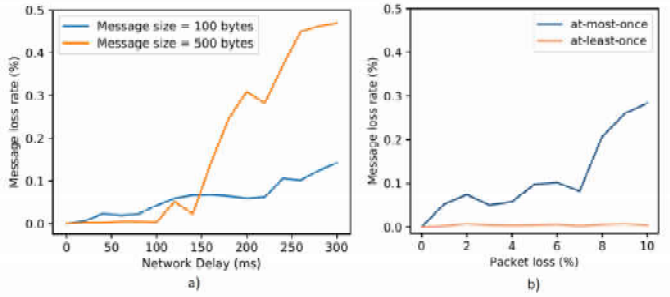

Fig. 4. a) Message loss rate for increasing network delay with at-most-once semantics; b) Message loss rate for increasing network packet loss rate

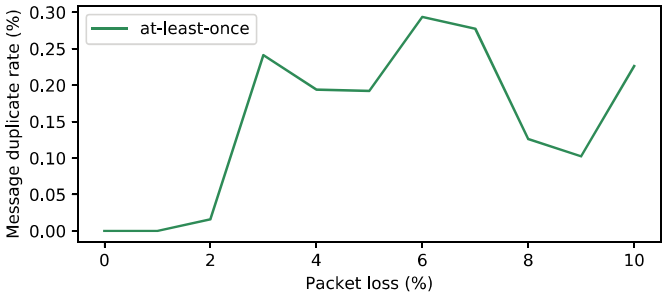

Fig. 5. Network packet loss and message duplicate rate

delivery, but sending larger messages under low network delay can save messages, while under higher delay sending smaller messages shows better reliability. Since each message is sent only once by the producer, the duplicate rate in this experiment is always zero.

Packet loss is another feature of poor network quality and measured as the percentage of packets lost with respect to the number of packets sent. Packet loss is very common in wireless networks, but also complex systems such as the Apache Kafka system can experience considerable packet loss. From Fig. 4b we observe that the message loss rate under at-most-once semantics increases as the packet loss rate increases, while the loss rate using at-least-once semantics stays at a low level of below 0.01%. The loss rate in at-least-once delivery is most likely greater than zero because messages are lost when the limit of retry times is reached or a timeout expires. Thus when encountering high packet loss applying at-least-once delivery semantics can significantly reduce the message loss rate.

As shown in Fig. 4b, the message loss rate of at-least-once delivery is 0.0038% when the network packet loss reaches 10%, while the loss rate of at-most-once is 0.283%. From the existing results we have not found a strong correlation between packet loss and message loss rate under at-least-once semantics.

The duplicate rate of messages is zero under at-most-once semantics, while at-least-once delivery can cause duplicated messages due to the retry mechanism. The impact of network packet loss on the message duplicate rate is depicted in Fig. 5. The duplicate rate is close to zero while the packet loss rate is below 2%. However, with higher packet loss the duplicate rate of messages increases rapidly and stays around 0.3%. There is no apparent growth trend of the message duplicate rate as the packet loss increases.

VII. Conclusion

We have presented our testing tool TRAK for studying the reliability of Apache Kafka. We have discussed three different delivery semantics of Apache Kafka, the at-most-once, the at-least-once and the exactly-once delivery. Often, the user has to balance reliability and performance, but it is not clear what the consequence of a parameter choice will be. To study these problems, we build a testbed of a Kafka cluster on Docker containers and use a network emulation tool to tune the network properties including delay and packet loss. We observed two reliability metrics for data delivery in Kafka. The two reliability metrics are the message loss rate and the message duplicate rate. The experimental results show that in different network scenarios different delivery semantics lead to better performance. In our future work we will consider more parameters, such as batch size and the retry timeout and will evaluate the performance metrics including message throughput. The target of studying reliability in Apache Kafka is to guide users into informed choices of delivery semantics for certain applications in given network scenarios. TRAK is easy to build and flexible to use, facilitating subsequent usage and research. The Docker image of Kafka brokers is available from DockerHub (https://cloud.docker.com/u/woohanx/repository/docker/woohanx/kfk_node) and the scripts to start it are pushed to GitHub (https://github.com/woohan/kafka_start_up.git).

References

[1] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[2] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye, "Building linkedin's real-time activity data pipeline." *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 33–45, 2012.

[3] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, 2017, pp. 227–238.

[4] V. John and X. Liu, "A survey of distributed message broker queues," *arXiv preprint arXiv:1704.00411*, 2017.

[5] T. R. Mayer, L. Brunie, D. Coquil, and H. Kosch, "On reliability in publish/subscribe systems: a survey," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 369–386, 2012.

[6] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 7–16.

[7] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.

[8] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[9] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon 2016*. IEEE, 2016, pp. 1–5.

[10] A. ledenev. Pumba-chaos testing and network emulation tool for docker. [Online]. Available: https://github.com/alexei-led/pumba