Learning to Reliably Deliver Streaming Data with 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*—The rise of streaming data processing is driven by mass deployment of sensors, the increasing popularity of mobile devices, and the rapid growth of online financial trading. Apache Kafka is often used as a real-time messaging system for many stream processors. However, efficiently running Kafka as a reliable data source is challenging, especially in the case of real-time processing with unstable network connection. We find that changing configuration parameters can significantly impact the guarantee of message delivery in Kafka. Therefore the key to solving the above problem is to predict the reliability of Kafka given various configurations and network conditions. We define two reliability metrics to be predicted, the probability of message loss and the probability of message duplication. Artificial neural networks (ANN) are applied in our prediction model and we select some key parameters, as well as network metrics as the features. To collect sufficient training data for our model we build a Kafka testbed based on Docker containers. With the neural network model we can predict Kafka's reliability for different application scenarios given various network environments. Combining with other metrics that a streaming application user may care for, a weighted key performance indicator (KPI) of Kafka is proposed for selecting proper configuration parameters. In the experiments we propose a rough dynamic configuration scheme, which significantly improves the reliability while guaranteeing message timeliness.**

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

I. Introduction

Streaming data is now flowing across various devices and applications around us. It can be sensor data transmitted among different Internet of Things (IoT) devices, messages generated in all kinds of microservices, or user activity records from websites. Despite the diversity of its contents and formats, streaming data generally arrives continuously in real-time. In modern large distributed systems this type of data grows to a huge volume called *big streaming data*, which needs to be quickly processed for extracting insights and useful trends [1]. A typical stream processing application consists of multiple processing nodes in the topology of a DAG (directed acyclic graph). To transfer data among those processing nodes, a messaging system, or middleware, is commonly applied to build real-time streaming data pipelines [2].

Apache Kafka is an open-source distributed streaming platform with growing popularity [3]. As a messaging system, it can persist streaming data, named messages in its cluster, and also provides Kafka Streams API for building stream processors [4]. The Kafka producer is responsible for acquiring streaming data from multiple sources and delivering it to the Kafka cluster, from which other processors can read the data they need. Therefore, the reliability of a producer is critical to ensure the completeness and correctness of streaming data. Unreliable contents of streaming data may finally result in faults in stream processing.

In this paper we address the challenge of choosing a proper configuration to guarantee reliable data delivery for complex streaming application scenarios. Developers often rely on empirical ways to configure the parameters in Kafka. However, running experiments and tests in enterprise-scale systems to figure out the best configuration can be time-consuming and costly. There are hundreds of configurable parameters in Kafka and only some of them have varying degrees of impact on system reliability. Besides, the requirements for reliability vary in different scenarios. Losing some messages in cases like website clickstream tracking or video streams transcoding can be tolerable. While when applying Kafka in banking systems, all messages in the stream should be processed exactly once without any exception [5]. We solve the above problems by building a prediction framework which can accurately predict the reliability metrics we define, given certain configuration parameters.

We create a Kafka testbed on Docker containers for exploring the approach of building a prediction model. In the testbed we can easily and quickly deploy a Kafka cluster, and restore different failure scenarios on it. To evaluate the reliability of message delivery, we define two reliability metrics for Kafka, the probability of message loss and the probability of message duplicate. Both are defined as the outputs of our prediction model. Due to the complexity of Kafka's architecture and the diversity of its configuration parameters, we use machine learning techniques to build the model. From the experimental results we find that the reliability metrics are significantly affected by both, the configuration parameters and the network condition. The types of streaming data, including the message size and timeliness, also have an impact on the metrics. We select several of the most sensitive factors as the main features for the prediction model. Through numerous experiments on our testbed, we collect plentiful training data. The accuracy of our predicted results is sufficient for comparing the impact from different configuration parameters.

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00068

In order to handle various requirements from all kinds of streaming applications, we present a weighted KPI (key performance indicator) to help choose proper configurations for Kafka. In this indicator both performance metrics (i.e. throughput) and the proposed reliability metrics are evaluated. Referring to the work in [6], the performance metrics are also predictable. When the configurations as well as the streaming data type and network status are known, we can generate the current weighted KPI through our prediction model. The weights can be adjusted depending on the requirements of different streaming applications. Thus the user can select proper configuration parameters of Kafka by checking its weighted KPI. Then a dynamic configuration method is proposed in our experiments to evaluate the effectiveness of the prediction model.

II. Related Work

Various kinds of streaming systems have been developed to process big streaming data [7]–[10], followed by numerous research work around them. While most studies focus on improving system performance, like throughput and latency, to achieve real-time processing, only a few of them consider the reliability of streaming data delivery. The investigation in [11] shows that streaming systems rely on processing semantics to guarantee the correctness of data transfer, and in Facebook's environment they mostly choose at-most-once or at-least-once semantics. Similarly, while developing a stream processing system for Twitter, the developers first implemented at-most-once and at-least-once semantics [12]. To achieve exactly-once semantics, additional computing resources and data stores are required to support transactions, thus imposing a performance penalty.

Research work on Kafka normally focuses on the performance evaluation [4], [13]. A queueing model has been proposed in [6] to predict the performance metrics of Kafka, including throughput and latency. Performance prediction models have been studied for different contexts in various systems [14]. The modeling work generally involves two parts: first the specification of the response variables that need to be predicted, and second the selection of the input features that may impact the outputs. The next step is to choose the proper model from rule-based or machine learning techniques [15]–[17]. In this paper we apply an Artificial Neural Network (ANN) in our model for its high accuracy and the many uncertainties of Kafka's architecture.

For real-time streaming applications, the message timeliness is considered an important indicator for reliability evaluation. In [18] the tradeoff between staleness and error of the data in a geo-distributed streaming system is studied. The timeliness of messages is also considered as one of the main features in our prediction model. Batching is another feature that has significant effect on Kafka's reliability. From [11] we know that a mix of streaming and batch processing can speed up long pipelines by hours. Furthermore, a control algorithm for dynamically adapting the batch size in stream processing systems has been presented in [19]. The research in [20] also indicates that dynamic configuration is efficient in streaming systems.

III. Methodology Description

In this section we outline the structure of Kafka and analyze the process of data delivery by introducing the concept of message states. To build a predictive model we determine the reliability metrics we want to measure and select the features that are considered strongly correlated. Then we introduce our experiment methodology and the approach used to collect training data for our model.

A. Kafka Overview

Apache Kafka offers a publish/subscribe service for processors in a streaming system, as the pipeline depicted in Fig. 1. From upstream applications the Kafka producer acquires source data like streaming video, mobile application requests, sensor data and bank transfer records. Then this data will be processed as messages and sent to the Kafka cluster, which is a distributed data storage that persists messages published by a Kafka producer. On a logical level, all messages are categorized into different topics in the cluster. Meanwhile, messages under the same topic are physically stored in multiple partitions. A number of brokers, which are actually server nodes, make up the hardware structure of a Kafka cluster, and the partitions in each topic are distributed across those brokers. Stream processors subscribe to a topic of their interest and read messages from it via Kafka consumer API.

The source data received by a producer not only comes from external applications. In Fig. 1 stream processor B conducts data transformation work like mapping, filtering and joining. In these cases it also publishes messages as a producer. Thus, the data streams in the streaming system are all ingested by Kafka, and the producer plays an important role in guaranteeing the completeness and consistency of the streaming data.

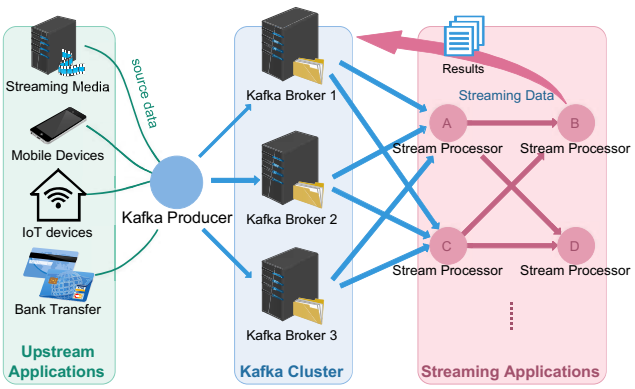

Fig. 1. Kafka in Streaming system

B. Message states

We use a state diagram to describe the states of a message from a Kafka producer to its topic, as depicted in Fig. 2. We take the state of a message before it is sent over the network as its initial state, marked as *Ready to be sent*. All the possible cases of a message delivery are listed in Table I. Under normal circumstances a message will be successfully delivered in its initial sending, denoted by $Case1$. Once a message is persisted on a broker for the first time, it is in the state *Delivered*. If the initial sending fails, as $Case2$ shows, the message will not exist in the Kafka cluster, it is in the *Lost* state.

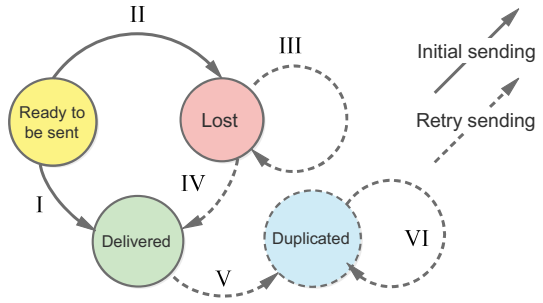

Fig. 2. The states of a message in Kafka

Under at-most-once delivery semantics, only $Case1$ and $Case2$ can happen. Because Kafka producer sends each message once and does not need response from the brokers. If at-least-once semantics is applied, the Kafka cluster will respond with an acknowledgement upon a successful message delivery. The producer will retry sending a message if it does not receive an acknowledgement until a timeout expires. If the retry still fails, the message stays in the *Lost* state. Since a producer can be configured to retry multiple times, it can keep retrying before receiving the acknowledgement. We use τ_r to denote the total number of retries, and τ_r·III to denote τ_r times transition III. Thus $Case3$ indicates that the message is still not delivered when τ_r reaches the maximum number of retries. Then the producer stops retrying and this message remains in state *Lost*. In $Case4$ we observe a successful delivery in the end. However, sometimes a message does not end in the *Delivered* state. In $Case5$ the message is already persisted in the cluster, but the broker fails to respond with an acknowledgement. Thus the producer still sees the message in the *Lost* state and retries to send it, resulting in duplicated messages. We use τ_d to denote those duplicated retries, and the message transfers to a new state called *Duplicated*. For the streaming applications without idempotent operation, duplicated messages may result in failures (i.e. a bank transfer is processed twice).

C. Reliability Metrics

Reliability metrics are derived from failure occurrence expressions. As a deliverer in the streaming system, a Kafka producer is responsible for moving its cargos (messages) from upstream applications to the cluster efficiently and safely. Each cargo should be carried to its destination exactly-once. In

TABLE I
ALL POSSIBLE CASES OF A MESSAGE DELIVERY IN KAFKA

Case Number	Transitions Order
1	I
2	II
3	II $\rightarrow \tau_r$·III
4	II $\rightarrow \tau_r$·III $\rightarrow$IV
5	II $\rightarrow \tau_r$·III $\rightarrow$IV $\rightarrow$V $\rightarrow \tau_d$·VI

Table I only $Case1$ and $Case4$ indicate a successful delivery, and the others are considered failures. We define two reliability metrics referring to the Probability of Failure on Demand (POFOD), which represents the probability of failure when a service is requested [21]. Here the service means sending a message in Kafka.

When a Kafka producer attempts to send a message to the cluster, we use $P_l = P(Case2 \bigcup Case3)$ to denote the probability of a message loss failure. The other metric is the probability of a message duplicate failure, denoted by $P_d = P(Case5)$.

D. Features for Prediction

Predicting the above reliability metrics is far from trivial, since several complex factors have an impact on them. Both physical resources (i.e., CPU and memory) and logical resources (i.e, data size) can significantly influence the reliability. The network connection between the Kafka producer and the cluster is another key factor that may cause failures. Network packet loss is very common for mobile and IoT devices, since they transfer data across wireless links. In this paper we assume that the hardware resources for a producer are fixed. We define the poor network connection between a Kafka producer and the cluster as faults to be injected. This is because we study how to obtain the best configuration in a scenario with a given machine of fixed resources. There are over 50 configurable parameters in a Kafka producer. Excluding the basically fixed parameters that are mandatory to connect producer and cluster, there are still many left. To select proper features for the prediction model, we run numerous experiments in our testbed. Normally, the default settings of Kafka will keep the system running, but far from a well performing one, therefore we select parameters based on a sensitivity analysis. A change in the quantitative parameter's default value of 50% should have observable impact on reliability metrics, otherwise the parameter is neglected.

In this paper we mainly introduce the following features for the prediction model: (a) Message size (b) Message timeliness (c) Network delay (d) Network packet loss rate (e) Delivery semantics (f) Batch size (g) Polling interval (h) Message timeout. The features consist of three parts. The first two features indicate the type of the streaming data. The next two are the network environment metrics. The latter are the configurable parameters in Kafka. In Section IV we introduce those main features through analyzing their impact on the reliability metrics in various application scenarios.

E. Testbed Design

We build a Kafka testbed on Docker containers. With this popular virtualization technology we can easily and quickly start, scale and test a Kafka system. The Kafka cluster consists of several running containers representing the brokers (three brokers in our experiments). The producer and consumer are also containers that join the same bridge network. In our experiments the messages should have these characteristics: the size is customizable, the lost and duplicated messages can be counted, and the content of a message is not our concern. Therefore we add an incremental message unique key to the source data, and the payload of the message is a string of definable length. To study the scenario of poor network connection between Kafka producer and cluster, we inject faults using NetEm, a popular Linux based network emulator [22].

We perform each experiment as follows. To avoid the legacy effects from the previous experiment, we start a new Kafka system and then create a new topic. The messages described above are provided as source data. The producer runs when the network faults are injected. When the producer finishes, we stop the fault injection and start a consumer container to consume all messages in this topic. Finally, we analyze the results by comparing the unique keys from source data and the messages received by the consumer.

The Docker image of Kafka brokers can be obtained from the 'woohanxkfk_node' repository on DockerHub (https://hub.docker.com/). The scripts to start a Kafka cluster are available from GitHub (https://github.com/woohan/kafka_start_up.git).

F. Training Data Collection

We use $\hat{P}_l$ and $\hat{P}_d$ to denote the predicted results of the reliability metrics (*probability of message loss* and *probability of message duplicate*). Thus the inputs and outputs of our prediction model can be expressed as follows.

$$\{\hat{P}_l, \hat{P}_d\} = f(M, S, D, L, Confs), \tag{1}$$

where M stands for the size of the message which the producer tries to send, and S represents the timeliness of the message. The network delay and packet loss rate at that time are denoted by D and L, respectively, while $Confs$ contains all the configuration features mentioned before. In each experiment we set the above features with fixed value and record them. Then we provide the source data of 1 million messages for the producer to process. From the messages received in the consumer container we count the number of messages in $Case2$ or $Case3$, denoted by N_l. Thus we derive the probability of message loss in this experiment as $P_l = N_l/10^6$. Similarly, we use N_d to denote the number of messages in $Case5$ and $P_d = N_d/10^6$.

Many experiments have been performed in our testbed to collect sufficient data points for a predictive model. Since the space of all possible features grows exponentially as we extend their ranges, we have to minimize the time spent on collecting training data while achieving sufficient accuracy. By observing the results of our benchmark tests, we proposed a collection method to simplify the space of training data, as illustrated in Fig. 3. We divide the cases we need to study into two major parts, according to the current network environment. If the network environment is normal ($D < 200ms$ and $L = 0\%$), we consider this a normal case since no network fault is injected. In the normal cases we filtered out the features that have no effect on the reliability metrics, and the remaining effective features are listed in the figure. By studying the impact of these effective features, we learnt how to deliver messages reliably in a normal network environment. Thus in the abnormal cases, where network faults are injected, proper values are chosen for those effective features and their impact can be neglected. The effective features we studied in the abnormal cases are listed in the right oval of Fig. 3. For each feature we specify the range of possible variables according to real world systems.

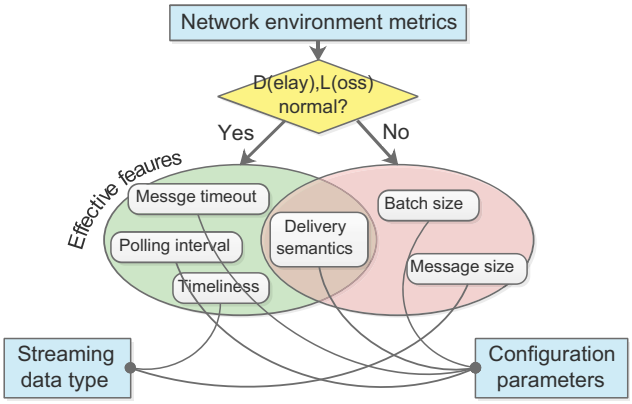

Fig. 3. Training data collection design

G. Prediction model

The collection method mentioned above reduces both the number of experiments needed and the size of the training data. Another advantage is that we can adjust the structure of the ANN accordingly to improve prediction efficiency and accuracy. For instance, for at-most-once delivery semantics we only have to predict P_l since we know there will be no duplicated messages. Thus the output layer contains just one neuron and the input layer can be reduced as well. Due to our explicit training data collection, the outputs and inputs are strongly correlated, therefore a fairly standard ANN model performs well enough. Generally, we use 4 hidden layers in the ANN model and the number of neurons are 200, 200, 200 and 64 respectively. The learning rate is 0.5 and the number of epochs is 1000. Some adjustments for different cases and the details of the ANN model can be found in https://github.com/woohan/kafkaPrediction.git.

To build the ANN model that best fits the training data, we apply the Stochastic Gradient Descent (SGD) optimizer. SGD fits our case well and avoids over-fitting or corner cases such that $\hat{P}_l$ or $\hat{P}_d$ become negative. Our experimental results

show that the mean absolute error (MAE) is below 0.02, which is sufficient for comparison and for choosing the appropriate configuration parameters. This can be observed from Fig. 4 to Fig. 6, where we compare samples of the test data with the predicted results.

IV. Lessons Learned

By observing the results from our prediction model, we see how to properly configure Kafka under complex scenarios. From the perspective of a Kafka user, there are several KPIs that demonstrate how efficiently the producer is serving user demands. For simplicity we propose a weighted KPI of Kafka:

$$\gamma = \omega_1\varphi + \omega_2\mu + \omega_3\left(1 - P_l\right) + \omega_4\left(1 - P_d\right), \left(\sum_{i=1}^{4}\omega_i = 1\right). \tag{2}$$

The criterion to choose the best configuration for Kafka is to maximize the value of γ. The performance metrics φ and μ are the utilization of network bandwidth and the mean service rate of a Kafka producer under normal circumstances (i.e. good network connection). Both can be predicted for a given system deployment and configuration parameters, referring to the research in [6]. In this paper we provide empirical default weights as $\omega_1, \omega_2, \omega_3, \omega_4 = 0.3, 0.3, 0.3, 0.1$, since duplicated messages can be tolerated by most applications due to idempotent mechanism. The user should adjust the weights ω_i depending on the specific application. For instance, if high throughput and low latency are prioritized, while losing or duplicating some messages is acceptable, ω_1 and ω_2 can be increased to 0.4, while setting $\omega_3 = 0.1$.

A. How message size matters

A Kafka producer transports streaming data as byte arrays before sending them over the network. The efficiency of this serialization work shows strong correlation with the size of a message, M, where with larger M the service rate μ is lower [6]. In real systems M fluctuates around hundreds of bytes, i.e. the average size of web server access records is around 200 bytes. In our experiments where we injected a poor network connection with high packet loss rate, the reliability metrics are also affected by M. In the example shown in Fig. 4 we set the network delay to $D = 100ms$ and the packet loss rate is $L = 19\%$. We observe the changes in P_l with M ranging from 50 to 1000 bytes.

We show the results for the two different delivery semantics and find that small messages are more likely to be lost under both semantics. It is interesting that at-most-once outperforms at-least-once for messages smaller than 200 bytes approximately. We observe that for given message size $M = 100\ bytes$, the probability of message loss under at-most-once semantics ($P_l \approx 85\%$) is more than 20% higher than under at-least-once semantics ($P_l \approx 63\%$). We explain this as follows. Since Kafka uses a binary protocol over TCP, there is a basic retransmission mechanism on the transport layer. The retransmissions on the network interface of the docker container can

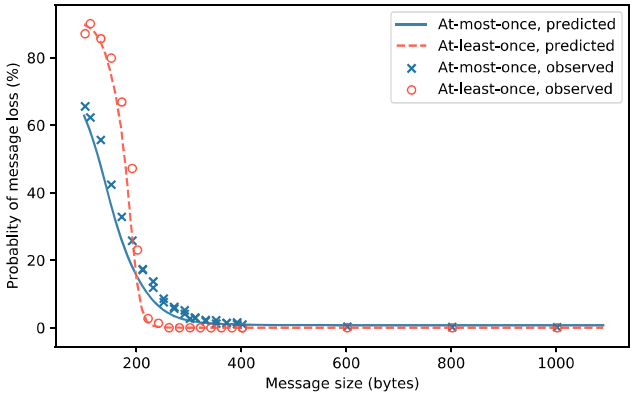

Fig. 4. The message size M and its probability of loss P_l for network packet loss rate $L = 19\%$

be observed by Wireshark, an open-source packet analyzer. Under at-least-once delivery semantics the producer requires an acknowledgement from the brokers, which will preempt network bandwidth with TCP retransmissions. The conflict is more pronounced with smaller messages as the service rate μ grows, and the number of acknowledgements required per time unit increases. For larger messages P_l of both semantics is below 1%, where at-least-once performs better. Although this is not shown in Fig. 4 in each experiment at-least-once can save approximately 3000 more messages than at-most-once.

B. Message timeliness

In some streaming systems only the newest data is valuable (i.e. automatic vehicle location) and delivering a stale message can be futile. We use T_p to denote the interval between the time when a message arrives to the producer and the time when it is $Delivered$. A message delivery that costs $T_p > S$ is stale. In our work the message timeliness S denotes valid time of data which depends on specific applications. The configuration parameter *message timeout*, denoted T_o is the maximum time a producer can use to deliver a message, including retries. Therefore, the total time to deliver a message is $T_p = min\{(1/\mu + D), T_o\}$. Choosing a proper value of T_o avoids spending too much time on one message and guarantees the timeliness of messages. For instance, when the network delay D is high, a strict T_o is required. However, from our experiments we observe that setting T_o lower than $1500ms$ can cause message loss in at-most-once delivery, even when no network faults are injected, as illustrated in Fig. 5. In this case using the at-least-once delivery can significantly reduce P_l.

C. Scalability of a producer

In the example of Fig. 5, the producer is fully loaded, which means it acquires source data in the highest speed that I/O devices can handle. However, receiving messages faster than the producer can handle increases the waiting time of messages, and those exceeding T_o are lost. In order to guarantee both timeliness and reliability, a common solution

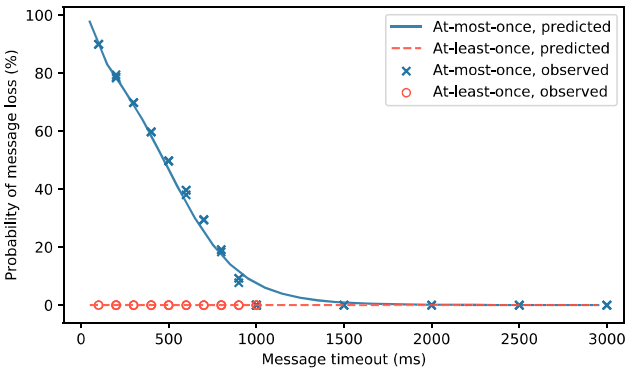

Fig. 5. The configuration parameter message timeout T_o and the probability of message loss P_l, no network fault injected

is to reduce the rate at which the producer receives data and to scale the number of producers. The polling interval δ is the configurable time interval between a producer's calls to acquire source data from upstream applications, thus the message arrival rate is $\lambda = 1/\delta$. The results in Fig. 6 indicate that increasing δ can effectively avoid message loss, while T_o is fixed at $500ms$. We observe that under full load ($\delta = 0$) the probability of message loss is above 45%, while setting $\delta = 90ms$ reduces P_l to less than 10%.

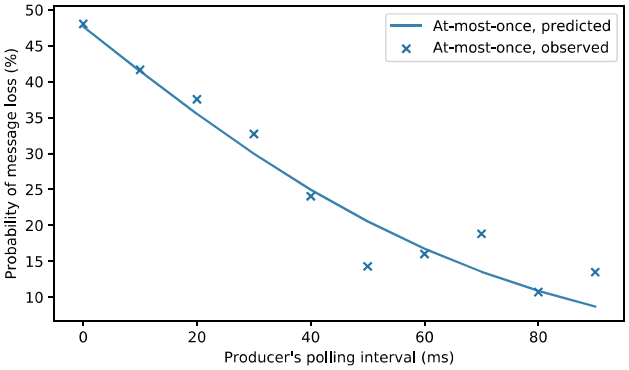

Fig. 6. The configuration parameter polling interval δ and the probability of message loss P_l, no network fault injected, $T_o = 500ms$

Supposing that We know increasing the polling interval of a single producer from δ to $\delta + \Delta\delta$ meets our desired P_l, the next step is to scale the number of producers from N_p to N_p'. The principle of scaling is to meet the overall message arrival rate from upstream applications, which means $N_p/\delta = N_p'/(\delta + \Delta\delta)$. Scaling an overloaded Kafka producer is an effective solution to improve its reliability and avoid message staleness. However, this method requires additional hardware resources to run scaled producers.

D. Batching can be effective

In the above experiments we apply typical stream processing which means that messages are not batched and each message is sent once it is ready. However, in some failure scenarios we find that batch processing can significantly improve the producer's reliability. We run experiments with network packet loss rate ranging from 0% to 50%, and observe the effects of batch processing on the probability of message loss. From Fig. 7 we observe the predicted effects under at-most-once and at-least-once delivery semantics, denoted by solid and dashed lines, respectively. The batch size B is the number of messages per batch. This parameter indicates how many messages will be accumulated in the producer and then sent at once. Both solid and dashed curves illustrate the results with B ranging from 1 to 10 in the order from left to right. The curves with $B = 1$ (far left) represent stream processing when there is no batching.

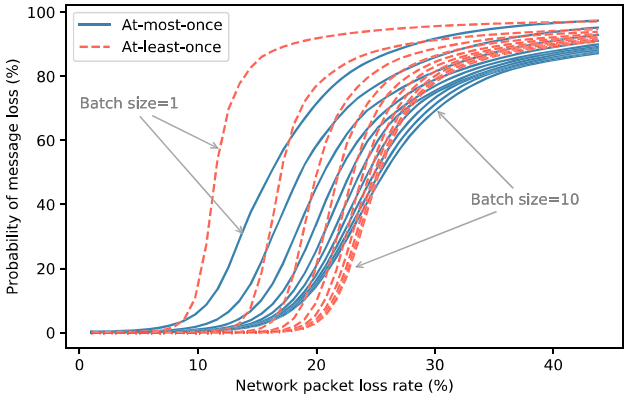

Fig. 7. Comparison of different batch size and delivery semantics, injected with various network packet loss rate

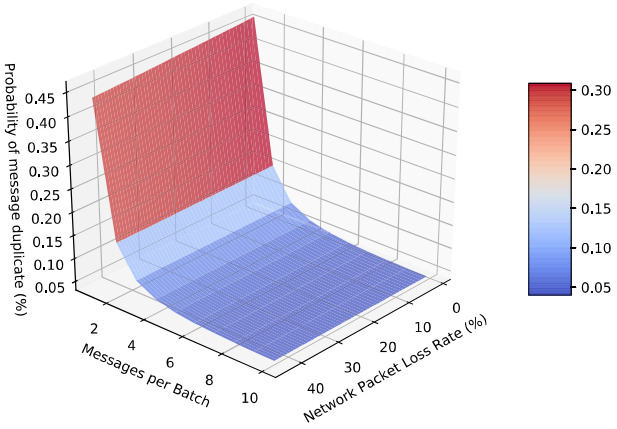

Fig. 8. The configured batch size and the probability of message duplicate with at-least-once delivery, injected with various network packet loss rate

The TCP retransmission mechanism performs well under the packet loss rate $L \approx 8\%$ above which P_l rises rapidly as L grows. Our insight is that TCP retransmission has its limits while facing high packet loss rate, but we can still do more by properly configuring Kafka producer. We observe that in at-least-once delivery, accumulating two messages in one batch remarkably reduces P_l from over 80% to less than 5%, when

$L = 13\%$. With higher packet loss rate setting larger batch size B saves more messages, but the effect falls as B increases. This is similar with at-most-once delivery semantics.

The result above forms a part of the basis for dynamic configuration. In the case with 30% packet loss rate, changing configuration parameters has little effect. However, for the failure scenarios with L around 10% to 20%, choosing the proper delivery semantics or batch size will significantly improve producer's reliability. Our prediction model helps users make decisions in varied network environments. In fact, the KPIs in Equation (2) are all somehow correlated with batch size B. Larger B results in lower μ and also increases the end-to-end latency [19]. The impact of B on the probability of message duplicate P_d is shown in Fig. 8. It is obvious that P_d can be reduced by batching, while no strong correlation between P_d and L is observed.

V. Dynamic Configuration

To measure the effectiveness of the prediction model, we run the Kafka producer with dynamic configuration in a complex network environment. In our evaluation we assume the network status to be known, thus we generate the corresponding configuration parameters offline and save them to a configuration file. While using the unstable network environment, the producer reads these parameters from the configuration file. Running an online algorithm for dynamic configuration is beyond the scope of this paper.

The predicted reliability metrics can be obtained according to Equation (1). Combining with the model in [6], we obtain the weighted KPI γ from Equation (2). If γ is less than the user-defined requirement, the parameters should be adjusted to increase its value. From our observation on the predicted results, the outputs generally increase or decrease monotonically with the inputs. Therefore the purpose of this method is not to find the maximum value of γ, but the one that meets the user's requirement. For each parameter, we move its current value stepwise forward or backward and substitute the value into our prediction model to obtain the predicted results. We repeat this until the predicted γ meets the requirement. Changing configuration parameters too frequently will cause additional coordination overhead due to shuffling communication among producer and brokers [20]. Thus in our experiments we check γ every other time interval (i.e. every 60 seconds).

The network status in our experiment is depicted in Fig. 9. To simulate a realistic network environment, the network delay follows a Pareto distribution [23], while the network packet loss rate is generated from the Gilbert-Elliot model [24]. It is a two-state Markov model that has been widely applied to analyze measurements on wireless networks. We design three kinds of data streams in this network environment and evaluate the performance of dynamic configuration with them, respectively, as depicted in Table II. The text messages from social media must be delivered quickly with the lowest loss rate. In log analysis applications, timeliness of data streams (i.e. web server access records) is not strict but the messages are required to be complete, while duplicates can be acceptable due to idempotent processes. Any individual message in online games is small (i.e. less than 100 bytes), since they only contain mouse or keyboard signals. However, the game traffic message needs to be delivered accurately in real-time, otherwise the player's gaming experience is greatly reduced. Our suggested values of weights ω_i are listed in Table II.

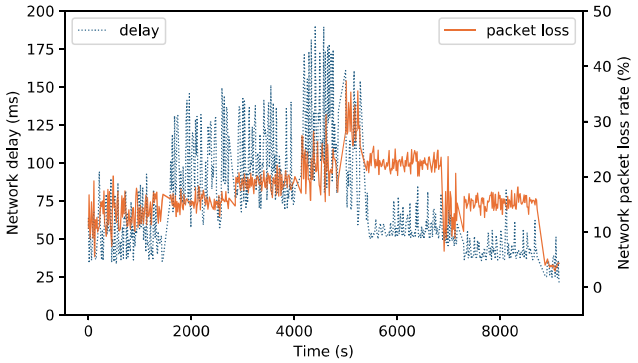

Fig. 9. Network connection between Kafka producer and cluster in the dynamic configuration experiment

According to Equation (1), since the features M, S, D and L are all some functions of the elapsed time t, we use $P_l(t)$ and $P_d(t)$ to denote the reliability metrics in this experiment. Given the work load of source data $\lambda(t)$, which also changes with t, we obtain the total number of messages that arrives the producer as $\int \lambda(t)dt$. Therefore the overall message loss rate R_l and duplicate rate R_d in this experiment are defined as follows:

$$R_l = \frac{\int \lambda(t)P_l(t)dt}{\int \lambda(t)dt}, \quad R_d = \frac{\int \lambda(t)P_d(t)dt}{\int \lambda(t)dt} \tag{3}$$

TABLE II
The Overall Message Loss Rates and Duplicate Rates

	Messages from social media		Web sever access records		Game traffic messages	
	Default	Dynamic	Default	Dynamic	Default	Dynamic
R_l	55.76%	17.58%	42.94%	6.54%	87.50%	13.9%
R_d	0.32%	0.63%	0.04%	0.00%	0.01%	0.00%
ω_i	0.4, 0.3, 0.2, 0.1		0.1, 0.1, 0.7, 0.1		0.2, 0.4, 0.2, 0.2	

From our experimental results in Table II we observe that comparing to the static default configuration of Kafka, our dynamic configuration method reduces R_l significantly. In the experiments with default configurations about half of messages are lost. For message streams from social media, the dynamic configuration method reduces the loss rate at the expense of an increased duplicate rate. We observe that while handling the web server access records, which do not have high requirements for timeliness, dynamic configuration performs well. However, the priority of timeliness and accuracy are both very high in gaming traffic data, hence we

have to scale the Kafka producer to reduce the loss rate. The details of those dynamic configuration files can be found in https://github.com/woohan/dynamicConf.git.

VI. Conclusion

In this work, we learn how to reliably deliver messages using Apache Kafka in streaming systems. Based on the analysis of possible message states in Kafka, we introduce two metrics for the reliability evaluation. A machine learning model is proposed to predict those metrics, while the application scenarios are known. We run experiments on a testbed of Kafka built on Docker containers to observe possible influencing factors. From all the factors we select the most sensitive ones as the main features, which contain the type of source data, the network status and the configuration parameters of Kafka. Based on the prediction model we introduce some approaches for using Kafka reliably in various cases. We present a weighted KPI for the users of Kafka to select proper configuration parameters in complex network environments. A dynamic configuration method is applied in our experiments to evaluate the effectiveness of our prediction model. The results show that changing configuration parameters dynamically is the ideal solution for guaranteeing reliability in complex and changing scenarios. The main takeaways for Kafka users are:

- If the user can choose the message size for upstream applications, messages should be larger than 300 bytes as our analysis shows that larger messages see a lower risk of getting lost.
- Even under good network condition, the overloaded producer may lose messages. We provide a scaling strategy for users to balance the load and hence reduce the loss rate.
- When the user cannot change the size of incoming messages, batching messages before sending them over the network can significantly reduce the loss rate.

Our research can be extended in several directions, which call for further research. More failure scenarios including the failure of brokers should be studied to complete the prediction model. We do not make a deep dive into the retry strategy in Kafka, since the impact is not pronounced in our experiments. Currently Kafka does not provide dynamic configuration for all parameters and the producer needs to be restarted every time the configuration changes. The dynamic configuration method that we proposed is very rough and could be refined in the future.

References

[1] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 1–14.

[2] I. Lee and K. Lee, "The internet of things (iot): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[3] ApacheKafka. Powered by - apache kafka - the apache software foundation. [Online]. Available: https://kafka.apache.org/powered-by

[4] 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.

[5] G. Ramalingam and K. Vaswani, "Fault tolerance via idempotence," in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2013, pp. 249–262.

[6] H. Wu, Z. Shang, and K. Wolter, "Performance prediction for the apache kafka messaging system," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 154–161.

[7] ApacheSpark. Powered by - apache kafka - the apache software foundation. [Online]. Available: https://https://spark.apache.org/

[8] ApacheStorm. Apache storm - a free and open source distributed realtime computation system. [Online]. Available: https://https://storm.apache.org/

[9] ApacheFlink. Apache flink - stateful computations over data streams. [Online]. Available: https://https://flink.apache.org/

[10] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[11] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1087–1098.

[12] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[13] 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.

[14] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 363–378.

[15] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC international conference on performance engineering*. ACM, 2015, pp. 145–156.

[16] E. Thereska and G. R. Ganger, "Ironmodel: Robust performance models in the wild," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 253–264, 2008.

[17] P. Romano and M. Leonetti, "Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning," in *2012 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2012, pp. 786–792.

[18] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 361–373.

[19] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–13.

[20] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1891–1904, 2017.

[21] G. Kaur and K. Bahl, "Software reliability, metrics, reliability improvement using agile process," *International Journal of Innovative Science, Engineering & Technology*, vol. 1, no. 3, pp. 143–147, 2014.

[22] A. Jurgelionis, J.-P. Laulajainen, M. Hirvonen, and A. I. Wang, "An empirical study of netem network emulation functionalities," in *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2011, pp. 1–6.

[23] W. Zhang and J. He, "Modeling end-to-end delay using pareto distribution," in *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*. IEEE, 2007, pp. 21–21.

[24] A. Bildea, O. Alphand, F. Rousseau, and A. Duda, "Link quality estimation with the gilbert-elliot model for wireless sensor networks," in *2015 IEEE 26th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE, 2015, pp. 2049–2054.