2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)

A Reactive Batching Strategy of Apache Kafka for Reliable Stream Processing in Real-time

Han Wu*, Zhihao Shang†, Guang Peng*, Katinka Wolter*
*Institut für Informatik, Freie Universität Berlin, Berlin, Germany
†School of Information Engineering, Zhengzhou University, Henan, China
Email: *{han.wu, guang.peng, katinka.wolter}@fu-berlin.de,† iezhshang@zzu.edu.cn

***Abstract*—Modern stream processing systems need to process large volumes of data in real-time. Various stream processing frameworks have been developed and messaging systems are widely applied to transfer streaming data among different applications. As a distributed messaging system with growing popularity, Apache Kafka processes streaming data in small batches for efficiency. However, the robustness of Kafka's batching method against variable operating conditions is not known. In this paper we study the impact of the batch size on the performance of Kafka. Both configuration parameters, the spatial and temporal batch size, are considered. We build a Kafka testbed using Docker containers to analyze the distribution of Kafka's end-to-end latency. The experimental results indicate that evaluating the mean latency only is unreliable in the context of real-time systems. In the experiments where network faults are injected, we find that the batch size affects the message loss rate in the presence of an unstable network connection. However, allocating resources for message processing and delivery that will violate the reliability requirements implemented as latency constraints of a real-time system is inefficient. To address these challenges we propose a reactive batching strategy. We evaluate our batching strategy in both good and poor network conditions. The results show that the strategy is powerful enough to meet both latency and throughput constraints even when network conditions are variable.**

***Index Terms*—Stream processing, Reliability, Real-time, Performance, Apache Kafka, Docker**

I. Introduction

In modern big data applications the challenge of processing streaming data comes from four facets, known as the 4V's, i.e., volume, variety, velocity and veracity [1]–[3]. Streaming data means any unbounded, ever growing, infinite data set which is continuously generated by various sources. Common examples include sensor data produced by Internet of Things (IoT) devices, user activities collected on websites and payment requests sent from mobile devices. In many application scenarios, the streaming data needs to be processed in real-time because its value can be futile over time. For instance, an ideal fraud detection system should identify a fraudulent transaction before it completes, thus avoiding financial losses [4].

To deal with streaming data in real-time, a new fleet of stream processing systems are developed, including Apache Samza, Apache Flink, Google's MillWheel, Twitter's Heron [5]–[8], etc. Despite some functional differences, these systems commonly consist of multiple processing nodes in the topology of a DAG (directed acyclic graph). To build real-time streaming data pipelines among those nodes, message middleware technology is widely applied. As a distributed messaging system with high durability and scalability, Apache Kafka has become very popular among modern companies like Uber, Netflix, Twitter and Spotify [9]. Kafka can ingest streaming data from upstream applications and store the data in its distributed cluster, which provides a fault-tolerant data source for stream processors [10]. Kafka processes streaming workloads as a continuous series of batch jobs on small batches of streaming data. The size of the batches can significantly impact the throughput and end-to-end latency in the stream processing system. Larger batch size improves the utilization of network bandwidth, while causing higher end-to-end latency [11]. Choosing a proper batch size is a key step towards efficient stream processing. However, the robustness of the batching strategy in Kafka against various operating conditions has not been well explored.

In this paper, we address problems in the performance measurement of existing stream processing systems. Since most of them evaluate the mean end-to-end latency, it is difficult to guarantee the timeliness of message delivery comprehensively. Moreover, Kafka provides flexible batching configuration parameters, but the effects on message timeliness remain unknown. In practice, enterprise businesses cannot afford the manual work to find the optimal configuration of batch size. To address these challenges, we study how the batch size impacts Kafka's performance and reliability. To collect sufficient experimental data, we build a Kafka testbed using Docker containers, with which a Kafka cluster can be easily and quickly deployed. During the analysis of the experimental results, we fit the distribution of end-to-end latency and propose a prediction model to estimate the latency violation rate. A new quality-of-service (QoS) metric, *timely throughput* is defined to help choose the appropriate batch size. We then study the effects of the batch size on the reliability of data delivery when network faults are injected, and use machine learning techniques to predict the message loss rate. In the experimental evaluation we compare the performance of our batching strategy with other empirical methods under both good and poor network conditions. The experimental results show that our batching strategy is able to deliver the most messages within user-defined latency constraint. The Kafka users may refer to the workflows in this paper to obtain the corresponding features in their real-time systems,

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00028

thus optimize their batching strategy according to the specific operation environment.

II. Background

In this section we discuss the role that Kafka plays inside a stream processing system. Then we introduce the components of Kafka's end-to-end latency.

A. Kafka in the stream processing system

As a messaging system, Kafka's job in the stream processing system is nothing but transferring streaming data from one place to another for various processing purposes. The messaging schema of Kafka is Publish/Subscribe (pub/sub), which is built on the concepts of producer, topic, broker and consumer. A producer is the application that sends messages (streaming data) to their specific topics, which are different logical categories of messages. Correspondingly, a consumer is the application that subscribes to the topic of its interest and reads messages from it. The core part of this messaging system, the Kafka cluster is a distributed storage system constructed of multiple brokers. Each broker acts as a server that receives messages from a Kafka producer, stores messages on disks and replicates them for fault-tolerance. All messages in the same topic are distributed across the brokers, stored in ordered write-ahead logs called partitions. In enterprise applications, they always have a configured number of partitions per topic, and each broker hosts one or more partitions under this topic [10].

Fig. 1 depicts a simplified example of the stream processing pipeline using Kafka as the messaging system. The Kafka cluster consists of three brokers, and each circle above represents a stream processor that runs a certain operation on the streaming data. The source data comes from various upstream applications like IoT sensors, payment terminals and mobile devices, then it is ingested by a stream processor. In this example the processor performs filter processing and publishes the valuable messages to a specific topic, e.g. topic A in the figure. Other processors in the streaming system ingest the required messages from the Kafka cluster by subscribing to the corresponding topics. For instance, the map processor subscribes to topic A, consumes messages, operates map processing and produces the results to topic B for other processors to subscribe. Additionally, downstream applications may apply their UDFs (user-defined function) in the stream processors to obtain the results for various services or for decision making. In other words, each processor applies both the producer and the consumer API to interact with the Kafka cluster, thus the streaming data flows across different processors in the system.

B. End-to-end latency

The end-to-end latency is evaluated as one of the most important performance metrics in Kafka, generally described as the time it takes for a message to move from leaving the producer until arrival to the consumer. However, observing on a lower level, the definition of this latency is ambiguous in related work [10], [12]. In this paper we give an explicit

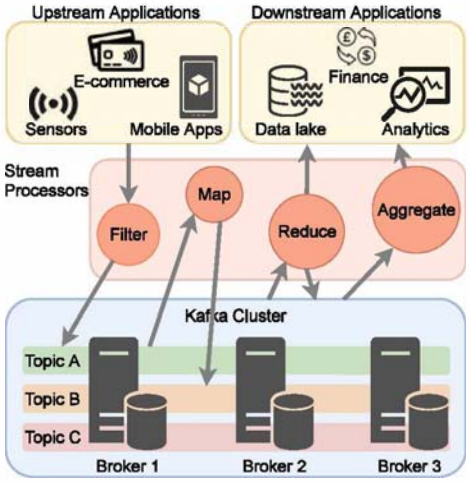

Fig. 1. Kafka in Streaming system

definition of the end-to-end latency, denoted by L_K, and introduce its major components in detail. Fig. 2 shows the path of a message from a Kafka producer to a consumer through the cluster. L_K is the time starting when the producer calls the **send()** method to produce a message and until this message is received by the consumer via the **poll()** method.

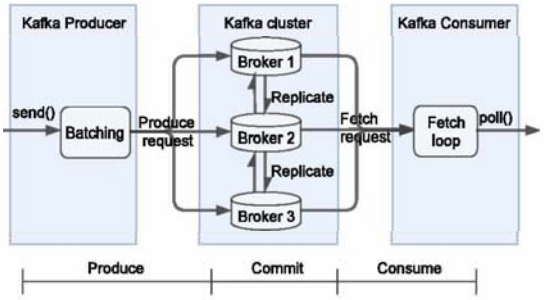

Fig. 2. The components of Kafka's end-to-end latency

The first part of L_K is the *produce time*, during which the message is batched with other messages in the producer, then the messages are sent together to one of the brokers. The *produce period* lasts until the message is appended to the leader partition's log. Then Kafka replicates the message for fault-tolerance, e.g. the batch on broker-2 is replicated to the follower partitions on broker-1 and broker-3. Generally, a message is committed, which means it is ready to be consumed, only after the replication process finishes, and this phase is called the *commit time*. The last part, the *consume time* ends when the consumer receives the message from the cluster by executing a fetch loop.

III. Problems and Challenges

In this section we introduce the problems of current performance metrics and challenges in the batching strategy of Kafka.

A. Batching methods

From Fig. 2 we know that a Kafka producer batches messages before sending them to the cluster, since treating **streaming data in small batches improves efficiency at scale** [13]. An individual request for sending each message across the network would result in excessive overhead, which can be reduced by accumulating messages into a batch and sending them together. It is observed that under heavy load, using **batching significantly improves Kafka's throughput [11]. The** reason is that batching reduces the overhead (e.g. number of system calls and hardware interrupts), thus freeing more task threads. However, larger batch size also leads to higher end-to-end latency, because more messages will wait in a batch before being propagated. Hence, choosing an appropriate batch size is the key for optimizing the tradeoff between latency and throughput.

In most cases the batch size refers to the maximum data size of accumulated messages in one batch, while it also can **be defined in terms of time interval [13]. Kafka provides both configuration parameters to control batching. In this paper,** for simplicity, we use the maximum number of messages per batch to represent the parameter which limits the space of a batch, namely *spatial batch size* and use B_S to denote it. The parameter which limits the maximum time to construct a batch is called *temporal batch size*, denoted by B_T. When **both parameters are configured, the Kafka producer will send** a batch when either B_S or B_T **is fulfilled. For instance, in** our tests, we set $B_T = 1000ms$, and change the value of B_S to observe the end-to-end latency L_K and its components. In each test $20,000$ messages are delivered to Kafka, L_K of every message is analyzed statistically. The implementation details of those tests will be introduced in Section. IV. We use l_1 to denote the *produce time* in Fig. 2, and l_2 to denote the *commit* and *consume time*, thus $L_K = l_1 + l_2$. Fig. 3 illustrates the impact of B_S on the mean and standard deviation of L_K, l_1 and l_2.

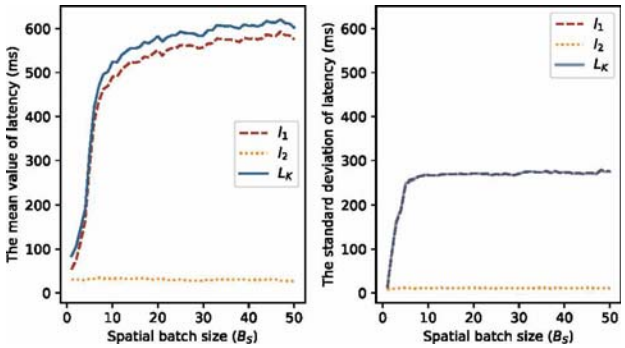

Fig. 3. The impact of spatial batch size on latencies when temporal batch size $B_T = 1000ms$

Obviously, the *produce time* l_1 dominates the end-to-end latency, and it is strongly correlated with B_S due to the batching overhead. Meanwhile, the *commit* and *consume time* remain unchanged. This is because Kafka decouples the production and consumption processes, as mentioned in Section II-A, thus changing B_S has no effect on l_2. We observe that the mean L_K, denoted by $\bar{L_K}$, rises rapidly from less than $100ms$ to over $500ms$ as we increase B_S from 1 to 10. This happens when B_S takes effect during batching, while the limit of B_T has not been reached. As we continue to increase B_S, $\bar{L_K}$ stabilizes at around $600ms$, since the batching method is controlled by B_T, **which is a fixed value.**

To our knowledge, the configuration parameters *spatial batch size* and *temporal batch size* of Kafka have not been **studied together. In practice, reasonable configuration of these** two parameters determines the performance of the batching strategy in Kafka. We address this challenge with a reactive batching strategy in our paper.

B. Latency evaluation

The timeliness of a message is important for real-time **guarantees in stream processing systems, and its definition depends very much on the specific application. Many downstream applications stipulate user-defined latency constraints** for processing the newest messages from upstream applications. For instance, in a user-interactive system, the round-trip latency should be less than 100 ms for stability [14]. We **define this upper bound as the end-to-end latency constraint,** denoted ζ_L. **Given a specific** ζ_L, the purpose of our reactive **batching strategy is to properly configure** B_S and B_T to guarantee $L_K < \zeta_L$. In existing performance studies on Apache Kafka they only evaluate $\bar{L}$, the mean L_K, which is not comprehensive [10]–[12]. We use σ_K to denote the standard deviation of L_K, as depicted on the right of Fig. 3. It can be observed when B_S takes effect, as more messages are accumulated in a batch, σ_K increases, which indicates that the dispersion of the messages' L_K becomes higher. Fig. 4 shows the distributions of L_K in three selected individual tests, with B_S **configured as** $1, 4$ and 10, respectively.

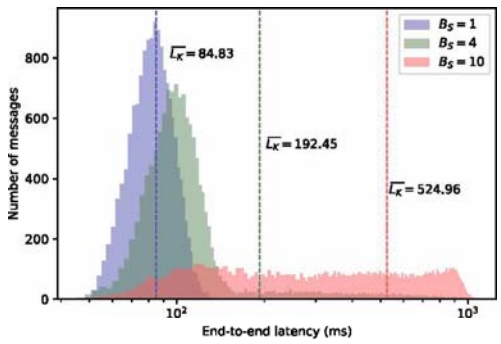

Fig. 4. The distribution of end-to-end latency with different spatial batch size

For better comparison L_K is shown on a log scale. With increasing B_S, the distribution of L_K is stretched. In the test with $B_S = 1$, which means the producer publishes messages without batching, the distribution of L_K ranges approximately between $60ms$ and $120ms$. When $B_S = 4$, the distribution shows a heavy tail, where L_K of some messages exceeds

$900ms$. Considering an application which requires ζ_L to be $200ms$, if we only use the mean value (marked as $\bar{L_K}$ in Fig. 4), then the constraints are met in both cases. However, in the case with $B_S = 4$, over 22% of the messages violate the latency constraint due to the heavy tail. Moreover, when B_S increases further and B_T starts to limit the batch size, the distribution of L_K is even more widely dispersed.

From these experimental results we see the limitations of only using the mean end-to-end latency as performance metric. This unreliability becomes more pronounced when the batches become larger.

IV. Testbed Design

We collect a vast array of performance metrics in Kafka for statistical analysis. Although Kafka provides a lot of metrics for monitoring via Java Management Extensions (JMX), we choose to build our own performance analysis tool for several reasons. Kafka exposes internal conclusive metrics for enterprise-level monitoring. For instance, the latency metric is measured as an average over the past 1 or 5 minutes. The 75th and 99th percentiles of the latency are also available, but **the latency violations of a specific ζ_L** remains unclear. In this section we introduce our testbed of Kafka built using Docker, an industrial-grade lightweight virtualization technology [15]. As depicted in Fig. 5, the testbed consists of four parts: the Kafka system based on Docker containers, the *message generator*, the *network emulator* and the *statistical analysis tool*.

A. Testbed components

All components of the Kafka system testbed are based on Docker. Each Kafka broker, producer or consumer is a running Docker container which embeds all the required dependencies and codes. Using container technology guarantees the correctness of Kafka's configuration, within a lightweight, standalone and reproducible execution environment [16]. The containers join the Docker bridge network to communicate with each other and transfer messages. With the Docker Compose tool a Kafka cluster with user-defined number of brokers can be started or shut down in a fast and convenient manner. We run the testbed on a computer with Intel Core i7-8700 CPU (12 cores), 32GB RAM and 2TB HDD. We build the Kafka cluster from version 2.3.0, using the Docker version 19.03.6, running on Ubuntu 18.04.4 LTS. Since all the Docker containers run on the same machine, their clocks are synchronized. In our tests we observe that the time difference between two containers is less than $0.04ms$, which guarantees the accuracy of the collected end-to-end latency results.

B. Message generator

Through the *message generator* we define the attributes of the streaming data from upstream applications, such as the size of messages and the arrival rate. We need to investigate the trace of every single message, including the end-to-end latency and delivery state. Therefore, each message is integrated with an incremental unique key for tracing. The timestamps of the operations on an individual message are recorded to determine the latency as introduced in Section III-A. To obtain the *produce time* l_1, we record both timestamps when the producer calls the **send()** method to publish a message, and when the message is written to its partition. Once the consumer receives the message, the timestamp is recorded into counter l_2.

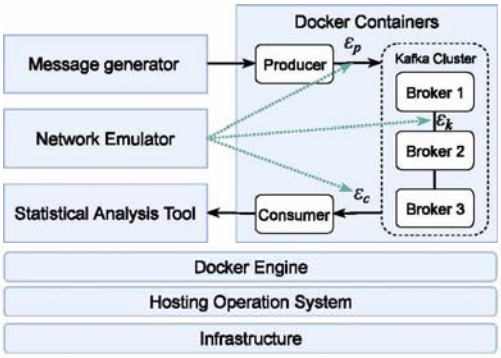

Fig. 5. The architecture of docker container based Kafka testbed

C. Network emulator

As illustrated in Fig. 5, we use $\varepsilon_p(d_p, l_p)$ to denote the network condition between a Kafka producer and the cluster, where d_p and l_p represents the round-trip network delay and packet loss, respectively. Similarly, the network environment among brokers is $\varepsilon_k(d_k, l_k)$, while the connection between a Kafka consumer and the cluster is $\varepsilon_c(d_c, l_c)$. All these network metrics are controlled by the *network emulator*, which uses the Linux Traffic Control tool to emulate a real world network environment [17]. We also explore the effect of the batch size on the consistency of message delivery under poor network condition, e.g. high network delay and packet loss rate. Therefore another important function of the *network emulator* in our study is network fault injection, which causes message loss or duplication. In some specific application scenarios, e.g. a banking system, losing messages or receiving duplicated messages may result in failures [18].

D. Statistical analysis tool

Through the experiments we explore the impact of the batch size on Kafka's performance and reliability. Before each individual test, all the existing containers are killed and a new Kafka system is built, thus avoiding the legacy impacts from the previous test. In each test the Kafka configuration parameters are fixed, as well as the network environment. Then the producer ingests the configured streaming data (e.g. with predefined number and size) from the *message generator* and publishes them to a new topic. Finally, those messages are received by the consumer and the *statistical analysis tool* collects all the metrics for evaluation. From the timestamp data we obtain the end-to-end latency of every message and its distribution under the given configuration and operation condition. By comparing the unique keys of the received messages with those ingested from the *message generator*, the

number of lost and duplicated messages are determined. More insights and laws of these metrics are introduced in the next section.

The related resources of the testbed, including the link to the Docker image of Kafka brokers and the scripts to start a cluster are available from GitHub (https://github.com/woohan/kafka_start_up.git).

V. Reactive Batching Strategy

The purpose of the reactive batching strategy is to well configure the batch size parameters B_S and B_T given any operation condition, in order to achieve optimal performance while guaranteeing reliability. We use two reliability metrics, (i) the latency violation rate η_v: it represents the timeliness of messages. As explained in Section III-B, a system with most of its resources processing staled messages is unreliable and (ii) the message loss rate η_l: it indicates the consistency of messages. As mentioned in Section IV-C: message loss or duplication is detrimental to system reliability.

Since the reliability and performance requirements vary in different application scenarios it is necessary to introduce the operation conditions of our batching strategy. The conditions come in three aspects: streaming data rate, hardware resources and network quality. Considering those conditions, we discuss the scenarios when batching is necessary and introduce our methods for choosing the appropriate batch size reactively.

A. When is a batching strategy needed?

Streaming data is continuously generated by upstream applications, then ingested by a Kafka producer. Since the producer actively pulls messages from upstream applications, the ingest rate has its upper limit due to hardware limitations (e.g. network I/O or memory). When the producer reaches the maximum ingestion rate we call it fully loaded, which means it ingests the next message as soon as the last one is sent. It is worth noting that in the tests mentioned in Section III, the producer is under full load. We argue for the need to study the performance of fully loaded producers because this is when batch processing becomes beneficial and worth exploring in depth. Therefore, an essential premise of this study is that there is sufficient streaming data for the producer to continuously ingest messages.

Considering the stream processing when there is no batching, i.e. $B_S = 1$ and $B_T = 0ms$, sending a message can be either synchronous or asynchronous. In synchronous sending, the producer blocks and waits for a reply from the Kafka cluster after sending each message, therefore batching is not feasible. This is often applied when the processing order of streaming data is critical. However, it is inefficient to block the producer after each send operation, thus in most cases we use asynchronous sending. We observe that for the same streaming data (message size approximately 500 bytes), the throughput of a synchronously sending producer is 0.017 MB/s, while it is 17.73 MB/s under asynchronous sending. The distributions of the end-to-end latency L_K with these two sending modes are compared in Fig. 6. We can observe that the L_K under synchronous sending fluctuates in a much smaller interval, about $45ms$ to $75ms$. Sending messages asynchronously leads to a higher $\bar{L_K}$ due to L_K's dispersed distribution. Batch processing is feasible only under asynchronous sending, and it improves the throughput because sending more messages at once reduces the overhead per message. Thus another premise is that a producer sends messages asynchronously and exhausts hardware resources.

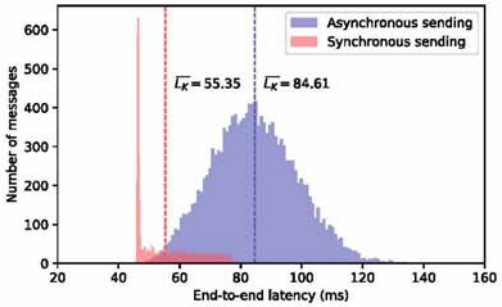

Fig. 6. The distributions of L_K under synchronous and asynchronous sending

B. Violation rate prediction

Our hypothetical ideal solution to the problems introduced in Section III is using a predictive model to estimate the latency violation rate. To better elaborate on this idea, we use the following equation to represent the predictive model.

$$\eta_v = f(\varepsilon_\star, B_S, B_T, \zeta_L) \tag{1}$$

The latency violation rate η_v is the proportion of messages that violate the user-defined latency constraint ζ_L. In addition to ζ_L, the other inputs are the factors that have effects on η_v. We use $\varepsilon_\star$ to denote all the network status' mentioned in Section IV-C. Imagine the application scenario where $\varepsilon_\star$ and ζ_L are known, then the impact of the batching strategy on η_v can be estimated via the model. However, generating such an ideal model is far from a trivial task. In Section III-B we have briefly discussed how the distribution of L_K changes over different batch sizes. A strong correlation is observed between $\{B_S, B_T\}$ and the shape of the distribution, including its uniformity, tendency to skew and its tail. We use $L_K \sim \Omega(\mu, \sigma)$ to denote the distribution of L_K, where Ω indicates that the distribution type is not fixed, and μ, σ denote the expectation and standard deviation, respectively. Thus given any latency constraint ζ_L, the violation rate can be obtained via the cumulative distribution function (CDF) $F_\Omega(x)$:

$$\eta_v = P(L_K > \zeta_L) = 1 - P(L_K \leq \zeta_L) = 1 - F_\Omega(\zeta_L) \tag{2}$$

Although the shape of the distribution varies greatly we can estimate the extreme values of L_K. The minimum value, denoted L_{Kmin}, can be roughly estimated as the minimum L_K observed with synchronous sending, denoted $min\{L_{sync}\}$. The reason is that there is no batching delay under synchronous sending, as introduced in the subsection above. To

estimate the maximum value of L_K, we consider the worst case that a message may encounter: it is the first message in a batch and waits until B_T is reached. Thus we can use $L_{Kmax} = max\{L_{sync}\} + B_T$ to estimate the maximum value, where $max\{L_{sync}\}$ represents the maximum L_K in synchronous sending. This implies how to coordinate B_S and B_T in our batching strategy: We use B_T to restrict the range of L_K's distribution, and configure B_S to obtain the desired η_v.

C. Latency violation rate prediction

The workflow of building the above prediction model is depicted in Fig. 7. A small set of the streaming data is required as the sample data in the tests. The timestamps of these messages are recorded and the end-to-end latencies are calculated by the statistical analysis tool. Then we record the configured batch size in each test as well as the network status. To simulate a good operation environment, the network conditions illustrated in Fig. 5 are emulated as $\varepsilon_p(30, 0)$, $\varepsilon_k(0, 0)$, $\varepsilon_c(30, 0)$. It should be pointed out that we ignore the network delay among Kafka brokers because it is negligible in a cluster (less than $1ms$). The *temporal batch size* B_T is configured as $1000ms$, the same as in the example in Section III-A.

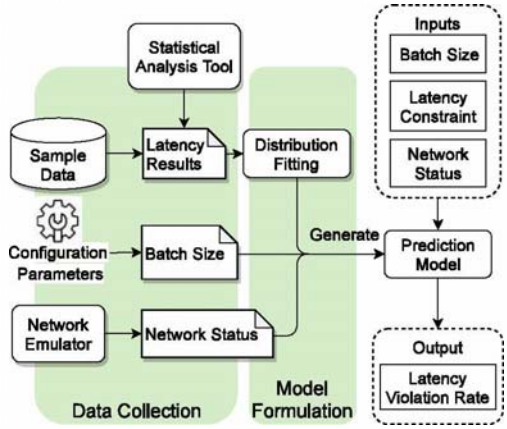

Fig. 7. The prediction model of latency violation rate η_v

An important and challenging aspect in the workflow is the distribution fitting. Ultimately, we need a method that provides a reasonably good fit based on little empirical information, such as moment estimates. Identifying an efficient method is left for future work, here we closely study the distributions of the collected data.

Due to the variety of the shapes, it is impossible to fit the same distribution to all data sets. We try the parametric method with the MATLAB distribution fitting tool and use the maximum likelihood estimation (MLE) to estimate the parameters. MLE is often applied in statistics to find the most likely values of distribution parameters for a sample of data, via maximizing the value of likelihood function. We use the log-likelihood value, denoted by θ, to assess the fit of the distribution to the experimental data. Although θ is not constrained to a certain range and may have any value, it can be used to compare the fit of multiple distributions to the same data. The distribution with higher θ is a better fit statistically. We select four distributions with the highest log-likelihoods and illustrate them in Fig. 8.

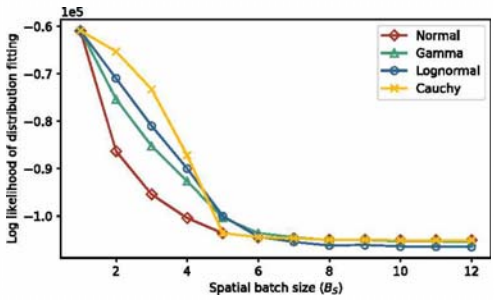

Fig. 8. The log-likelihood θ of different types of distributions

It can be observed that for any of the distributions, as B_S increases, θ decreases and stays flat after B_S reaches a certain value. This indicates that it becomes more difficult to fit the distribution of L_K as more messages are batched. By observing the histograms of the experimental data, we divide the shapes of distribution Ω into three stages depending on the effects of B_S and B_T, as mentioned in Section II-B:

1) The first stage is when there is no batching, i.e. $B_S = 1$.
2) The second stage is when B_S takes effect, which means B_S limits the maximum number of messages per batch.
3) The third stage is when B_S reaches its limit and B_T starts to take effect.

For the first stage, we use a Gamma distribution to fit the experimental data. The probability density function (PDF) of the fitted process is depicted in Fig. 9.

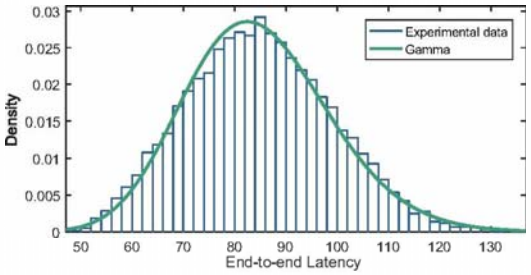

Fig. 9. The PDF of distribution fitting result when $B_S = 1$

From Fig. 10 we see that the curve of CDF fits the experimental data very well, thus η_v can be calculated with fairly high accuracy. We use mean absolute error (MAE) to measure the accuracy of prediction, calculated via the equation below:

$$MAE = \sum_{i=1}^{n} \frac{|\hat{\eta_v} - \eta_v|}{n} \tag{3}$$

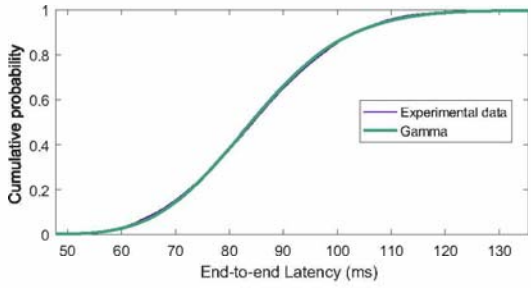

Fig. 10. The CDF of distribution fitting result when $B_S = 1$

We choose MAE because it intuitively reflects the proportion of messages that are incorrectly predicted to be violated or timely. In our experiment we apply the Gamma distribution to calculate the predicted violation rate $\hat{\eta_v}$ in the first stage, via Equation (4).

$$\eta_v = 1 - F_{\mathbf{\Omega}}(\zeta_L) = 1 - \frac{1}{b^a \Gamma(a)} \int_0^{\zeta_L} t^{a-1} e^{\frac{-t}{b}} dt \quad (4)$$

$\Gamma(a)$ is the Gamma function and the parameters are $a = 36.0582$ and $b = 2.35251$ respectively. Then, given the latency constraint $\zeta_L = 100ms$, we obtain $\hat{\eta_v} = 0.14$ with 95% confidence level, while $\eta_v = 0.1401$ in terms of the experimental data. The MAE is 0.0001, which indicates that only 0.01% of all messages' L_K are incorrectly predicted to be less than ζ_L.

In the second stage, where $2 \leq B_S \leq 10$, it is observed in Fig. 8 that the log-likelihood θ drops faster with Gamma and Lognormal distribution fitting. The tail of L_K's distribution becomes heavier as B_S increases. This is because more messages are waiting in the batch and they become the majority in the histogram. Although the fitting with Cauchy distribution performs best among all these heavy tail distributions, it is unable to capture the peak and tail with higher B_S. The prediction accuracy based on Cauchy distribution fitting decreases dramatically. For instance, if $\zeta_L = 700$, the MAE can be 0.06 while using Cauchy distribution, i.e. the L_K of 6% messages are overestimated to be timely.

To improve the prediction accuracy in the third stage, we use the fitting tool HyperStar2 to obtain the phase-type (PH) distribution of the experimental data, since any distribution with a strictly positive support in $(0, \infty)$ can be approximated arbitrarily close by a PH distribution [19], [20]. The PDF and CDF of the fitted process is a hyper-Erlang distribution, which is a subclass of PH distribution, and the parameters of the Erlang branches are shown in Table. I. We can observe from Fig. 11 and Fig. 12 that the curves fit well.

The MAE of predicting η_v using PH distribution is less than 0.005, which is less than one tenth of the prediction using Cauchy distribution. The predicted metric η_v can be estimated via the equation:

$$\hat{\eta_v} = 1 - F_{\mathbf{\Omega}}(\zeta_L) = \boldsymbol{\pi} e^{\zeta_L \boldsymbol{T}} \boldsymbol{I} \quad (5)$$

TABLE I
THE PARAMETERS OF ERLANG BRANCHES

Probability	Phase	Rate
0.173600	44	0.10376298
0.145600	148	0.20615273
0.192600	13	0.10828728
0.177733	18	0.06701713
0.150733	170	0.19943805
0.159733	72	0.12419047

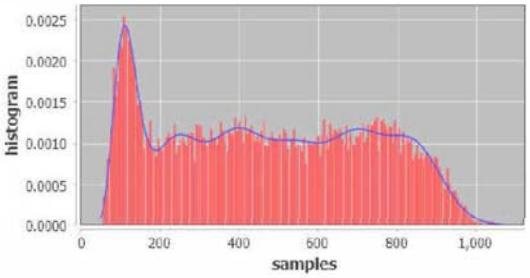

Fig. 11. The PDF of distribution fitting with HyperStar2 when $B_S = 7$

Here $\boldsymbol{\pi}$ is the initial probability vector, $\boldsymbol{T}$ is generator matrix of an absorbing Markov chain and $\boldsymbol{I}$ is the identity matrix.

In the third stage, the distribution hardly changes with the increase of B_S, as we discussed in Section III-A. We can simply use uniform distribution to fit the the experimental data in this stage and obtain high accuracy, and the CDF fitting result of one example ($B_S = 11$) is illustrated in Fig. 13.

We see that L_K is approximately continuously and uniformly distributed within a certain range, and the MAE is less than 0.01. The predicted violation rate η_v can be obtained via the equation:

$$\hat{\eta_v} = 1 - F_{\mathbf{\Omega}}(\zeta_L) = \frac{\alpha L_{Kmax} - \zeta_L}{\alpha L_{Kmax} - L_{Kmin}} \quad (6)$$

where α is a correction factor, and in this case we suggest $\alpha = 0.9$. Fitting phase-type distributions leads to very good results, but it is relatively expensive in terms of computation complexity. We will explore moment-matching methods for faster PH fitting in the future.

D. Timely throughput

Considering the case that a latency constraint ζ_L is given by the user, based on the prediction model studied above, the latency violation rate η_v can be obtained with specific $\{B_S, B_T\}$. However, this is still a few steps away from making proper decision in the batching strategy, because it is generally not clear to the users that how high the η_v is acceptable. Choosing the B_S with the lowest η_v can be meaningless because the throughput is sacrificed with smaller B_S. The correlation analysis between producer's throughput χ_P and B_S has been well explored in our previous work [11], thus in this paper we predict χ_P according to the proposed model.

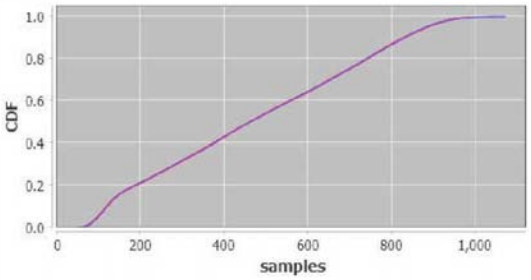

Fig. 12. The CDF of distribution fitting with HyperStar2 when $B_S = 7$

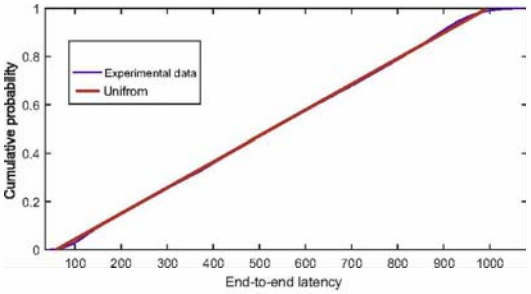

Fig. 13. The CDF of distribution fitting result when $B_S = 11$

Starting from the actual feature of messaging system, we propose *timely throughput* as the QoS metric of Kafka, which is generated from the equation below:

$$\chi'_P = (1 - \eta_v)\chi_P \tag{7}$$

This metric is used to measure the throughput of timely messages, whose end-to-end latencies are within the user-defined latency constraint. Thus the purpose of our batching strategy is maximizing χ'_P when resources are available. In Section VI we evaluate this metric in the tradeoff between L_K and χ_P and compare it with other empirical methods.

E. Message loss rate prediction

Upstream applications like mobile and IoT devices publish messages via a Kafka producer API and transfer the streaming data across wireless links. Thus network failures are common and could harm the success rate of message delivery. We inject faults to the network condition $\varepsilon_p(d_p, l_p)$ using the *network emulator* and observe the message loss rate η_l. In each test we send 1 million messages under poor network condition, and count the number of received messages according to the unique key introduced in Section IV-B. Thus η_l can be calculated as the proportion of lost messages. A Kafka producer provides at-most-once and at-least-once delivery semantics for sending messages. Under at-most-once delivery each message is sent only once and the producer does not need any response from brokers. An acknowledgement from the brokers is required with at-least-once delivery, and the producer retries sending when there is no response after waiting for a certain period. We choose to ignore message duplication because the impact is minor and most applications have an idempotence mechanism. From the experimental results we observe that injecting high packet loss rate l_p leads to high η_l, and the value of η_l is impacted by B_S under different semantics.

Due to the complexity of Kafka's architecture and network mechanism, we do not aim at building a model that would capture the system dynamics. Instead, we use machine learning techniques to build a prediction model for estimating η_l. The model can be expressed as the equation:

$$\hat{\eta_l} = f(l_p, acks, B_S) \tag{8}$$

The features for the prediction model are the packet loss rate l_p, the delivery semantics *acks*, and the *spatial batch size* B_S. Fig. 14 shows the workflow of building this prediction model. In the data collection process, we perform thousands of experiments with various values of these features and record their corresponding message loss rates as the training data for the model. Then we build an Artificial Neural Network (ANN) and train it with the collected experimental data.

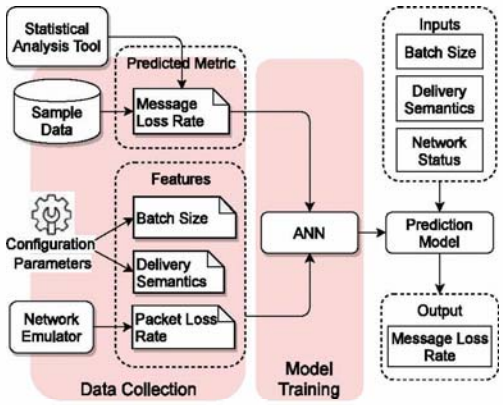

Fig. 14. The prediction model of message loss rate η_l

Since the input and output experiment data are strongly correlated, we can use a fairly standard ANN model to achieve considerable prediction accuracy. We apply the Stochastic Gradient Descent (SGD) optimizer in the ANN model, and build 4 hidden layers with $200, 200, 200$ and 64 neurons respectively. We set the learning rate to 0.5 and the number of epochs to 1000. The MAE of this prediction model is below 0.02, and it is accurate enough to help us compare the impact of different batch sizes B_S, as illustrated in Fig. 15.

The x-axis is the network packet loss rate ranging from 0% to 50%. The solid curves represent the predicted η_l under at-most-once delivery, while the dashed curves are the results under at-least-once delivery. These curves are grouped by the values of B_S, and the ones on the far left are configured with $B_S = 1$. As B_S increases, the predicted curve moves from left to right. This shows that Kafka tolerates a packet loss rate $l_p \leq 8\%$ due to the TCP retransmission mechanism. We observe the retransmissions on the Docker container network interface. The prediction model is more meaningful for the

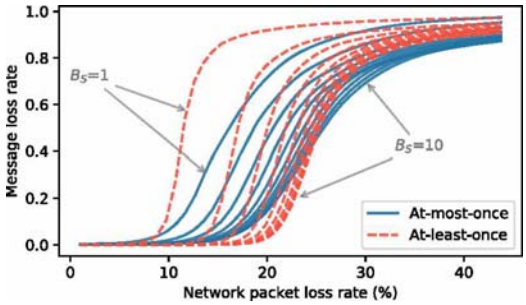

Fig. 15. The impact of delivery semantics, B_S and l_p on message loss rate

cases with $10\% \leq l_p \leq 25\%$ because changing the batch size B_S could have significant impact on the message loss rate. For instance when $l_p = 15\%$, increasing B_S from 1 to 3 under at-least-once delivery reduces the loss rate η_l from over 80% to less than 5%.

Using the prediction model of η_l, we are able to formulate the concept of *timely throughput* when encountering poor network conditions, which is an extension of Equation (7):

$$\chi'_P = (1 - \eta_v)(1 - \eta_l)\chi_P \tag{9}$$

VI. Experimental Evaluation

In order to evaluate the batching strategy in Section V we assume the following experimental scenario. Connected cars are the vehicles connected to wireless networks, and have become significantly important in the foreseeable IoT area [21]. The services of connected cars include traffic safety and cost efficiency and the demand for real-time processing is increasing [22]. In this experiment we use the Kafka producer to publish the vehicle sensor data in Berlin, which is derived from the public website of car sharing business [23]. Each message is generated in real-time and contains the current location of the car. We run a stream processor which receives the streaming data, and calculates the distance between the car and a specific charging or gas station. We conduct the experiment in both good and poor network conditions. The network delay follows a Pareto distribution to emulate real-world network status [24]. In the experiments with fault injection, we generate the network packet loss rate from the Gilbert-Elliot model, which is a two-state Markov model that has been widely applied to analyze measurements on wireless networks [25].

A. Latency and throughput tradeoff

We show the advantage of the proposed batching strategy by comparing it with the methods in other work. For instance, in [14] they just batch as much as possible in their streaming system as long as the $20ms$ latency constraint is guaranteed. This is a simple way to trade off latency and throughput. As the experimental results depicted in Fig. 16, both the mean end-to-end latency $\bar{L_K}$ and producer throughput χ_P rise as the B_S increases. Their theory is that given the latency constraint ζ_L, choosing the largest B_S within the constraint can maximize the throughput.

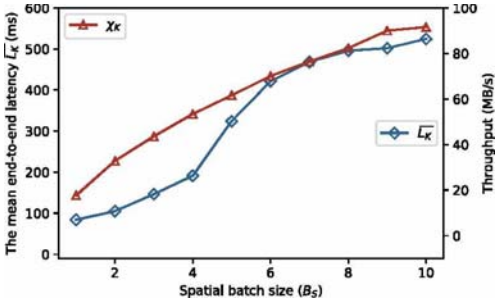

Fig. 16. The impact of B_S on $\bar{L_K}$ and χ_P

However, this method is not reliable because they measure only the mean value of L_K. To elaborate this problem, we plot the Pareto front of throughput and latency, as shown in Fig. 17. The y-axis is $1/\chi_P$, therefore lower values are preferred to the higher ones. Given the requirements that $\zeta_L = 350ms$, and the throughput to be higher than 50MB/s, this Pareto front based empirical method suggests the user to configure $B_S = 5$.

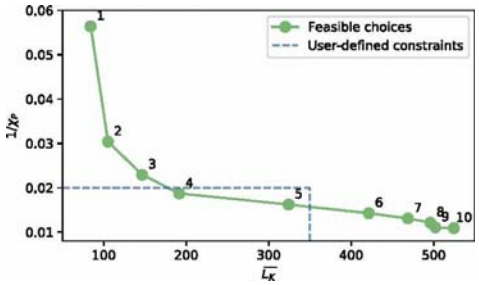

Fig. 17. The Pareto font of throughput vs latency

With our batching strategy, the value of B_S that maximizes the timely throughput χ'_P is configured, and it is $B_S = 4$, according to the results illustrated in Fig. 18. This configuration improves the throughput of timely messages by 7.37MB/s, compared to the choice above.

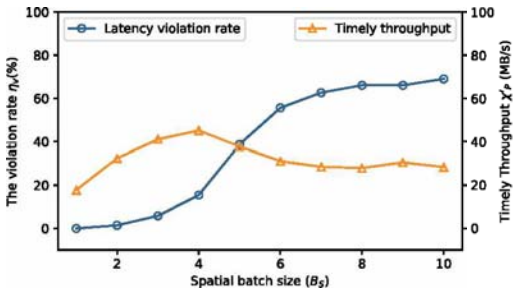

Fig. 18. The impact of B_S on η_v and χ'_P

TABLE II
COMPARISON OF MESSAGE LOSS RATE AND TIMELY THROUGHPUT

	Reactive batching	Empirical method	$B_S = 1$	$B_S = 10$
η_l	18.69%	12.63%	71.0%	3.9%
χ'_P	42.7MB/s	33.0MB/s	5.14MB/s	27.3MB/s

B. Message success rate evaluation

In this experiment with network fault injection we assume that network status is known as depicted in Fig. 19. We check the network metrics $\varepsilon_p(d_p, l_p)$ every 60 seconds and reconfigure the batch size when necessary. This is because changing configuration parameters too frequently will cause additional coordination overhead due to shuffling communication among producer and brokers [26]. We substitute the operation and configuration parameters to Equation (9) and observe how the predicted result changes over different B_S. Then we reconfigure B_S to obtain the maximum χ'_P.

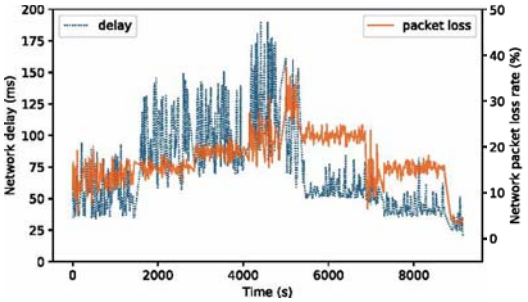

Fig. 19. Network status $\varepsilon_p(d_p, l_p)$

The overall message loss rate and timely throughput in this experiment are illustrated in the Table II. We compare the results obtained via our reactive batching strategy with those using Pareto front based empirical methods, and fix $B_S = 1$, $B_S = 10$. It is observed that the reactive batching outperforms others in terms of timely throughput χ'_P.

It is important to note that these characteristics are specific to the Kafka testbed and the workload we use. Docker containers offer lower latency costs and lower variability than hardware virtualization [27]. Therefore the absolute number of those metrics are best ignored when other researchers try to apply our method to other distributed systems. The main takeaways for Kafka users are the workflows for predicting the reliability metrics in Section V, which can be carried out on their operation environments. The new tradeoff between throughput and latency using the QoS metric timely throughput is another outcome that other Kafka users can apply in their systems. The users may follow the idea of using timely throughput as Kafka's performance optimization goal in real-time systems. For other similar applications, with comparable batching mechanisms, the insights we obtained (E.g. larger batches lead to higher latency violation rate but lower losses) can be applied too.

VII. Related Work

Stream processing systems are taking the place of traditional big data systems, and numerous research works have been done over the past decade. Recent stream processing frameworks focus on the state management and fault tolerance, like Samza and Flink, which both use Apache Kafka for distributed messaging [5], [6]. Compared to the conventional message brokers based on Advanced Message Queueing Protocol (AMQP), Kafka shows rather higher performance when processing massive messages [12]. It is able to handle the streaming data load on the highest traffic hour (over 172,000 messages per second) at LinkedIn [10].

A reactive strategy to enforce constraints over average latencies in scalable Stream Processing Engines (SPEs) is proposed in [14]. The work in [28] indicates that choosing the appropriate buffer size can significantly reduce the mean latency in Nephele, a stream processing framework. Adaptive batch size control algorithm can guarantee the average end-to-end latency as the situation necessitates [13] . In [14] they evaluate the mean latency within a finite time interval (e.g. 10 s). However, using the mean value of latency as the performance metric cannot evaluate system performance comprehensively. In [29] they use elastic scaling to optimize the utilization under certain latency constraints, and the number of latency violations are studied. In the latency-aware scheduling of an extended Hadoop architecture, both the average and the quantile of latency are considered [30].

Batching strategy is commonly studied in stream processing systems. The configured size of the batch significantly impacts the performance, and needs to be carefully treated in latency and throughput trade-off [14]. The impacts of batch size on the performance of Apache Spark has been explored in [13]. Changing batch sizes reactively can speed up long pipelines by hours in [31].

VIII. Conclusion

In this work, we explore how to batch messages properly in Apache Kafka in order to meet real-time requirements. We discuss the limitations of existing batching methods, and use experimental data to illustrate their unreliability. In order to collect experimental data efficiently, we build a Docker testbed of Kafka with fault injection components. We explore the correlation between batch size and the distribution of end-to-end latency. Using distribution fitting techniques, we propose a model to predict the latency violation rate under various operation conditions, given batch size and latency constraint. The message loss rate with poor network status is predicted by an ANN model. We use a newly defined metric, the timely throughput, as the optimization objective in our batching strategy. In the end we compare the batching strategy with those normal methods, and the experimental results confirm the superiority of our new method. For future work we plan to reduce the overhead of the prediction model formulation process by using moment matching to fit a distribution. More variable real-world workloads with multiple client scenarios will be evaluated with our batching strategy.

REFERENCES

[1] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *The VLDB Journal*, vol. 27, no. 6, pp. 847–872, 2018.

[2] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–44, 2013.

[3] C. Doulkeridis and K. NØrvåg, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, 2014.

[4] A. Abdallah, M. A. Maarof, and A. Zainal, "Fraud detection system: A survey," *Journal of Network and Computer Applications*, vol. 68, pp. 90–113, 2016.

[5] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 399–410.

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

[16] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Securestreams: A reactive middleware framework for secure data stream processing," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 2017, pp. 124–133.

[17] 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.

[18] 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.

[19] Z. Shang, T. Meng, and K. Wolter, "Hyperstar2: Easy distribution **fitting of correlated data," in** *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 139–142.

[20] P. Buchholz, J. Kriege, and I. Felko, *Input modeling with phase-type distributions and Markov models: theory and applications*. Springer, 2014.

[21] R. Coppola and M. Morisio, "Connected car: technologies, issues, future trends," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, pp. 1–36, 2016.

[22] L. Zhu, F. R. Yu, Y. Wang, B. Ning, and T. Tang, "Big data analytics in intelligent transportation systems: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 1, pp. 383–398, 2018.

[23] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting iot platform requirements with open pub/sub solutions," *Annals of Telecommunications*, vol. 72, no. 1-2, pp. 41–52, 2017.

[24] 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.

[25] 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.

[26] 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.

[27] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *2016 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2016, pp. 1–7.

[28] B. Lohrmann, D. Warneke, and O. Kao, "Nephele streaming: stream processing under qos constraints at scale," *Cluster computing*, vol. 17, no. 1, pp. 61–78, 2014.

[29] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 2014, pp. 13–22.

[30] B. Li, Y. Diao, and P. Shenoy, "Supporting scalable analytics with latency constraints," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1166–1177, 2015.

[31] 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.