Dissertation

zur Erlangung des akademischen Grades
des Doktors der Naturwissenschaften

(Dr. rer. nat)

Performance and Reliability Evaluation of Apache Kafka Messaging System

eingereicht
am Institut für Informatik
des Fachbereichs Mathematik und Informatik
der Freien Universität Berlin
von

Han Wu

Berlin, 2020

Gutachter:

Prof. Dr. Katinka Wolter
(der erste Rezensent)
Department of Computer Science
Freie Universität Berlin, Germany

Prof. Dr. Aad van Moorsel
(der zweite Rezensent)
School of Computing
Newcastle University, UK

Das Datum der Disputation: 14th. Jan 2021

Selbständigkeitserklärung

Ich versichere, dass ich die Doktorarbeit selbständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit hat keiner anderen Prüfungsbehörde vorgelegen.

Han Wu

Berlin, den 4. Februar 2021

Abstract

Streaming data is now flowing across various devices and applications around us. This type of data means any unbounded, ever growing, infinite data set which is continuously generated by all kinds of sources. Examples include sensor data transmitted among different Internet of Things (IoT) devices, user activity records collected on websites and payment requests sent from mobile devices. In many application scenarios, streaming data needs to be processed in real-time because its value can be futile over time. A variety of stream processing systems have been developed in the last decade and are evolving to address rising challenges.

A typical stream processing system consists of multiple processing nodes in the topology of a DAG (directed acyclic graph). To build real-time streaming data pipelines across 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. It ingests streaming data from upstream applications and store the data in its distributed cluster, which provides a fault-tolerant data source for stream processors. Therefore, Kafka plays a critical role to ensure the completeness, correctness and timeliness of streaming data delivery.

However, it is impossible to meet all the user requirements in real-time cases with a simple and fixed data delivery strategy. In this thesis, we address the challenge of choosing a proper configuration to guarantee both performance and reliability of Kafka for complex streaming application scenarios. We investigate the features that have an impact on the performance and reliability metrics. We propose a queueing based prediction model to predict the performance metrics, including producer throughput and packet latency of Kafka. We define two reliability metrics, the probability of message loss and the probability of message duplication. We create an ANN model to predict these metrics given unstable network metrics like network delay and packet loss rate. To collect sufficient training data we build a Docker-based Kafka testbed with a fault injection module. We use a new quality-of-service metric, timely throughput to help us choosing proper batch size in Kafka.

Based on this metric, we propose a dynamic configuration method, which reactively guarantees both performance and reliability of Kafka under complex operation conditions.

Zusammenfassung

Streaming-Anwendungen sind heute sehr verbreitet und viele Informationen erreichen uns durch Datenfluss über verschiedenste Geräte. Die Menge der Daten die von verschiedenen Arten an Quellen generiert wird wächst kontinuierlich. Bekannte Beispiele sind Sensordaten, die zwischen den verschiedenen Geräten des IoT ausgetauscht werden, auf Websites gesammelte Benutzeraktivitätsaufzeichnungen und von Mobilgeräten gesendete Zahlungsanforderungen. In vielen Anwendungsszenarien müssen die Streaming-Daten in Echtzeit verarbeitet werden, was bedeutet, dass sie wertlos werden, wenn die Verzögerung in der Datenübertragung zu groß ist. In den letzten zehn Jahren wurde eine Vielzahl von Stream-Verarbeitungssystemen entwickelt, die sich den wachsenden Herausforderungen stellen.

Ein typisches Stream-Verarbeitungssystem besteht aus mehreren Verarbeitungsknoten in der Topologie eines DAG (directed acyclic graph). Um streaming pipelines für Echtzeit-Daten über diese Knoten zu legen, wird häufig eine Middleware-Technologie für Nachrichten angewendet. Als verteiltes Nachrichtensystem mit hoher Haltbarkeit und Skalierbarkeit ist Apache Kafka bei modernen Unternehmen sehr beliebt geworden. Es nimmt Streaming-Daten von Upstream-Anwendungen auf und speichert die Daten in seinem verteilten Cluster, der eine fehlertolerante Datenquelle für Stream-Prozessoren darstellt. Daher spielt Kafka eine entscheidende Rolle, um die Vollständigkeit, Korrektheit und Pünktlichkeit der Bereitstellung von Streaming-Daten sicherzustellen.

Es ist jedoch unmöglich, alle Benutzeranforderungen in Echtzeitfällen mit einer einfachen und festen Datenlieferungsstrategie zu erfüllen. In dieser Arbeit befassen wir uns mit der Herausforderung, eine geeignete Konfiguration auszuwählen, um sowohl die Leistung als auch die Zuverlässigkeit von Kafka für komplexe Streaming-Anwendungsszenarien zu gewährleisten. Wir untersuchen die Funktionen, die sich auf die Leistungs- und Zuverlässigkeitsmetriken auswirken. Wir schlagen ein auf Warteschlangen basierendes Vorhersagemodell vor, um die Leistungsmetriken vorherzusagen, einschließlich des Produzentendurchsatzes und der Paketlatenz von Kafka. Wir definieren zwei Zuver-

lässigkeitsmetriken, die Wahrscheinlichkeit eines Nachrichtenverlusts und die Wahrscheinlichkeit einer Nachrichtenduplizierung. Wir erstellen ein ANN-Modell, um diese Metriken bei instabilen Netzwerkmetriken wie Netzwerkverzögerung und Paketverlustrate vorherzusagen. Um ausreichende Trainingsdaten zu sammeln, bauen wir ein Docker-basiertes Kafka-Testfeld mit einem Fehlerinjektionsmodul auf. Wir verwenden eine neue Metrik für die Servicequalität und einen zeitnahen Durchsatz, um die richtige Batch-Größe in Kafka auswählen zu können. Basierend auf dieser Metrik schlagen wir eine dynamische Konfigurationsmethode vor, die sowohl die Leistung als auch die Zuverlässigkeit von Kafka unter komplexen Betriebsbedingungen reaktiv garantiert.

Acknowledgements

My study is financially supported by China Scholarship Council (CSC) and has been carried out at Dependable Distributed Systems (DDS) group at Freie Universität Berlin in Germany. I am very grateful to CSC and DDS for providing me such a valuable opportunity to do research. In 2020 we encountered the pandemic caused by COVID-2019, and everything has changed as well as my work style. I would like to express my sincere gratitude to all the people who contributed to the completion of this thesis during this difficult period.

First and foremost, I would like to thank my supervisor Prof. Dr. Katinka Wolter for her guidance and help. At the beginning of my PhD study, I am very confused with my research topic and made very little progress. Prof. Wolter was very patient with me and provided valuable feedback in my research. She offered me the opportunity to attend a top conference in the U.S., where I met various research people with innovative ideas. She checked my paper word by word before submitting, and taught me how to write research paper. Without her precious suggestions and encouragement, this thesis would have never been accomplished.

Thanks to all my colleagues who are very nice and friendly. I really enjoyed working with them and sometimes we party together. I wish to thank Guang Peng, Xiao Jia, Dr. Tianhui Meng and Dr. Zhihao Shang for their help and interesting ideas. I also want to say thank you to Yuanwei Pan, Suqiong Zhou and Dr. Hao Ren, who shared those nice holidays with me.

Last but not least, I am very grateful for the love and support of my parents. They comforted and encouraged me when I met difficulties.

Contents

Chapter 1

Introduction

In this chapter, we discuss the general problems in real-time streaming systems, and describe the research challenges with Apache Kafka. Then we introduce the main contributions and list the outline of this thesis.

1.1 Problem Statement

For the last decade, we lived through the *Big Data Era* where the computational power and the complexity of data are continuously growing [41, 95, 119]. In a technical report from Cisco [54], it is predicted that the global Internet traffic in 2021 will be equivalent to 127 times the volume of the entire global Internet in 2005 [34]. Nowadays, many certain types of data such as stock values, financial transactions, and traffic conditions rapidly depreciate in value if not processed instantly. For instance, an ideal fraud detection system should identify a fraudulent transaction before it completes, thus avoiding financial losses [1]. This type of data is called *big streaming data*, which means any unbounded, ever growing, infinite dataset that is continuously generated by various sources, and must be processed under very short delays [36].

The characteristics of *big streaming data* are described as the 4V's, i.e., *volume*, *variety*, *velocity* and *veracity* [40, 92, 101]. A new fleet of stream processing systems are developed to tackle the 4V's, including Apache Samza, Apache Flink, Google's MillWheel, Twitter's Heron [4, 26, 69, 84], etc. Despite some functional differences, these systems commonly consist of multiple processing nodes in the topology of a DAG (directed acyclic graph). Apache Kafka is an open-source distributed messaging system for transferring data among those processing nodes [68, 72]. 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 [43]. Due to its high durability and scalability, Kafka has become very popular among modern companies like Uber, Netflix, Twitter and Spotify [11].

Kafka's high scalability allows it to handle large quantity of data (*volume*), and its publish/subscribe pattern is designed for processing different types of data (*variety*). As the driving force of a real-time system, *velocity* is one of the main subjects in this thesis. Streaming data is time-sensitive in scenarios such as real-time healthcare monitoring, fraud detection, online machine learning and decision making [5,81,115]. Latency and throughput are two commonly used metrics to evaluate the performance of Kafka in real-time, when it is desired to deliver huge volumes of data in milliseconds or seconds.

Veracity refers to the accuracy or truthfulness of the data. Since Kafka is responsible for transferring data from one place to another in stream processing, its veracity is reflected in the completeness and correctness of streaming data. Losing some data or delivering replicated data is detrimental to this veracity. The uncertainty of hardware resources and unstable network status are the inducements of unreliable streaming data delivery, which may in the end lead to faulty outcomes of stream processing. The ideal goal is that each piece of data is delivered to its destination exactly as it is generated, without any data loss or duplication. In general, a reliable Kafka messaging system with high performance is critical for applications to process *big streaming data* in real-time.

1.2 Main Research Challenges

While plentiful experiments have been performed for the evaluation of Kafka's performance, its robustness and reliability against real-time requirements have not been well explored [39, 43, 58]. To achieve the goal of a real-time messaging system with high performance and reliability, the challenges come from four facets, as illustrated in Fig. 1.1.

- **Big streaming data:** The variety of data types, sizes, arrival rates and sources make it difficult to apply a general stream processing method for all cases. To evaluate Kafka's quality-of-service (QoS) for different data streams, users define experienced non-functional properties like latency and throughput [114]. The optimization priorities of these different QoS metrics depend on the data contents and the functions of downstream applications. For instance, when applying Kafka to process the stream of invoices in retail business, low latency is highly prioritized because customers are waiting for the check-out processes. Meanwhile, for the data analysis applications that make decisions based on a whole day's data, the latency requirement is not as strict as those on the throughput. Therefore the first main challenge is the uncertainty

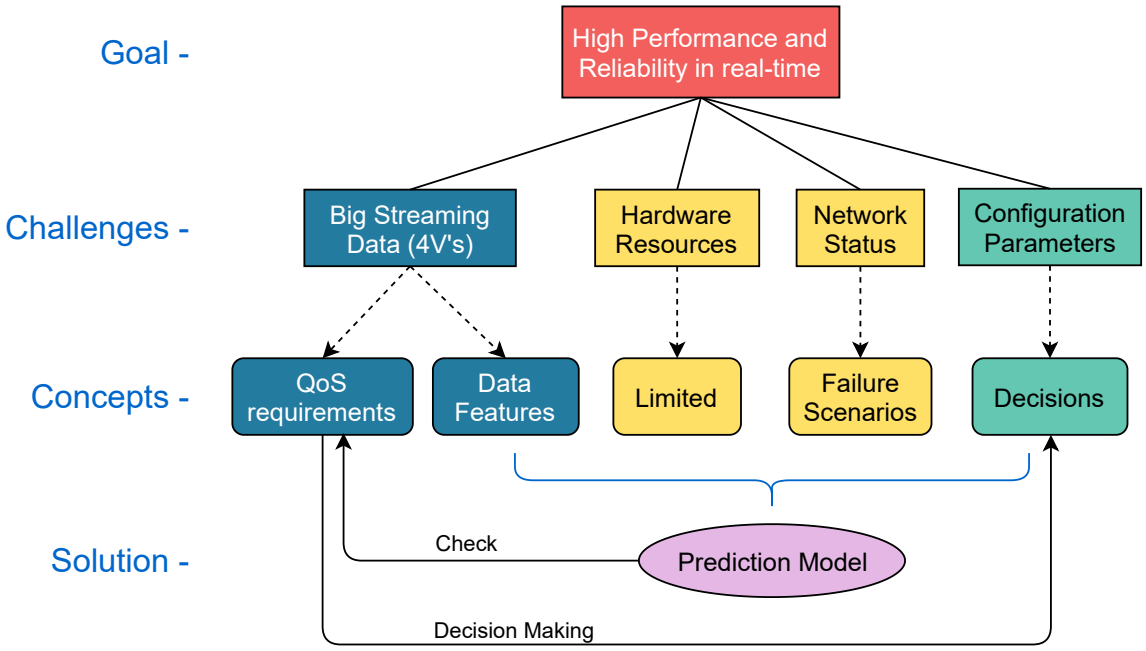

Figure 1.1: The research challenges and solution in this thesis

of these user requirements, as the content of big streaming data varies from application to application.

- **Hardware Resources:** The hardware resources directly affect the performance, i.e. the speed that Kafka writes data into the disks of distributed machines. Furthermore, many cloud vendors provide *Kafka as a service* for users who want to build and run applications that use Apache Kafka to process data. Kafka as a service is a cloud computing service model similar to Infrastructures as a Service (IaaS), where basic computing resources like CPUs and storage are provisioned over the Internet. Users generally pay according to the length of use, while the cloud vendor manages and maintains the Kafka cluster. In this thesis, for simplicity, we assume the hardware resources are fixed, as depicted in Fig. 1.1.
- **Network Status:** The quality of the network condition among the components of Kafka is a challenge for the reliability of data delivery. Unstable network connection often occurs among numerous IoT sensors, mobile devices or autonomous vehicles [6, 60, 66]. Through our experiments, we found that high network delay or packet loss may cause streaming data loss or duplication, which is a critical impairment to the reliability of data delivery. The incorrect or incomplete streaming data published by Kafka can finally lead to errors in the

Table 1.1: Number of configuration parameters in Apache Kafka

Configuration Type	Number of Parameters
Broker Configs	208
Topic Configs	27
Producer Configs	65
Consumer Configs	70

results of downstream applications.

- **Configuration Parameters:** Considering the challenges above, Kafka is developed as a very flexible system with various configuration parameters. Changing configuration parameters can significantly impact the performance of Kafka. In the industry, 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, as depicted in Table. 1.1. The configuration and fine-tuning remains a big challenge for using Kafka to meet the real-time requirements in big streaming data processing.

1.3 Contributions

The purpose of this thesis is to achieve efficient and reliable real-time streaming data delivery with Apache Kafka. Assuming that the hardware resources of the Kafka cluster are fixed, then given any type of streaming data, our target is to configure Kafka properly to meet real-time processing requirements. The key to solving this problem is to predict Kafka's performance and reliability given various configuration and operation conditions. Then with the proposed prediction model, users are able to check if the current configuration meets their QoS requirements and tune the parameters accordingly, as shown in Fig. 1.1. Following this line of reasoning, the main contributions of our work are listed below.

- **Novel metrics:** We define several novel metrics for the evaluation of Kafka's performance and reliability. In stream processing systems, the *mean end-to-end latency* is generally used for the measurement of its real-time performance [75]. In this thesis we point out the problems with this metric. Its measurement is not dependable or comprehensive in many cases. Therefore we propose a new performance metric, the *latency violation rate*, which is the proportion of streaming data that exceeds user-defined *latency constraint*. Accordingly, we improve the

measurement of throughput by introducing the *timely throughput*. To evaluate the reliability of streaming data delivery, 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 [62]. One metric is the *message loss rate*, which we use to measure the probability of loss while sending a piece of streaming data. The other metric is the *message duplication rate* and it indicates the probability that the sending duplicates a message.

- **Flexible Kafka testbed based on Docker:** Our solution needs to collect a vast array of the metrics mentioned above, as well as the configuration parameters and features of operation conditions. Furthermore, numerous tests are necessary for generating sufficient analysable data and accurate results. Thus we create a Kafka testbed on Docker containers for exploring the approach of building prediction models [21]. We choose the industrial-grade lightweight virtualization technology for a couple of reasons. It guarantees the correctness of Kafka's configuration, within a lightweight, standalone and reproducible execution environment [48]. Each component of Kafka is a running Docker container which embeds all the dependencies and codes. Therefore it is fast and easy to start, shut down and scale up or down a Kafka cluster with our testbed. We add a fault injection module using NetEm, a popular Linux based network emulator [59]. With this module we are able to study the effects of unstable network connection on Kafka's reliability. The flexibility of the testbed makes it easy for other researchers to reproduce and follow our work. More details about the testbed is available from our published paper [110].
- **Performance prediction:** We propose a queueing based packet flow model to predict performance metrics of Kafka [113]. The input configuration parameters of this model contain the number of brokers in a Kafka cluster, the number of partitions in a topic and the batch size of messages. We use queueing theory to evaluate the end-to-end latency of packets. In the experimental validation we see a strong correlation between packet size and packet send interval, and the service time of packets fits a phase-type distribution. The correlation and fitting results are substituted to the essential constants in the model. Experiments are performed with various configurations for observing their effects on performance metrics. The results show that our model achieves high accuracy in predicting throughput and latency. Users can tune the configuration parameters in the model to observe the parameters' impact on Kafka's performance metrics including the producer throughput, the relative payload and overhead and the change of disk storage usage over time. This can help users to utilize the limited resources provided by cloud vendors more economically and give explicit advice on the configuration settings of Kafka.

- **Reliability prediction:** Due to the complexity of Kafka's architecture and the diversity of its configuration parameters, we use machine learning techniques to build the reliability prediction model [111]. 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.
- **Reactive batching:** In order to handle various requirements from all kinds of streaming applications, we present a weighted QoS metric to help choose proper configurations for Kafka [112]. In this metric both of the proposed performance metrics and reliability metrics are evaluated. When the configurations as well as the streaming data type and network status are known, we can generate the current QoS metric 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 QoS metric. Among all the parameters we find the impact of *batch size* is the most significant. Then a reactive batching strategy is proposed in our experiments to evaluate the effectiveness of the prediction models.

1.4 Thesis Outline

In this chapter the problems and research challenges are addressed. The remaining part of the thesis is structured as follows:

Chapter 2 describes the characteristics of steaming data, some popular streaming data processing systems. The role of Kafka in the stream processing pipeline is explained in detail. We introduce existing research efforts in performance and reliability evaluation of streaming systems.

Chapter 3 introduces the mathematical theory used in this thesis, including the concept of Phase-type distribution, the derivation of an M/PH/1 queue, and the workflow of a typical ANN model.

In Chapter 4 we evaluate the performance of Apache Kafka by proposing a queueing based prediction model. We study the correlation among various configuration parameters and the performance metrics.

Chapter 5 emphasizes the importance of reliable delivery. In order to study Kafka's reliability in failure scenarios, we present a flexible Kafka testbed built upon Docker containers. Two reliability

metrics, the probability of message loss and the probability of message duplication are introduced. Then we use an ANN based prediction model for predicting these two metrics. Through the lessons on the reliability issues we learned, a few main takeaways are listed for Kafka users.

Chapter 6 shows the problems of current latency measurement methods. Then we explored the effects of batching on the latency and throughput of Apache Kafka. By studying the distribution of end-to-end latency, we introduce a new Quality-of-Service (QoS) metric, the timely throughput. A reactive batching strategy is proposed for optimising this new metric under unstable network condition.

Chapter 7 concludes the main contributions in this thesis, and describes some future research directions.

Chapter 2

Messaging in Stream Processing

We introduce the background of stream processing in this chapter. Through the background introduction we provide a comprehensive overview of current stream processing systems. Other researchers may find interesting topics or challenges in stream processing from this chapter.

2.1 The Context of Stream Processing

In this section we describe the landscape of streaming data processing systems and their general pipeline.

2.1.1 An Overview of Streaming Technologies

Modern data-driven organizations need to handle and process streaming data in real-time as they occur and extract an actionable insight as quickly as possible. Traditional batch processing systems such as MapReduce [37] and Hive [100] suffer from latency problems because they need to collect input data into a dataset before processing them. Extracting useful insights from the continuous and unbounded data streams is challenging, especially under the application scenarios like fraud detection in financial trading and healthcare analytics based on IoT devices [36].

Some examples of streaming data sets are depicted in Fig. 2.1. The car trace data is the streaming json-data generated by the vehicle sensors and includes the information of cars and their location. The weather trace data is also in json format and collected from the weather sensors inside a building where the temperature and humidity information are monitored. The third example is retrieved from point of sale (POS) terminals in Extensible Markup Language (XML) format (tags excluded), which contains the transaction information at the checkout of supermarkets. The streaming data is

generated from large amounts of nodes all the time, and grows to huge volume for the processing systems. Early systems for big data processing could only load data from servers into a data warehouse for offline analysis. For instance, Hadoop is a commonly used framework for the distributed processing of large data sets across clusters using simple programming models [10]. It processes large volumes of data that was previously stored in the Hadoop Distributed File System (HDFS). However, in the cases that new data keeps arriving at the storage, this approach becomes inefficient.

Car trace data:

```
325 1;hpp/cars/B-GO2136;{"address":"Barstraße 10, 10713 Berlin","coordinates":[13.
    31384,52.48752,0],"engineType":"CE","exterior":"UNACCEPTABLE","fuel":15,
    "interior":"UNACCEPTABLE","name":"B-GO2136","smartPhoneRequired":false,
    "vin":"WME4513341K580877"}
326 1;hpp/cars/B-GO3239;{"address":"Gasteiner Straße 2, 10717 Berlin","coordinates":
    [13.32586,52.48752,0],"engineType":"CE","exterior":"UNACCEPTABLE","fuel":21,
    "interior":"GOOD","name":"B-GO3239","smartPhoneRequired":false,
    "vin":"WME4513341K541581"}
327 1;hpp/cars/B-GO2946;{"address":"Untertürkheimer Straße 27, 12277 Berlin",
    "coordinates":[13.38336,52.42701,0],"engineType":"CE","exterior":"GOOD",
    "fuel":60,"interior":"GOOD","name":"B-GO2946","smartPhoneRequired":false,
    "vin":"WME4513341K565087"}
328 1;hpp/cars/B-GO2943;{"address":"Friedenstraße 84 - 91, 10249 Berlin (Umkreis
    100m)","coordinates":[13.43662,52.5211,0],"engineType":"CE","exterior":"GOOD",
    "fuel":66,"interior":"UNACCEPTABLE","name":"B-GO2943",
```

Weather trace data:

```
456 0.015;hpp/weather/Node262;{"voltage":3.00,"light_par":114.41,"light_tsr":6.19,
    "temperature":28.02,"humidity":29.73}
457 0.015;hpp/weather/Node262;{"voltage":3.00,"light_par":114.41,"light_tsr":6.48,
    "temperature":27.99,"humidity":29.77}
458 0.015;hpp/weather/Node262;{"voltage":3.00,"light_par":114.41,"light_tsr":6.48,
    "temperature":28.02,"humidity":29.73}
459 0.015;hpp/weather/Node262;{"voltage":3.00,"light_par":114.41,"light_tsr":6.76,
    "temperature":27.99,"humidity":29.77}
460 0.015;hpp/weather/Node262;{"voltage":3.00,"light_par":114.41,"light_tsr":6.76,
    "temperature":28.01,"humidity":29.73}
```

Transaction data from POS machines:

```
918 1,1902191060433,"2019-02-19T07:43:28","2019-02-19T07:44:05",101,37,23,5,FALSE,
    TRUE,51.82
919 1,1902191060435,"2019-02-19T07:44:28","2019-02-19T07:45:03",101,35,23,8,TRUE,
    FALSE,12.6
920 1,1902191060436,"2019-02-19T07:45:26","2019-02-19T07:46:16",101,50,24,9,TRUE,
    FALSE,48.77
921 1,1902191060437,"2019-02-19T07:46:40","2019-02-19T07:48:16",101,96,18,25,FALSE,
    TRUE,316.91
```

Figure 2.1: Some examples of streaming data

Consequently, various streaming data processing systems (SDPSs) arise for ingesting, processing, storing and managing streaming data, such as Apache Samza, Apache Flink, Google's MillWheel, Twitter's Heron [4, 26, 69, 84], etc. Some of the SDPSs are marketed by large software vendors

while others are open-source projects. These SDPSs are often compared from different aspects in recent studies [55, 91, 101]. We list the most popular open-source technologies according to their capabilities in the streaming data processing pipeline, as illustrated in Fig. 2.2.

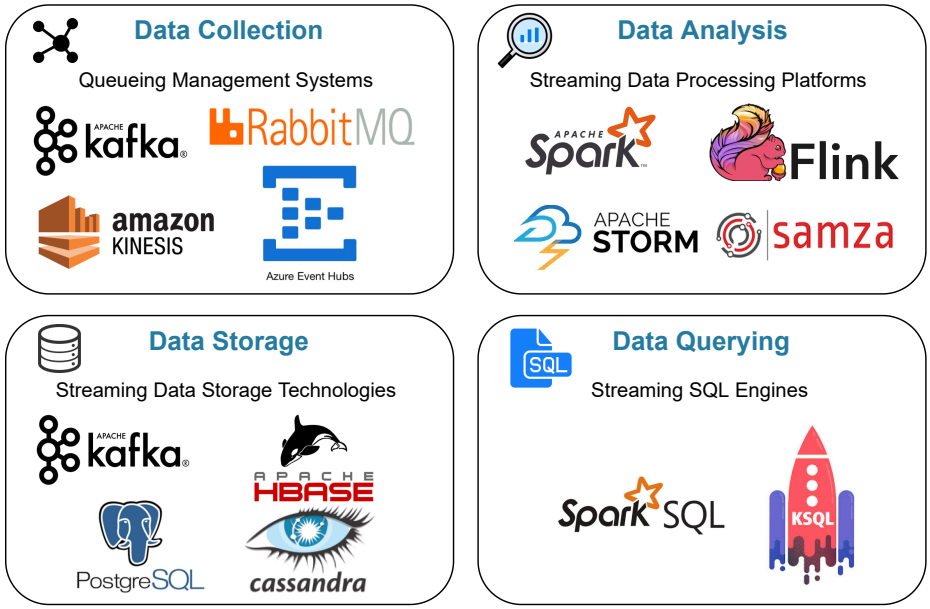

Figure 2.2: Open-source streaming data processing technologies

- **Data Collection:** Distributed queueing management technologies such as Kafka, RabbitMQ, Amazon Kinesis and Microsoft Event Hubs are applied for ingesting streaming data from disparate upstream applications, and feeding it to real-time downstream applications. These technologies support publish/subscribe messaging for transferring large-scale data across many real-time applications. Compared to the traditional queueing systems (e.g. Redis) which remove data after processing, distributed queueing technologies persist data on disks over replicated clusters, and provide consumer groups feature for taking multiple independent actions on the same event. The technologies in this step are of the utmost necessity for making streaming data accessible to data consumers in real-time.
- **Data Analysis:** Traditional batch processing technologies like Hadoop and Spark are inspired by Google's MapReduce design and have been the state-of-art big data platforms. To overcome the inefficiency problems, Apache Spark divides the stream of events into small batches to keep the processing latency under control. However, the performance of these traditional technologies is insufficient for building real-time applications. New technologies, including

Apache Storm, Apache Samza and Apache Flink have matured in the last few years to handle large amounts of data on the fly. They are designed to improve the speed of extracting insights from millions of stream events. The capabilities and features of these popular open-source technologies are depicted in Table. 2.1, among which Kafka Streams is part of the Kafka ecosystem.

- **Data Storing:** The big data storage technologies are mainly three types according to the data model: (i) *File System*, e.g., data in Hadoop Distributed File Systems for Hive Hadoop; (ii) *Document-based*, e.g., MangoDB; (iii) *Column-based*, e.g., Cassandra and Hbase. As per the CAP theorem, only two of the three features, Consistency, Availability and Partition Tolerance can be guaranteed for a distributed data store [23]. Therefore the choice of a proper storage technology highly depends on the specific application.
- **Data Querying:** Streaming platforms provide SQL (Structured Query Language) capabilities by windowing operations, which means repeatedly iterating over a series of micro-batches. Based on the concept of different windowing types, various open-source stream query engines are developed, including Spark SQL, Samaza SQL, Storm SQL and KSQL (Kafka SQL). In real-time stream processing, there are four types of windowing mechanisms: *Trumbling Window*, *Sliding Window*, *Hopping Window* and *Session Window*. Choosing the proper window is crucial for processing infinite streams, and KSQL supports all except the *Sliding Window*.

2.1.2 Comparison of Open-Source Stream Data Processing Systems

From the overview above we see various kinds of streaming data processing systems. We select the most popular open-source SDPSs depending on their wide range of applications and introduce them in detail for comparison.

- **Apache Storm:** Storm is an open-source project developed under the Apache License [14]. It is a distributed real-time stream processing computation framework for processing large volumes of high-velocity data. The topologies of a Storm application are composed of multiple components arranged in a directed acyclic graph (DAG). The components include spouts and bolts, as depicted in Fig. 2.3. A spout reads data from various sources like the Twitter API, and emits the data as streams. A bolt receives these streams as input, performs operations (e.g. filter, aggregate or join) on the data, and possibly emits new streams. The edges in Fig. 2.3 indicate which bolts subscribe to which streams. The data model used in Storm is called tuple, which is a named list of values. In case of failures, Storm relies on a framework called acking to track the completion of each tuple. This mechanism checkpoints the state of

bolt periodically and if the system crashes or restarts, the bolt will be initialised to its previous state.

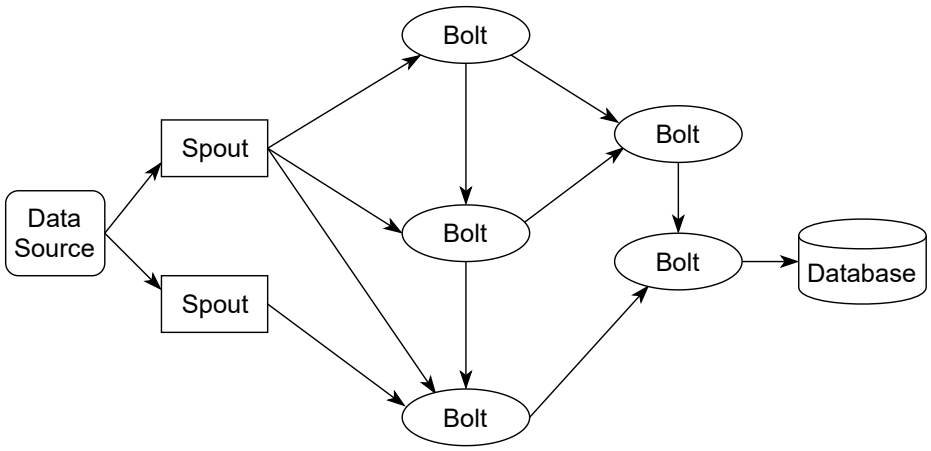

Figure 2.3: The topology of an Apache Storm application

- **Apache Spark Streaming:** Spark Streaming is an extension of Apache Spark, a unified analytics engine for large-scale data processing [13]. Spark Streaming enables scalable and real-time stream processing with high throughput. It ingests data from many sources like Kafka and Amazon Kinesis, and processes the data with high-level functions like *map*, *reduce* and *join*. The results are pushed to file systems, databases or live dashboards. The key abstraction in Spark Streaming is the discretised stream (DStream), which is a stream of data divided into small batches, called Resilient Distributed Datasets (RDDs). RDDs are distributed, fault-tolerant data sets that can be processed in parallel using any number of functions in clusters. Since version 2.3, Spark has introduced the Structured Streaming library, built on the Spark SQL engine [15]. The streaming data is processed in a new low-latency mode called *Continuous Processing*, and it is recommended for developing new streaming applications with Spark [55].
- **Apache Samza:** Apache Samza was originally developed in conjunction with Apache Kafka by LinkedIn [12]. Samza enables the implementation of stateful applications that process data in almost real-time from multiple sources including Kafka. The Samza application is built on the concept of streams and jobs. A stream is composed of immutable messages of a similar type or category, corresponding to the concept of topic in Kafka, which is introduced in Section 2.2. A job is the code that consumes input streams, performs a logical transformation and then produces the results as output streams. Samza breaks a job into multiple

tasks to achieve parallel processing. The architecture of Samza is composed of three layers, as shown in Fig. 2.4. The streaming layer provides partitioned streams that are replicated and durable, which is generally implemented by a Kafka system. The execution layer coordinates tasks across the machines, also called Samza containers. The processing layer applies logical transformations on the input streams.

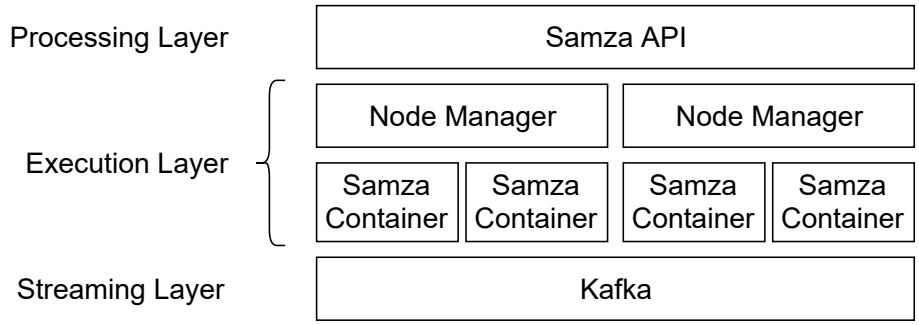

Figure 2.4: The architecture of an Apache Storm application

- **Apache Flink:** Apache Flink is a distributed processing framework for stateful operations on streaming and batch data [9]. Similar to the other SDPSs, Flink's basic building blocks include streams and transformation operators. Flink mainly provides three APIs for data processing, a DataStream API, a DataSet API and a Table API. The DataStream API is used to perform various operations (e.g. filter, aggregate, window functions) on bounded or unbounded streams of data. The DataSet API enables batch operations on bounded datasets. The Table API is a relational API with SQL-like expression language. Flink does not provide its own data storage system, instead it uses durable message queues (e.g. Apache Kafka or Amazon Kinesis) to replay data streams. The task states are maintained in memory or access-efficient on-disk data structures, as depicted in Fig. 2.5. In case of failures, Flink guarantees state consistency by checkpointing the local state to durable storage periodically and asynchronously.
- **Kafka Streams:** It should be noted that Kafka Streams is a stream processing library of Apache Kafka, thus it is part of the overall Kafka ecosystem. The abstraction of stream is similar to the other SDPSs. Kafka Streams uses the processor topology to define the stream processing computational logic for an application. As shown in Fig. 2.6, a topology is a graph of stream processors connected by streams. A source processor consumes messages from its subscribing Kafka topic, and produces an input stream to its topology. A sink processor sends the stream to a specific Kafka topic. The processed results can either be streamed back to Kafka or external applications. Kafka Streams provides two APIs, the Domain Specific

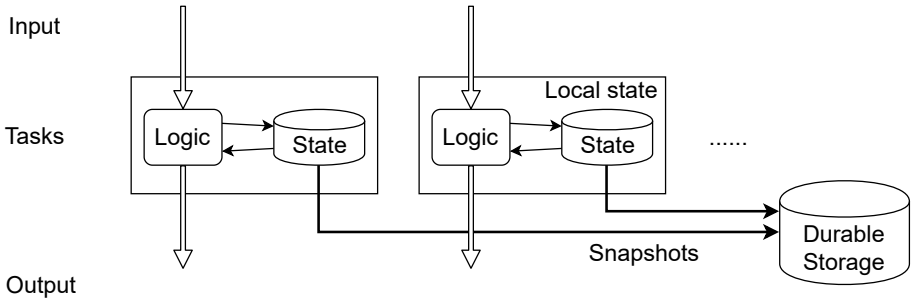

Figure 2.5: State management of Apache Flink

Language (DSL) API for high-level operations (e.g. map, filter, join) and the Processor API for the definition and connection of custom processors. In case of failure, Kafka Streams leverage the fault-tolerance capability of Kafka to restart tasks.

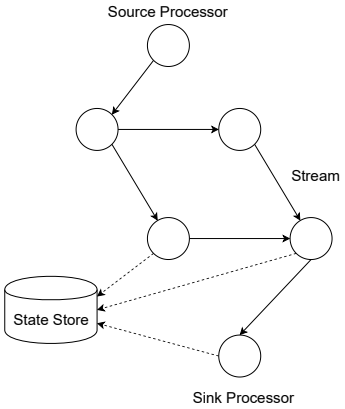

Figure 2.6: The topology of Kafka Streams

A summary of the comparison of these open-source SDPSs is listed in Table. 2.1. We know that most stream processing platforms support hybrid stream and batch processing, although they differ in architecture components. The delivery semantics is an important feature for guaranteeing message delivery in stream processing, and can be divided into three types:

- **At-most-once:** Each message is sent by the producer at most once. It is the easiest to implement because the producer does not need any response, thus any system is capable of this semantic. However, the disadvantage is that there is no measure to prevent message loss. In the table we list the other two semantics which provide more fault tolerance.

Table 2.1: Comparison of open-source distributed stream processing platforms

	Storm	**Spark**	**Samza**	**Flink**	**Kafka Streams**
Version	2.0	2.4.3	1.5.0	1.8	2.3.0
Components	Streams, Spouts, Bolts	Resilient Distributed Dataset	Samza, YARN, Kafka	Operators, Sources, Sinks	Kafka
Processing Model	Hybrid Stream and Batch	Hybrid Stream and Batch	Stream	Hybrid Stream and Batch	Hybrid Stream and Batch
Delivery Semantics	At-least-once, Exactly-once	At-least-once, Exactly-once	At-least-once	Exactly-once	At-least-once, Exactly-once
State Management	Key-value storage, Redis	Write-ahead log	Key-value storage	Key-value storage	Key-value storage, Hash map
Fault Tolerance	Stream replay, Checkpoint	Checkpoint	Stream replay, Checkpoint	Stream replay, Checkpoint	Stream replay
Queueing Management Systems	Kafka, RabbitMQ, Kinesis, Event Hubs	Kafka, RabbitMQ, Kinesis, Event Hubs, Google Pub/Sub	Kafka, RabbitMQ, Kinesis, Event Hubs	Kafka, RabbitMQ, Kinesis, Event Hubs	Kafka

- **At-least-once:** For each message that has been sent, a response from the receiver side is required, otherwise the producer keeps resending until it gets the response. It guarantees each message to be delivered, but may be delivered multiple times in some cases.
- **Exactly-once:** This is the most ideal delivery as each message is guaranteed to be delivered once only, not more or less. Implementing this semantic requires additional resources to support transactions or idempotent processing.

From the comparison of different streaming data processing systems, we know that Kafka plays a critical role in ingesting data from upstream applications and providing scalable data storage for stream processing. In this thesis, we choose to study the performance and reliability of streaming data delivery using Kafka instead of other popular streaming systems because of three reasons: First, some comparable message broker technologies like RabbitMQ, ActiveMQ and other enterprise messaging systems are ephemeral, which means they store data in memories or other light storages. On the other hand, Kafka provides durability by persisting data on disks, and this makes its application scenarios more extensive and comprehensive. Second, other streaming platforms such as Spark, Storm and Flink are all designed to provide computational framework for streaming data processing. However, Kafka is not a data processing framework, but instead a data transporting tool. This unique feature establishes its position in stream processing. The third reason is that Kafka is often applied in combination with other streaming data processing systems, as we observe from the last row of Table 2.1. The performance and reliability of Kafka is crucial to those streaming systems because it is responsible for the data transporting. Therefore, the mechanisms and models we introduce in this thesis (batching mechanism, delivery semantics) are highly available in these streaming data processing systems.

2.1.3 Stream Processing Pipeline

The pipeline of streaming data processing starts from the data ingest, where the distributed queueing management technologies are applied. The applications that generate source data for ingestion are called upstream applications. Then the streaming data will be published to the corresponding topics for further processing. The downstream applications, or streaming applications subscribe to the topics of their interest to extract useful insights. The pipeline of those downstream applications can be described by a directed acyclic graph (DAG), which consists of multiple stream processor nodes. Each processor performs an operation on the input data and transfers the outputs to the next processor. As depicted in Fig. 2.7, stream processors A and C consume messages from Kafka cluster, then produce the results to stream processor B and D. In some scenarios stream processor B may

publish the final results for other processors to subscribe.

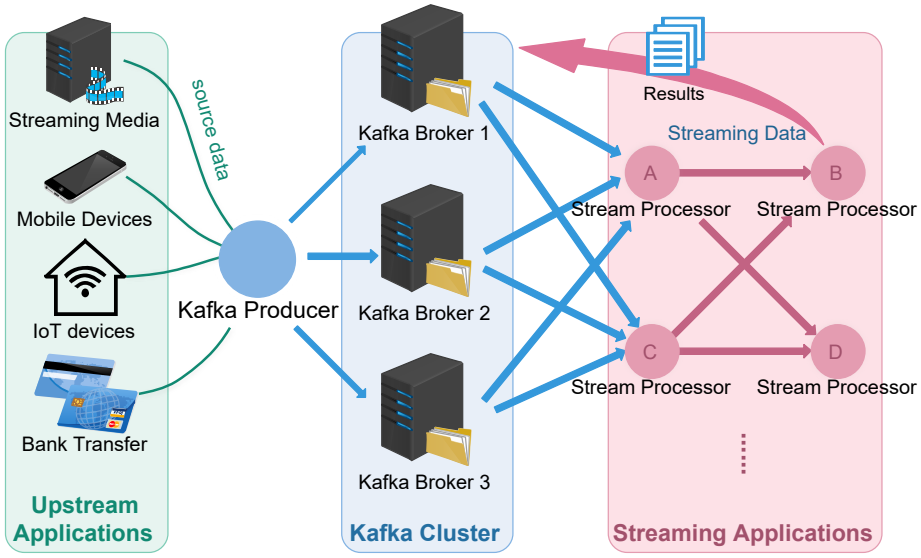

Figure 2.7: Kafka in stream processing pipeline

Depending on the demand for state management, stream processors can be divided into two categories, stateless and stateful processors. Here state refers to the intermediate value of a specific computation that will be used in subsequent operations during the processing of streaming data [101]. Stateless processors do not need to maintain any state, which means the operation on a message is independent from the operations on other messages. For instance, when ingesting one raw message from Kafka, the processor only needs to transform it and pass the result to other processors. Simple map operation on one message at a time, and filtering out messages based on some condition are the most common examples of stateless processing.

However, most processors require state to generate correct results. A stream processor is stateful whenever it needs to operate on a sequence of input messages, and potentially use state information to generate meaningful results. Examples include sorting, reducing, joining and aggregation. The advantage of state management is that a processor can resume from wherever it was paused or stopped, thus avoiding reprocessing of huge volumes. As depicted in Table. 2.1, Apache Storm provides state persistence using in-memory state storage and a Redis backed implementation, while Apache Spark maintains a write-ahead log to help the system recover from failure. Kafka provides state stores, such as RocksDB database or in-memory hash map, to store state data for later reference.

The failures in a stream processing pipeline happen due to an unstable network connection, software bugs, hardware issues like node crash and bottlenecks caused by the volume and speed of

incoming data [55]. These failures may cause data loss, thus leading to incorrect results and decisions. Failure recovery in stream processing demands additional resource and repeated processing, which is another overhead [27]. For real-time streaming applications, reprocessing from the starting point of the pipeline is impractical. An ideal system should restore itself to a previous state when the failure has not happened yet, in order to obtain minimum overhead. Apache Storm periodically checkpoints the state of the bolt, and replays tuples to the previous state in case of failures. Apache Flink also restores the state of its operators and replays the vents from the checkpoint. Kafka Streams utilizes the native fault-tolerance capability within the Kafka core, and this will be introduced in the next section.

Fig. 2.7 also demonstrates the role that Apache Kafka plays in the stream processing pipeline. Upstream applications, such as streaming media, mobile and IoT devices generate the original data streams. Kafka producer is responsible for ingesting the streams from these data sources and then publish them to the Kafka cluster, where the streaming data is stored across multiple Kafka brokers. Then downstream applications fetch the streaming data from the Kafka cluster with the Kafka consumer API. The architecture details and advantages of this publish/subscribe pattern is elaborated in the next section.

2.2 Apache Kafka: A Real-time Distributed Messaging System

The applications in a streaming system may be running on multiple computers, or different platforms and geographically dispersed. Therefore transferring data across these applications in real-time requires some critical criteria. Strong coupling of applications requires more time to develop, deploy and maintain the dependencies. Decoupling the sending and receiving applications can minimize the dependencies. Decoupling with an intermediate translator also improves the flexibility to unify the different data formats from applications. Establishing a synchronous connection between every two applications is unreliable as the applications or network may be temporarily unavailable. Therefore using a reliable middleware to facilitate asynchronous communication between the applications is a critical criterion. Then the sender application can continue processing without waiting for the receiver to acknowledge receipt of the message.

Kafka serves as a messaging system to manage streaming data exchange in the stream processing ecosystem. The publish/subscribe pattern is an ideal solution for real-time data integration against the criteria mentioned above. Other alternatives, such as Remote Procedure Call (RPC) and shared databases have drawbacks which violate the criteria. E.g. RPC is used to execute some commands on another application from a different machine and this approach creates a tight coupling among

the applications. A shared database is where the streaming data are stored, and all the applications write to and read from the same database. However, due to the diversity of streaming applications, it is impractical to design a unified schema which meets all the applications. Besides, multiple applications modifying the same data in the shared database greatly reduces the performance.

Apache Kafka provides a publish/subscribe messaging service, where a producer (publisher) sends messages to a Kafka topic in the Kafka cluster (message brokers), and a consumer (subscriber) reads messages from the subscribed topic. A topic is a logical category of messages, for instance the producer will send the logs of a web server access records to the $serverLogs$ topic, while the records of the querying records on a website will be sent to the $searches$ topic. As depicted in Fig. 2.8, we observe 2 producers sending messages to 2 topics in this Kafka cluster, topic A and topic B respectively.

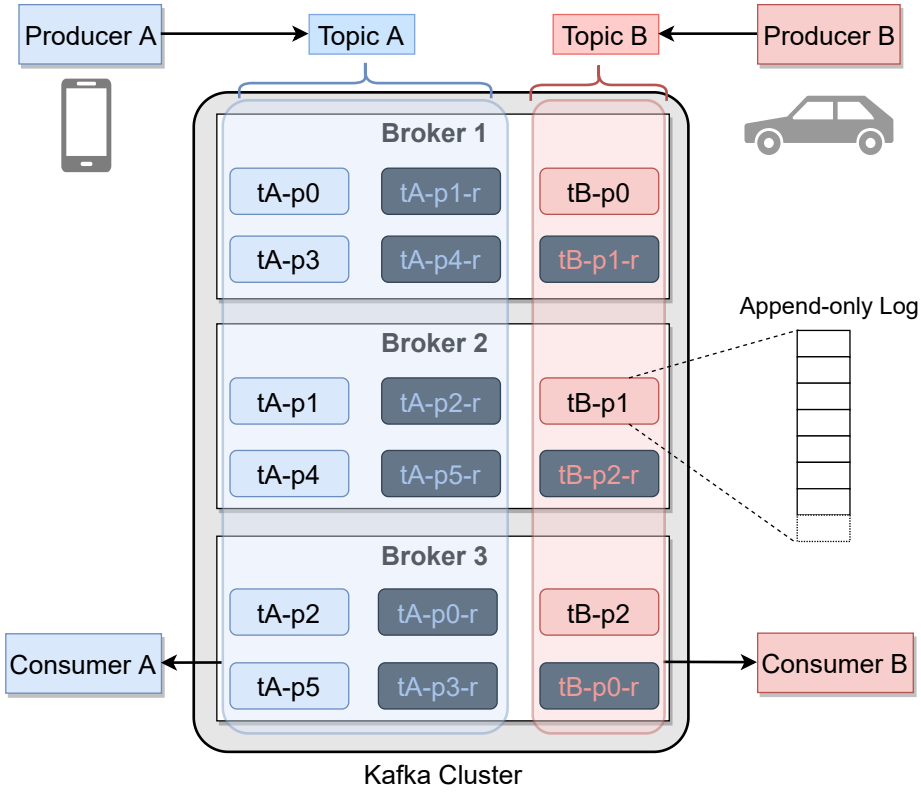

Figure 2.8: Kafka in stream processing pipeline

Any topic may be stored in one or more partitions, which are the physical storage of messages in the Kafka cluster. The Kafka cluster consists of several brokers (Kafka servers), and all the partitions

of the same topic are distributed among the brokers. In the example in Fig. 2.8 there are 3 brokers, and we see topic A consisting of 6 partitions (denoted by tA-p{0~5}) while topic B with 3 partitions (denoted by tB-p{0~2}).

Each partition is physically stored on disks as a series of segment files that are written in an append-only manner. It is replicated across the Kafka broker nodes for fault tolerance, and we denote the replica of a partition with the suffix -r in Fig. 2.8. The messages of the leader partition tA-p0 on broker1 are replicated to the partition tA-p0-r on broker3, so once the broker1 server fails, tA-p0-r is chosen as the leader partition, which is similar for other partitions on broker1. Only the leader partition handles all reads and writes of messages with producer and consumer, which is performed in FIFO manner. Kafka uses partitions to scale a topic across many servers for producers to write messages in parallel, and also to facilitate parallel reading of consumers.

We see 2 consumers in our example, consumer A and consumer B subscribing to topic A and topic B respectively. These two consumers work independently and no coordination between them is required. Inside each consumer, there can be multiple consumer instances, which will in parallel fetch messages from a different subset of the partitions in the topic. A consumer instance can fetch messages from multiple partitions, while one partition has to be consumed by only one consumer instance. As a result the number of partitions controls the maximum parallelism of the consumer group, and it is obvious that the number of consumer instances should not exceed the number of partitions in the subscribed topic, otherwise there will be idle consumer instances.

2.3 Related Work

In this section we introduce the related work on stream processing from three facets, the research on the performance of stream processing platforms, the study on the reliability problems and the work on batching strategy.

2.3.1 Performance Evaluation

Plentiful experiments have been done to compare the performance of Apache Kafka and traditional message brokers like RabbitMQ, which is primarily known and used as an efficient and scalable implementation of the Advanced Message Queuing Protocol (AMQP). The results indicate that both systems are capable of processing messages with low-latency, and increasing the number of partitions in Kafka can significantly improve its throughput, while increasing the producer/channel count in RabbitMQ can only improve its performance moderately [39]. The Kafka protocol is more suitable for an application that needs to process massive messages, but if messages are im-

portant and security is a primary concern, AMQP is more reliable [58]. Despite those horizontal comparisons, how to set the configuration parameters properly inside Apache Kafka, which is also the major concern of the users who purchase Kafka cloud services, still remains challenging. In [61] they build a novel framework for benchmarking streaming engines include Apache Storm, Apache Spark and Apache Flink.

Researchers have developed various tools to help the users of cloud services make appropriate decisions in pursuance of better performance with limited resources. CloudSim [25] is a well-known simulation framework which supports simulation tests across three major cloud service models (e.g. SaaS, PaaS, and IaaS) and it is capable of cloud simulations including VM allocation and provisioning, energy consumption, network management and federated clouds. CloudCmp [73] systematically compares the performance and cost of four different cloud vendors to guide users in selecting the best-performing provider for their applications. PICS (public IaaS cloud simulator) [64] provides the capabilities for accurately evaluating the cloud cost, resource usage and job deadline satisfaction rate. All those simulators aim at evaluating the public clouds without running numerous tests on real clouds, and focus more on power consumption and resource management.

However, from the perspective of users, none of the simulators can address their concerns because the particular mechanisms of Kafka are not considered. The users need a tool to help them choose proper configuration parameters in Kafka and this motivated us to conduct this research. We build a queueing model to evaluate the performance of Kafka, referring to other related works. The application processed at the cloud data centers has been modeled as an infinite $M/G/m/m + r$ queueing system in [63]. A queueing model has been proposed in [93] for elastic cloud apps where the service stage involves a load balancer and multiple workers running in parallel. The analytic model in [94] can evaluate simple cloud applications using message queueing as a service (MaaS), but still neglects the essential features in Kafka.

We build the tool TRAK using Docker containers. Docker is a popular emerging virtualization technology on operating system level, which is similar to a lightweight virtual machine [21]. A Docker container is piece of a software that packages up all code, runtime, system library and dependencies, and runs fast and reliably when moved from one environment to another. Containers are less resource and time consuming, and rising as an important part of microservices and the cloud computing infrastructure [57]. For fault injection we use a network emulation tool for Docker, Pumba, which is used to tune the delay and packet loss of the network between the Docker containers [71].

2.3.2 Reliability Evaluation

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 [29] 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 [69]. To achieve exactly-once semantics, additional computing resources and data stores are required to support transactions, thus imposing a performance penalty.

Performance prediction models have been studied for different contexts in various systems [103]. 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 [38, 88, 99]. In this thesis 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 [49] 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 [29] 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 [35]. The research in [44] also indicates that dynamic configuration is efficient in streaming systems.

2.3.3 Batching Strategy

A reactive strategy to enforce constraints over average latencies in scalable Stream Processing Engines (SPEs) is proposed in [75]. The work in [76] 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 [35] . In [75] 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 [50] 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 [74].

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 [75]. The impacts of batch size on the performance of Apache Spark has been explored in [35]. Changing batch sizes reactively can speed up long pipelines by hours in [29].

Chapter 3

Theoretical Background

3.1 Queueing Theory

Queueing theory is usually used for the performance evaluation of computer systems or networks. Using this theory we can study the stochastic systems in stream processing and the mean time it takes to process a job. In this section we introduce the background of queueing theory applied in this thesis, including exponential distribution, phase-type distribution and the $M/PH/1$ queue.

3.1.1 Exponential Distribution

The exponential distribution is a continuous distribution commonly used to measure the time interval between events. The probability density function (PDF) of X is given by:

$$f(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases} \tag{3.1}$$

where λ is the rate parameter. Some plots of the PDF with different values of λ are depicted in Fig. 3.1. Correspondingly, the cumulatve distribution function (CDF) of an exponential distribution is:

$$F(x) = \begin{cases} 1 - e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases} \tag{3.2}$$

The CDFs of some exponential distribution are shown in Fig. 3.2:

The mean value of X is an estimator for the expectation, which can be obtained by:

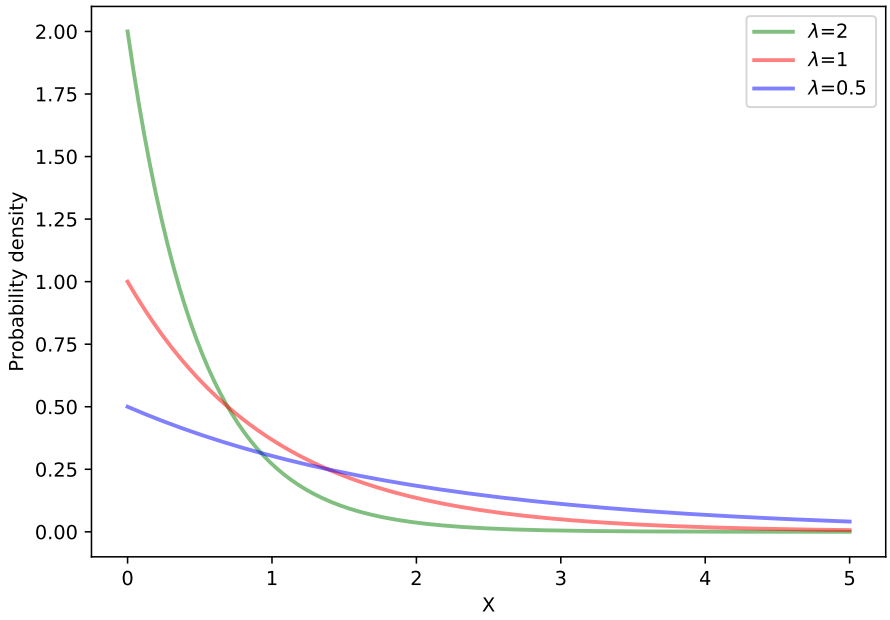

Figure 3.1: PDFs for some exponential distributions

$$E[X] = \frac{1}{\lambda} \tag{3.3}$$

The variance of X is:

$$Var[X] = \frac{1}{\lambda^2} \tag{3.4}$$

The moments of X is:

$$E[X^n] = \frac{n!}{\lambda^n} \tag{3.5}$$

The exponential distribution is one of the basic distribution types in queueing theory because of its memoryless property. This means that for any random variable X which is exponentially distributed, the past history of X plays no role in predicting its future. For instance, let X be the time that a job spends in a server, also called *service time*, which is exponentially distributed. Then the probability that the job is finished at a future time t is independent of how long this job has already been in the server.

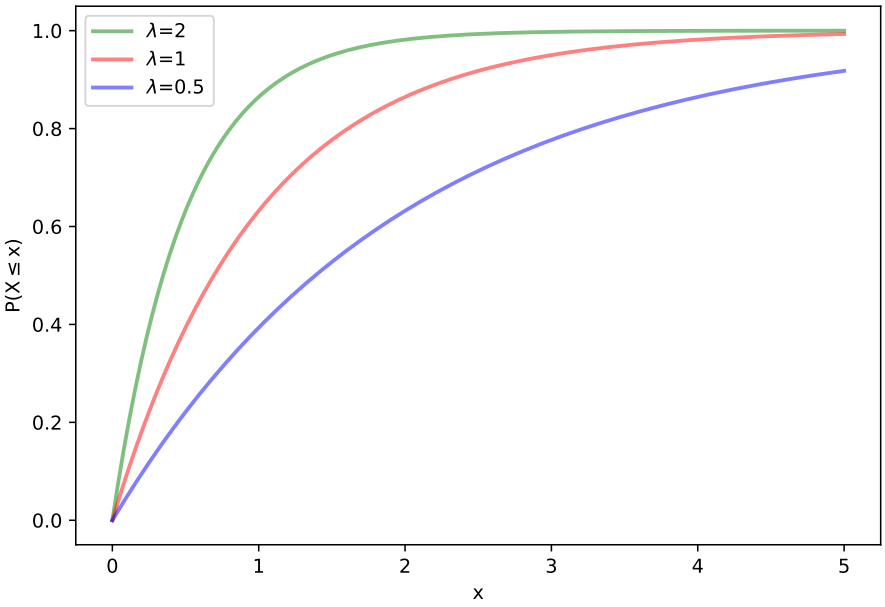

Figure 3.2: CDFs for some exponential distributions

3.1.2 Phase-type Distribution

Although the exponential distribution is widely used in performance evaluation, sometimes it is not adequate to model certain processes. Instead, Phase-type (PH) distributions become very popular for modelling the service time of computer systems [22]. The reason is that any distribution with a strictly positive support in $(0, \infty)$ can be approximated arbitrarily close by a PH distribution [24]. PH distributions are able to fit various experimental data into one distribution.

A PH distribution is defined as the distribution of the time to absorption in a continuous-time Markov Chain (CTMC) with one absorbing state [82]. A CTMC is used to describe the stochastic process with multiple states, and Fig. 3.3 illustrates the example of a CTMC with one absorbing state. Starting from state k with probability $\boldsymbol{\pi}_k$, the process stays in the state for a random period which follows an exponential distribution. Then it moves to another state with the probability as specified by a stochastic matrix. The time it takes for a process to reach the absorbing state (state 4 in Fig. 3.3) is a sum of samples from exponential distributions.

We use a tuple $(\boldsymbol{\pi}, \boldsymbol{D})$ to denote a PH distribution, where $\boldsymbol{\pi}$ is the initial probability vector that represents the probability of initial state. $\boldsymbol{D}$ is the generator matrix of the absorbing CTMC that represents the transition rates among states.

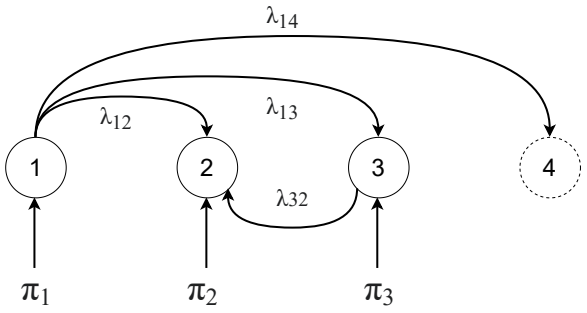

Figure 3.3: A CTMC with one absorbing state

The $(\boldsymbol{\pi}, \boldsymbol{D})$ is a Markovian representation of a PH distribution when:

$$\boldsymbol{\pi} = (\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \cdots \boldsymbol{\pi}_m) \in \mathbb{R}^m, \tag{3.6}$$

$$\boldsymbol{\pi e} = 1, \tag{3.7}$$

$$\boldsymbol{\pi} \geq 0, \tag{3.8}$$

and the generator matrix:

$$\boldsymbol{D} = \begin{pmatrix} \lambda_{11} & \cdots & \lambda_{1n} \\ \vdots & \ddots & \vdots \\ \lambda_{n1} & \cdots & \lambda_{nn} \end{pmatrix} \in \mathbb{R}^{m \times m} \tag{3.9}$$

is a non-singular matrix with:

$$\lambda_{ii} < 0, \tag{3.10}$$

$$\lambda_{ij} > 0 \quad \text{where} \quad i \neq j, \tag{3.11}$$

and

$$\boldsymbol{De} \leq 0, \tag{3.12}$$

$$\sum \boldsymbol{De} < 0, \tag{3.13}$$

where $\boldsymbol{e}$ is a column vector with all elements equal to one. For a CTMC with $m + 1$ states, the

size of $\boldsymbol{D}$ is $m \times m$. The generator matrix of embedded CTMC is

$$\hat{\boldsymbol{D}} = \begin{pmatrix} \boldsymbol{D} & -\boldsymbol{D}\boldsymbol{e} \\ \boldsymbol{0} & 0 \end{pmatrix} \in \mathbb{R}^{m \times m}, \tag{3.14}$$

The PDF of the PH distribution $(\boldsymbol{\pi}, \boldsymbol{D})$ is given by:

$$f(x) = \boldsymbol{\pi} e^{\boldsymbol{D}x}(-\boldsymbol{D}\boldsymbol{e}). \tag{3.15}$$

The CDF of the PH is given by:

$$F(x) = 1 - \boldsymbol{\pi} e^{\boldsymbol{D}x}\boldsymbol{e}. \tag{3.16}$$

The moments of the PH distribution is:

$$E[X^n] = n!\boldsymbol{\pi}(-\boldsymbol{D})^{-n}\boldsymbol{e}. \tag{3.17}$$

An m-phase PH distribution has $m(m+1)$ free parameters, which bring higher computation complexity. The number of free parameters can be reduced by introducing some special cases of PH distribution. The exponential distribution is a one-phase PH distribution. We mainly introduce two special cases of two or more phase PH distributions, Erlang distribution and hyper-Erlang distribution.

An Erlang distribution is the distribution of a sum of k i.i.d exponential random variables with rate λ. Erlang distribution can be denoted by $Er(k, \lambda)$.

The CTMC representation of an Erlang distribution is depicted in Figure 3.4.

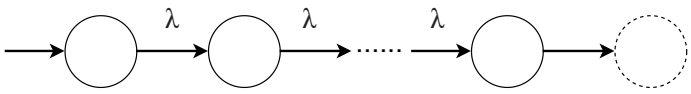

Figure 3.4: The CTMC representation of an Erlang distribution

A hyper-Erlang distribution is a typical representation of branch structure, where each branch is an Erlang distribution. The representation of a PH distribution has a branch structure when:

$$\boldsymbol{D} = \begin{pmatrix} \boldsymbol{D}_1 & & & \\ & \boldsymbol{D}_2 & & \\ & & \cdots & \\ & & & \boldsymbol{D}_n \end{pmatrix}, \tag{3.18}$$

and $\boldsymbol{D}_1, \boldsymbol{D}_2, \cdots, \boldsymbol{D}_n$ are the generator matrixes of the branches.

A representation of the branch structure consists of blocks of states, and the blocks are not connected. Therefore, only one block can be visited in a transition. The CTMC representation of a hyper-Erlang distribution is illustrated in Fig. 3.5

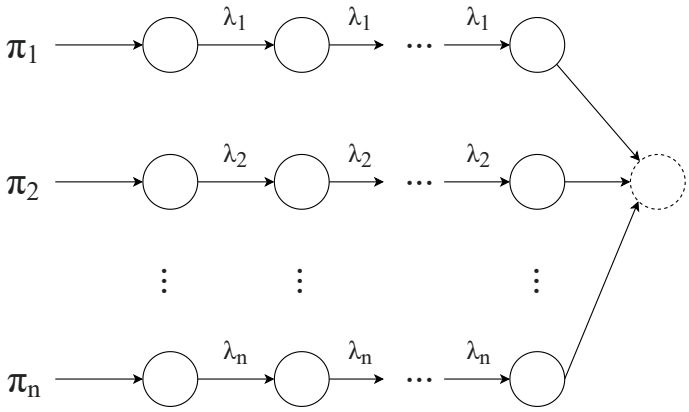

Figure 3.5: The CTMC representation of a hyper-Erlang distribution

A PH distribution $(\boldsymbol{\pi}, \boldsymbol{D})$ is a hyper-Erlang distribution if:

$$\boldsymbol{\pi}_i > 0 \quad if \quad i \in \{1, 1+s_1, \cdots, 1+\sum_{j=0}^{n-1} b_j\}, \tag{3.19}$$

$$\text{else} \quad \boldsymbol{\pi}_i = 0, \tag{3.20}$$

and

$$\boldsymbol{D}_j \in \mathbb{R}^{b_j \times b_j}, \tag{3.21}$$

$$\boldsymbol{D}_j = \begin{pmatrix} -\lambda_j & \lambda_j & & \\ & \cdots & \cdots & \\ & & -\lambda_j & \lambda_j \\ & & & -\lambda_j \end{pmatrix}, \tag{3.22}$$

where $j \in [1, n]$ and $\lambda_j > 0$.

3.1.3 M/PH/1 Queue

In a phase-type distribution with representation $(\boldsymbol{\pi}, \boldsymbol{D})$, $\boldsymbol{\pi}$ is the initial probability vector and $\boldsymbol{D}$ is the generator matrix of an absorbing Markov chain. An $M/PH/1$ queue can be analysed as a Quasi-birth-death (QBD) process with the state space $M = \{0, (i, j), i \leq 1, 1 \leq j \leq v\}$ and state 0 stands for the empty queue, while state (i, j) represents i jobs in the system and the service process is in phase j. We use λ to denote the arrival rate of the Poisson process, and by symbol $\boldsymbol{I}$ we always denote an identity matrix of the dimension appropriate to the formula in which it appears, then we obtain the generator matrix:

$$\boldsymbol{Q} = \begin{bmatrix} -\lambda & \lambda\boldsymbol{\pi} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{\tau} & \boldsymbol{D} - \lambda\boldsymbol{I} & \lambda\boldsymbol{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\tau\pi} & \boldsymbol{D} - \lambda\boldsymbol{I} & \lambda\boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{\tau\pi} & \boldsymbol{D} - \lambda\boldsymbol{I} & \lambda\boldsymbol{I} \\ \mathbf{0} & \mathbf{0} & \ddots & \ddots & \ddots \end{bmatrix} \tag{3.23}$$

and the steady state equations:

$$-\lambda x_0 + x_1\boldsymbol{\tau} = 0 \tag{3.24}$$

$$\lambda x_0\boldsymbol{\pi} + x_1(\boldsymbol{D} - \lambda\boldsymbol{I}) + x_2\boldsymbol{\tau\pi} = 0 \tag{3.25}$$

$$\lambda x_{i-1} + x_i(\boldsymbol{D} - \lambda\boldsymbol{I}) + x_{i+1}\boldsymbol{\tau\pi} = 0 \tag{3.26}$$

where $\boldsymbol{x} = [x_0, x_1, x_2, \cdots]$ is the stationary probability vector and $\sum_{i=0}^{\infty} x_i = 1$.

Multiplying Equation (3.25) and (3.26) by the column vector $\boldsymbol{e}$ on the right, and combining the

results we obtain:

$$x_i = x_0 \boldsymbol{\pi} \boldsymbol{M}^{\boldsymbol{i}}, \quad i \geq 1 \tag{3.27}$$

where $\boldsymbol{M} = \lambda(\lambda \boldsymbol{I} - \lambda \boldsymbol{e}\boldsymbol{\pi} - \boldsymbol{D})^{-1}$, then the average number of jobs in the queue can be denoted as:

$$E[N] = \sum_{i=1}^{\infty} i x_i \boldsymbol{e} = x_1 \sum_{i=1}^{\infty} \frac{d}{d\boldsymbol{M}} \boldsymbol{M}^i \boldsymbol{e} = x_1 (\boldsymbol{I} - \boldsymbol{M})^{-2} \boldsymbol{e} \tag{3.28}$$

and according to Little's law, the mean time a job spends in the system is:

$$E[R] = \frac{E[N]}{\lambda} = \frac{x_1 (\boldsymbol{I} - \boldsymbol{M})^{-2} \boldsymbol{e}}{\lambda} \tag{3.29}$$

3.2 Machine Learning

Machine learning is a type of artificial intelligence (AI) method that uses statistics to find patterns in large amounts of data. A machine learning model is built on the sample data, also called training data. The model is used to make predictions or perform other tasks instead of the explicitly designed programs [65]. Generally, machine learning approaches are divided into three types: supervised learning, unsupervised learning and reinforcement learning. Supervised learning is where both inputs and outputs are provided in the training data, thus the model maps new input data to the expected output. Supervised learning is often applied to solve classification or regression problems, and it is also the type of machine learning we applied in this thesis. Conversely, unsupervised learning means there are only inputs in the training data and the outputs are not known. Unsupervised learning is often applied to study the underlying structure or distribution of the data, e.g. the clustering and association problems. Reinforcement learning does not require correct inputs or outputs, instead it performs actions by a trial-and-error method. The reinforcement learning model uses an agent to take actions and interact with the environment to achieve the maximum reward. It is usually applied in robotics, chemistry or games.

3.2.1 Artificial Neural Network

Artificial Neural Networks (ANN) are computational models inspired by the biological neuron networks of the human brains [17, 18]. In the last decade, ANNs have been applied in all kinds of disciplines to model complex real-world problems including financial decision-making [70], energy consumption forecasting [3], natural resource utilization [78], pattern recognition [19], and game

playing [97]. ANNs are very adaptive in modeling nonlinear processes and become the most popular tool for regression and classification.

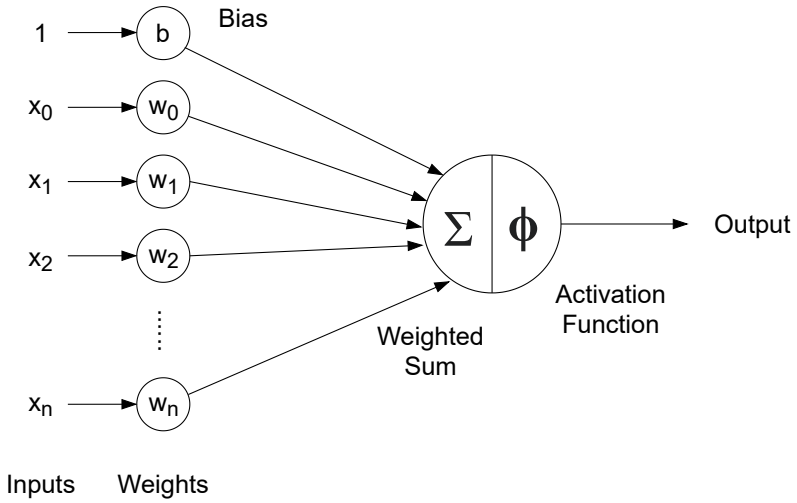

Figure 3.6: A basic artificial neuron

The components of an ANN include a group of artificial neurons and their connections with assigned weights, which makes the network a directed, weighted graph. A basic artificial neuron may have multiple inputs but produces a single output, as depicted in Fig. 3.6. The inputs, denoted by $x_0, x_1, x_2, ..., x_n$, can be the feature values from a piece of external data, or outputs from other neurons. Each of the inputs is multiplied by a corresponding weight, which shows the strength of a particular feature. The weights are represented by $w_0, w_1, w_2, ..., w_n$. To obtain the output, the first step is to produce the weighted sum of the inputs, denoted by Σ in the figure. Then the value is passed through an activation function Φ to generate the result as output, as represented mathematically as follows:

$$Output = \Phi(b + \sum w_i x_i) = \Phi(b + w_0 x_0 + w_1 x_1 + ... + w_n x_n) \tag{3.30}$$

where b is a bias value, a constant used to adjust the output along with the weighted sum. The activation function Φ is used to introduce nonlinearity into the output, and there are numerous kinds such as the Sigmoid function, the Hyperbolic Tangent function and the ReLU (Rectified Linear Unit) function. The Equations and the curves of the three functions which we applied in this thesis are introduced as follows:

- **Sigmoid function:** It is a function with "S" shaped curve and its return value (y axis) ranges

between (0,1), therefore it is commonly used for probability prediction. The equation of the Sigmoid function is listed below and we can observe from Fig. 3.7 that its curve has smooth gradient. This prevents jumps in the output values. However, for x values that are too high or low, there is almost no change to the return value, causing a vanishing gradient problem. In this case it takes the ANN more time to reach an accurate prediction, or it may even fail to learn further.

$$\Phi(x) = \frac{1}{1 + e^{-x}} \tag{3.31}$$

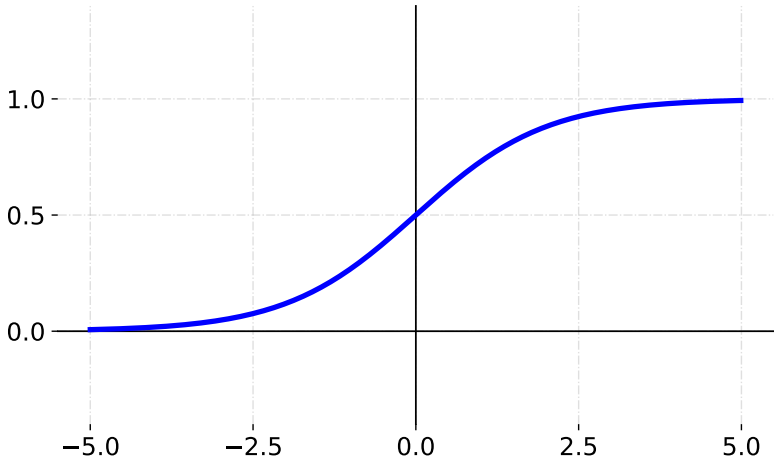

Figure 3.7: The curve of Sigmoid function

- **Hyperbolic Tangent function:** This is very similar to the Sigmoid function, but the return value ranges between -1 and 1, as depicted in Fig. 3.8. Thus it is easier to model inputs that are strongly negative, and only the zero-valued inputs are mapped to the near-zero outputs. The equation of this function is:

$$\Phi(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.32}$$

- **ReLU function:** From Equation (3.33) and Fig. 3.9 we can observe that the ReLU function generates x as the output if x is positive, otherwise the output value is 0. It allows the ANN to converge quickly, but when inputs approach zero, or are negative, the gradient of the function becomes zero, the ANN cannot perform backpropagation and fails to learn.

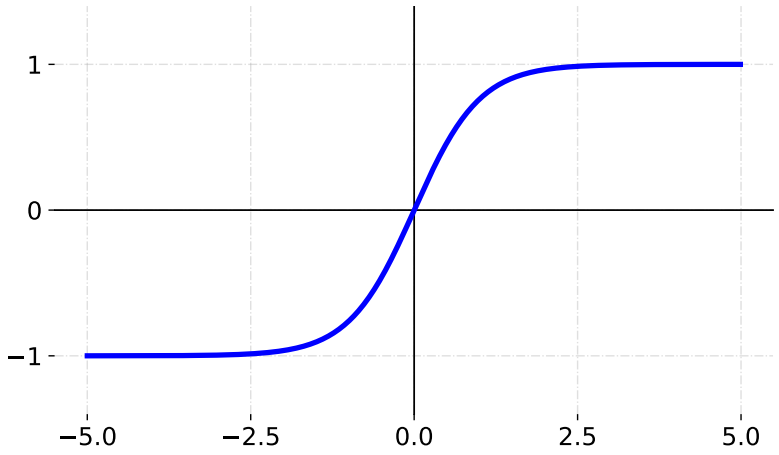

Figure 3.8: The curve of Hyperbolic Tangent function

$$\Phi(x) = \begin{cases} x & for \quad x > 0 \\ 0 & for \quad x \leqslant 0 \end{cases} \tag{3.33}$$

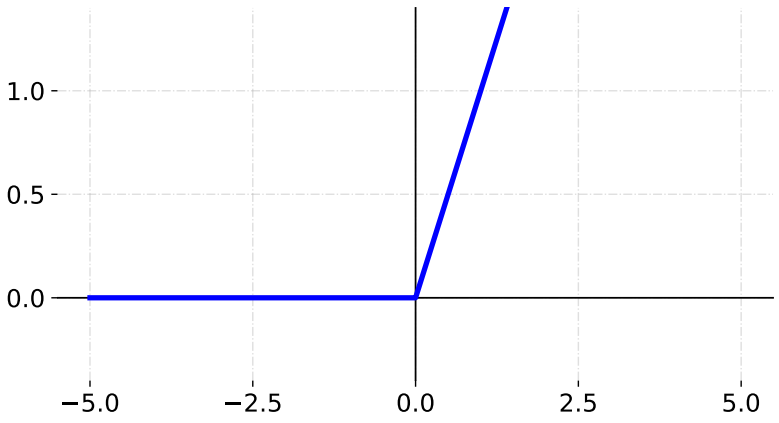

Figure 3.9: The curve of ReLU function

3.2.2 The Learning Process of ANN

Based on the introduction of the basic artificial neuron above, we describe how a typical ANN works. Generally, most applications organize multiple neurons into several layers to amplify the

power of the ANN and increase its accuracy. A typical ANN normally consist of three layers: (i) *input layer*, it receives data from external applications, such as files or sensors; (ii) *output layer*, it produces the final results to other processes such as a mechanical control system; (iii) *hidden layer*, it is the layer between the input layer and output layer, and there can be multiple hidden layers. Fig. 3.10 illustrates a simple ANN model with three hidden layers, which contain four, six and five neurons respectively. Each neuron in a hidden layer receives the results from the previous layer, performs its function and produce its result to neurons in the next layer. In this example the connection between two adjacent layers is fully-connected, while it can be not fully-connected depending on the specific case.

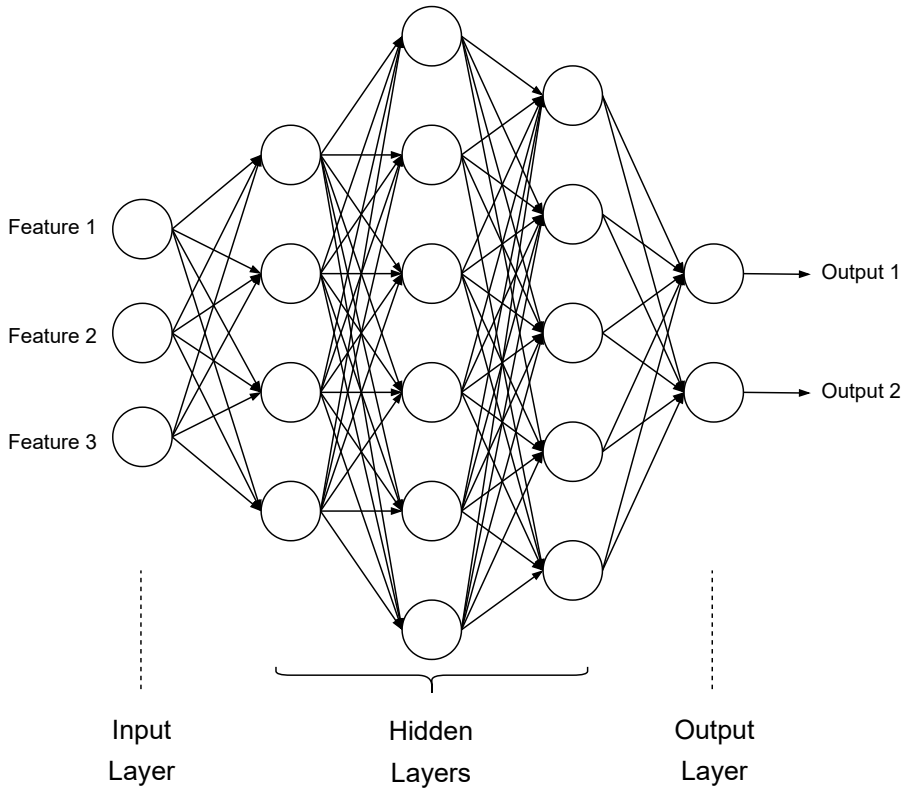

Figure 3.10: A simple Artificial Neural Network

Once an ANN is created and able to generate outputs as the workflow described above, a learning process is required to improve this ANN. The initial weights in the ANN are chosen randomly, and the purpose is adjusting these weights and threshold to obtain accurate results. Mathematically, the objective of learning is to minimize the error between the actual value and the predicted value. The

error is measured by a cost function, which is evaluated periodically during the learning process. The major learning approaches include supervised learning, unsupervised learning and reinforcement learning. In our work we applied supervised learning, as both the input and output data are provided. The set of this data which enables the learning process is called the *training set*. The cost function can be the mean absolute error (MAE), which is obtained according to the equation below:

$$MAE = \frac{\sum_{i=1}^{n} |\hat{y} - y|}{n} \tag{3.34}$$

where $\hat{y}$ is the predicted output and y is the actual value. The error is then propagated to the system and the weights are adjusted for the next run. This workflow repeats over and over as the weights are continually tweaked. Each error is effectively divided among the connections for compensation. In this way, the error becomes lower and finally reaches the desired level.

The procedure that tweaks the weights according to each error is called *Backpropagation*, and there are serveral methods for it, such as Extreme Learning Machines (ELM) and the No-Prop algorithm [52, 107]. We mainly introduce the Stochastic Gradient Descent (SGD) method, which is also applied in our work.

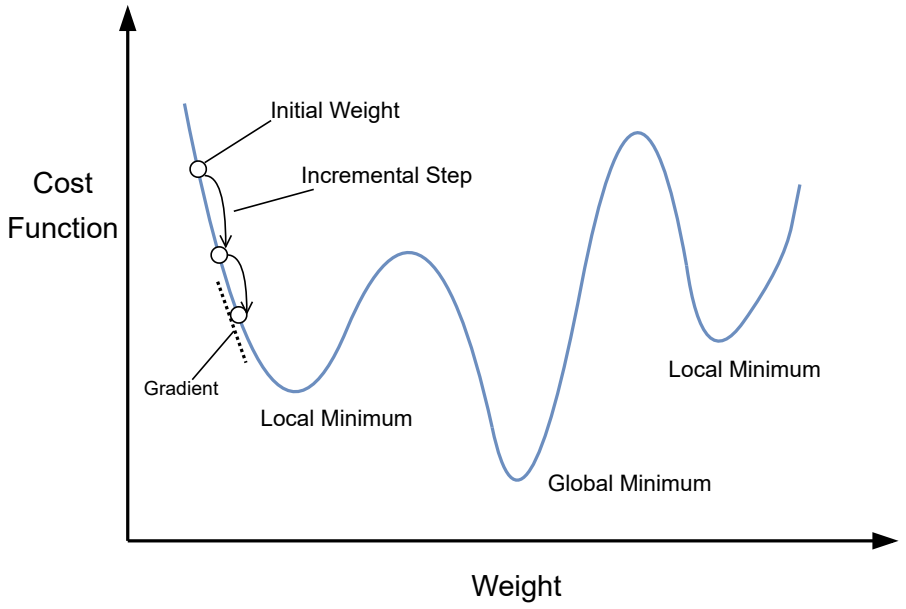

Figure 3.11: Stochastic Gradient Descent

The gradient is the derivate of the cost function in a specific state with respect to the weights, as depicted in Fig. 3.11. Gradient Descent means that the weight should be adjusted by taking a step

in the opposite direction of the cost gradient:

$$\Delta w_i = -\eta \frac{\partial \Phi}{\partial w_i} \tag{3.35}$$

where η is the learning rate which decides the magnitude of the adjustment. The cost reaches a minimum once the gradient equals zero, and this is the desired weight. However, the computing cost based on gradient using the entire training set is very expensive because it uses the whole data set in each iteration. In SGD, only a few samples are picked randomly in each iteration instead of the entire training data. Therefore it computes much faster than the regular Gradient Descent. It has been shown that SGD almost surely converges to the global cost minimum if the cost function is convex (or pseudo-convex) [89].

3.2.3 Different types of ANNs

There are many types of ANNs, which have different mathematical operations and parameters to determine the output. In order to obtain a general picture of these types, we select some commonly seen ones to be introduced, as listed in Fig. 3.12.

- **Feedforward Neural Network:** This is one of the simplest forms of ANN, and also the type of ANN that we apply in our work. Feedforward means the data travels from the input layer to the output layer in one direction without any loop. It can be further subdivided into time delay neural network (TDNN), probabilistic neural network (PNN), convolutional neural network (CNN) and some other types.

  A TDNN adds delays to the input to achieve time-shift invariance, thus the classifier does not require explicit segmentation prior to classification. The TDNNs are usually applied in speech recognition, handwriting recognition and video analysis [56, 104, 108].

  A PNN uses probability distribution function (PDF) for classification. The PDF of each class is approximated by a function, thus new input data is allocated to the class with the highest probability. The applications based on PNN include intrusion detection and remote-sensing image classification [117, 118].

  A CNN operates convolution on the inputs which are usually images. Convolution is a mathematical operation that produces results to show how the shape of a function is modified by the other function. Generally images are 2D structure data, and the convolution operation helps to process the data in batches. This allows the neural network to be deeper with fewer parameters [2]. In addition to image and video recognition, the CNNs are also applied in natural language processing (NLP) and games [31, 97].

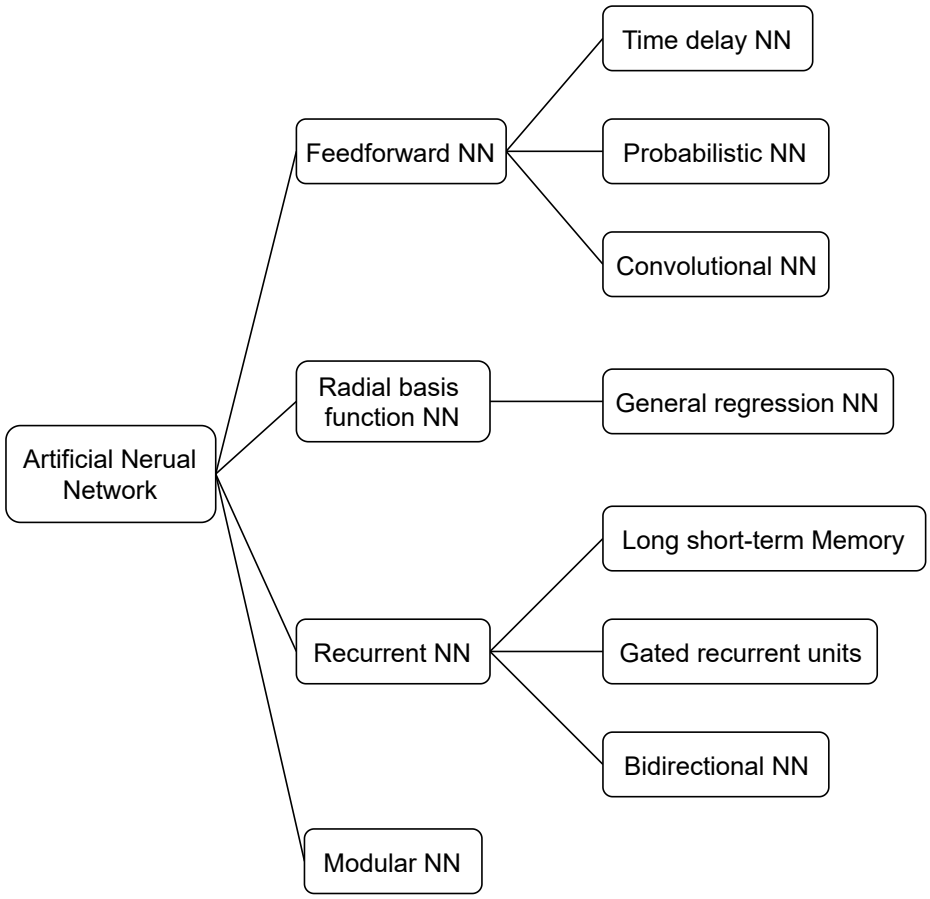

Figure 3.12: Some common types of Artificial Neural Networks (ANN)

- **Radial basis function Neural Network:** This type of neural network uses Radial basis function (RBF) as the activation function. RBF produces results from the distance between the input point and a fixed point. The input points are classified depending on their RBF results. If the input point is closer to a fixed point, the likelihood of this input point being classified to this class is higher. The applications based on RBF neural network include power system restoration and traffic flow prediction [28, 90]. A variation to RBF neural networks is the general regression neural network (GRNN), which is used for regression rather than classification. The GRNNs are used for hand gesture recognition and wind speed forecasting [83, 86].
- **Recurrent Neural Network:** Unlike the feedforward neural network, a recurrent neural network (RNN) also propagates data backwards. Apart from the input and output connection, each recurrent neuron has a feedback connection from its output back to its input. The feed-

back connections make RNN able to store information from the previous time step. The advantage is that RNN can handle sequential data as described in the previous paragraph. Therefore it is significant for applications like speech recognition, music composition and sentiment analysis [16, 51, 77].

The drawback of RNNs is the limitation on the range of contextual information and that the gradient will vanish or explode in some cases. Then during the learning process, the neural network may not work properly. Long short-term memory (LSTM) is an architecture widely used in RNN to solve these problems. By introducing the concept of three gates, the input gate, the output gate and the forget gate, the LSTM units track the dependencies between the input elements. The LSTM improves the RNN with the ability to bridge time intervals and works better on a large variety of problems like robot control and clinical events prediction [30, 79].

The gated recurrent unit (GRU) is similar to the LSTM, but it uses a reset and an update gate to adaptively reset or update its memory content. It has been studied that GRU outperforms LSTM in some certain applications such as motive classification of text [45].

The bidirectional recurrent neural network (BRNN) is a combination of two RNNs which move data in the opposite directions. Thus the BRNN has both backward and forward information about the sequence at every time step. BRNNs are often combined with LSTM in the applications including translation and traffic flow prediction [98].

- **Modular Neural Network:** It breaks down a large computational process into smaller components constructed by a series of independent neural networks. Each neural network has a set of separate inputs to perform some subtasks and these neural networks do not interact with each other. The results of each neural network are processed by a intermediary to produce the output as a whole. The advantages include higher computation speed, less training time and the network becomes more tolerable to the failure in nodes. Modular neural network (MNN) grows rapidly in recent years and it is widely applied in areas like fault diagnosis, image classification and quality of transmission prediction [42, 85, 102].

3.3 Summary

In this chapter we introduce all analysis elements that have been applied in this thesis. The theoretical background includes two parts, queueing theory and machine learning method. We introduced the phase-type distribution, which is widely applied for measuring the performance metrics in computer systems. We present the derivation of a $M/PH/1$ queue, which is applied in Chapter 4. Then we elaborate on the classifications of different machine learning approaches, as well as the various

artificial neural networks. We introduce the architecture of typical ANN model and the learning process of ANN. The application of machine learning technology in this thesis is in Chapter 5.

Chapter 4

Performance Evaluation of Apache Kafka

Apache Kafka has been developed originally at LinkedIn to process real-time log data with delays of no more than a few seconds, and it is designed to be a distributed, scalable, durable and fault-tolerant messaging system with high throughput [68, 105]. LinkedIn relies heavily on the scalability and reliability of Kafka for use cases like monitoring system metrics, traditional messaging or website activity tracking [43]. Kafka can handle more than 1.4 trillion messages per day across over 1400 brokers in LinkedIn [67]. Currently, due to its strong durability and high throughput with low latency, Apache Kafka has been widely used in today's Internet companies [11]. Twitter uses Kafka as a part of their Storm stream processing infrastructure, and Netflix applies Kafka in the Keystone pipeline for real-time monitoring and event processing.

Furthermore, many cloud vendors provide Kafka as a service for users who want to build and run applications that use Apache Kafka to process data. This makes it expensive to perform benchmark tests on Kafka to obtain the proper configuration. In this chapter, we propose a queueing based model to predict the performance of Kafka. Users can tune the configuration parameters in the model to observe the parameters' impact on Kafka's performance. This can help users to utilize the limited resources provided by cloud vendors more economically and give explicit advice on the configuration settings of Kafka.

4.1 Kafka as a Service

Kafka as a service is a cloud computing service model similar to Infrastructures as a Service (IaaS), where basic computing resources like CPUs and storage are provisioned over the Internet. Users generally pay according to the length of use, while the cloud vendor manages and maintains the Kafka cluster. The advantage is that users can easily scale up or down resources to meet their requirements, without purchasing, installing and networking Kafka cluster servers by themselves. Here we list several typical products. Currently there are services like Amazon Managed Streaming for Kafka (MSK) provided by Amazon Web Services (AWS) [7], Event Streams provided by IBM [53], Apache Kafka for HDInsight provided by Microsoft Azure [80], and Confluent Cloud provided by Confluent Cloud Enterprise, which is also built by the creators of Kafka [32]. Those vendors provide limited resources, such as CPUs, memory and disks, and in the given hardware environment, users can set multiple Kafka configuration parameters on their own, such as batch size, partition number, and log retention time. How to configure those parameters to gain the best performance for a specific application is unclear without numerous experiments, which are time-consuming and costly. Besides, it is very inefficient to tune those parameters after a Kafka cluster is running.

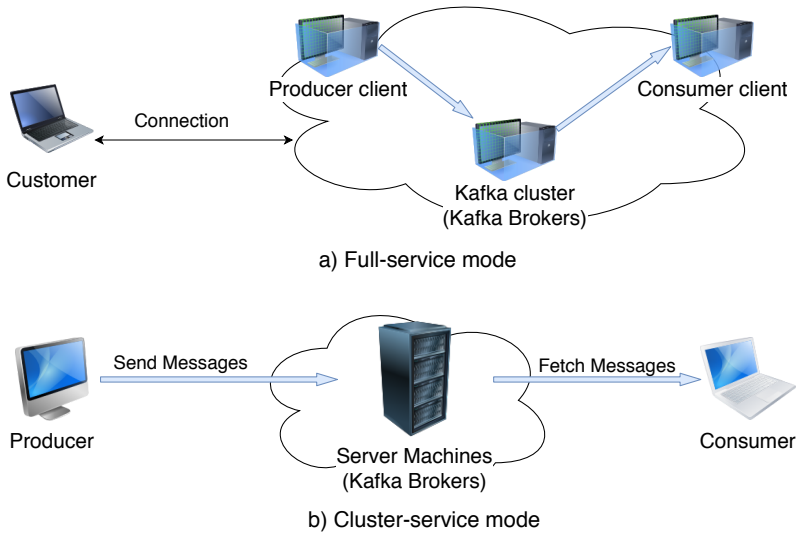

Figure 4.1: Kafka as a Service

Cloud service vendors provide Apache Kafka as a service mainly in two different modes, the full-service mode and cluster-service mode. The VPC provided by AWS offers full-service mode where a user can create producer clients, consumer clients and a Kafka cluster on the cloud servers, and the

user only needs a laptop connecting to the console for sending commands, as depicted in Fig. 4.1a. This mode is applicable when the upstream and downstream applications that cooperate with Kafka are also deployed on cloud services.

Under many circumstances data is generated or consumed locally, and users will run Kafka producer or consumer clients on their own machines. Then the choice should be cluster-service mode where cloud vendors like Confluent and Aiven provide a Kafka cluster as a service, as shown in Fig. 4.1b. In this mode producer clients send messages with local machines, then cloud servers distribute and store those messages for consumer clients to fetch. Normally users have to run test cases on a Kafka cluster and compare the performance results under different configuration parameters. However, the Kafka cluster needs a restart every time the configuration parameter is changed, and running test cases on cloud servers is time-consuming and expensive.

4.2 Performance Model

Predicting the performance of a Kafka messaging system accurately plays a critical role in improving the user's efficiency of deploying Kafka cloud services. This section presents the performance model of Kafka to estimate the performance metrics under given configuration parameters. The key configuration parameters include batch size and number of partitions, which are normally difficult to choose.

4.2.1 Producer Throughput

The producer performance metric for which the user cares most is usually the throughput of messages, the number of messages sent from the producer to the cluster in a certain time interval.

Since Kafka is a messaging system, the minimum unit of tasks is the message. Kafka will do some pre-processing for raw messages to get messages that fit its own format. There are several differences between Kafka's message formats for different versions, but they generally follow the structure in Fig. 4.2. Messages in Kafka consist of a variable-length header, a variable length opaque key byte array and a variable length opaque value byte array, as depicted in Fig. 4.2a. The key is used for partition assignment, and the value is the actual message content. In the header of a Kafka message, parameters are used to check the integrity of the message on the broker and consumer, and to allow backward compatible evolution of the binary message format, attributes holds metadata attributes including the compression codec. Kafka batches small messages together as a messageSet for higher efficiency: Producers send messageSets to brokers and consumers fetch messageSets instead of working with a single message. A messageSet is also the unit of compression in Kafka, which could

be compressed and stored in the value field of a single message with appropriate compression codec set, and then the receiving system parses the actual messageSet from the decompressed value. As shown in Fig. 4.2, Kafka allows messages to recursively contain compressed messageSets to allow batch compression in order to achieve additional efficiency. Here the example in Fig. 4.2c consists of three messageSets, which are unfolded in Fig. 4.2b.

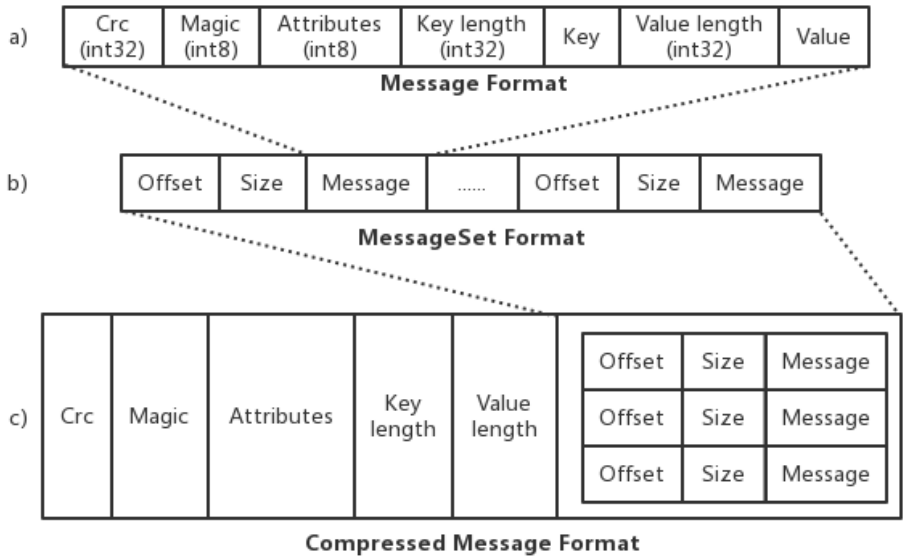

Figure 4.2: a)Message Format b)MessageSet Format, c)Compressed Message Format of Kafka

Since a Kafka producer generally performs a batch-based transmission scheme, where messages are accumulated in memory to larger batches before they are sent over the network. Through the Kafka producer API, the user of the Kafka cloud service can configure the batching to accumulate no more than a fixed number of messages, and the topic those messages go to must be specified. Therefore the user should create a topic before sending messages from producer to the Kafka cluster, and the number of partitions under this topic is another indispensable configuration parameter. We assume that the upstream applications always generate sufficient messages to deliver to the Kafka messaging system. Thus the batches are always filled with maximum number of messages and are sent before the timeout (details are introduced in Chapter 6) is reached. The producer client will group batches into a single packet based on the number of leader partitions on a broker, as depicted in Fig. 4.3.

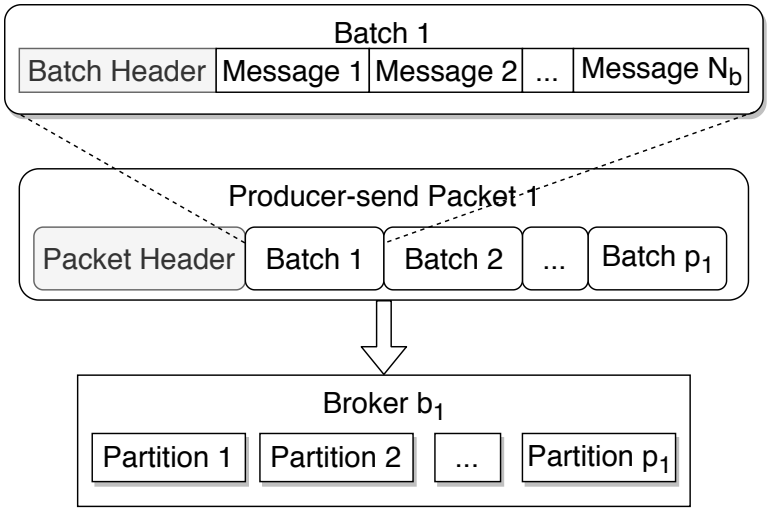

Figure 4.3: Packet sent from producer

We use N_b to denote the number of messages per batch, and B to denote the number of brokers in the Kafka cluster. Then for any topic, the number of its leader partitions on broker $b_i (1 \leq i \leq B)$ can be denoted as p_i, and we define the assignment matrix of partitions as $\boldsymbol{P} = \begin{bmatrix} p_1 & p_2 & ... & p_B \end{bmatrix}$. As illustrated in the example of Fig. 4.3, every packet sent by a producer to broker b_1 contains p_1 batches, because Kafka provides each leader partition with one batch in a single packet to achieve load balancing among partitions. Therefore, each element in the matrix $N_b\boldsymbol{P}$ denotes the number of messages per packet to its corresponding broker. Both batch and packet contain headers, as shown in Fig. 4.3. Those headers consist of many metadata attributes that are required to enable successful delivery. In this chapter we define the intended messages in a packet as the payload, and the left headers are the overhead of the packet. We use S_m to denote the message size in bytes, then the batch size can be obtained as $S_b = (S_m \cdot N_b + batchHeaderSize)/compressionRatio$, where the compression ratio is depending on the type of compression chosen by users. The size of the packet received by each broker can be expressed in the form of a matrix:

$$\boldsymbol{S}_r = S_b \cdot \boldsymbol{P} + packetHeaderSize \cdot \boldsymbol{e}_B^T \tag{4.1}$$

where $\boldsymbol{e}_B$ is a $B \times 1$ column vector with all B elements equal to one.

Considering all the packets sent from a producer to any individual broker b_i, we define the period of time between two subsequent packets as the packet send interval of b_i. In our experiments we observe that a Kafka producer sends packets to different brokers in random order, and the mean

packet send interval largely depends on the size of a packet. Therefore we assume the packet send interval is a function of the packet size $\boldsymbol{G}_s = f_p(\boldsymbol{S}_r)$, where each element in the $1 \times B$ matrix $\boldsymbol{G}_s$ represents the mean send interval of packets to the corresponding broker. In a correlation analysis of packet size and packet send interval we have obtained the parameters of the function $f_p(x)$. The experimental results are explained in Section 4.3. Then we define the number of packets sent from a producer to a broker in an unit of time as the packet rate of this broker, and use $\boldsymbol{R}_s$ to denote the matrix of packet rates to each broker which satisfies the equation $\boldsymbol{R}_s \circ \boldsymbol{G}_s = \boldsymbol{e}_B^T$, where $\circ$ stands for the Hadamard product. Given two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ of the same dimension, the Hadamard product $\boldsymbol{A} \circ \boldsymbol{B}$ is a matrix of the same dimension with elements $(A \circ B)_{ij} = (A)_{ij}(B)_{ij}$. Thus we can obtain the mean number of packets sent from producer per unit of time, namely the throughput of producer-send packets in quantity $X_{sp} = \boldsymbol{R}_s \cdot \boldsymbol{e}_B$ and the throughput of producer-send packets in bytes per second $X'_{sp} = (\boldsymbol{R}_s \circ \boldsymbol{S}_r) \cdot \boldsymbol{e}_B$, which is also the network bandwidth required to send packets.

Table 4.1: Notations

Symbol	Definition
N_b	Number of messages per batch
S_m	The message size in bytes
S_b	The batch size in bytes
$\boldsymbol{P}$	The matrix of partition assignment
$\boldsymbol{S}_r$	The matrix of packet size received by each broker
$\boldsymbol{R}_s$	The matrix of packet rates

The throughput in number of messages can be denoted by $X_{sm} = (\boldsymbol{R}_s \circ N_b\boldsymbol{P}) \cdot \boldsymbol{e}_B$ as well as the message throughput in bytes per second $X'_{sm} = (S_m(\boldsymbol{R}_s \circ N_b\boldsymbol{P}))\cdot$.

The required network bandwidth is determined by X'_{sp}, and we define the relative message throughput in the occupied bandwidth as the relative payload of producer-send packets $\varphi_{ps} = X'_{sm}/X'_{sp}$, therefore the relative overhead $\varphi_{os} = 1 - \varphi_{ps}$.

4.2.2 Broker Storage

As mentioned in Section 2.2, a partition is an ordered, immutable sequence of messages which can not be split across brokers. This means whenever a new message is published to a topic, it is appended to the end of a partition and assigned with an offset, which is a per-partition monotonically increasing sequence number. The file structure of a partition is illustrated in Fig. 4.4. There are multiple segment files with the same size (by default it is 500MB) in a partition. At first segment

file 1 is generated, including one log file, which is the actual file where messages are stored, and one index file, which stores the metadata information. When the size of the log file reaches 500MB, the next segment file (file 2) will be generated and its log file and index file will be named based on the offset value of the last message from the segment file 1. The metadata in the index file points to the physical offset addresses of the messages in the corresponding log file. The index file is made up of 8 byte entries with 4 bytes to store the offset relative to the base offset and 4 bytes to store the position. In Kafka the configuration parameter $index.interval.bytes$ sets an index interval which describes how frequently (after how many bytes) an index entry will be added. In cloud services the disk space is quite limited, so in order to save disk space, the user of Kafka as a service can specify how long data should be retained, after which time the data will be deleted. For instance, in an application which monitors the weather information (e.g. temperature and humidity), the Kafka cluster receives data continuously from the sensors and persist the data for weather prediction. The volume of occupied storage grows rapidly as the sensor data is generated in a never-ending way. Thus it will free more space without harming the prediction accuracy by deleting the data which contains the weather information from one week ago. The configuration parameter that controls the log retention time is $log.retention.hours$ and the default value is 168 hours.

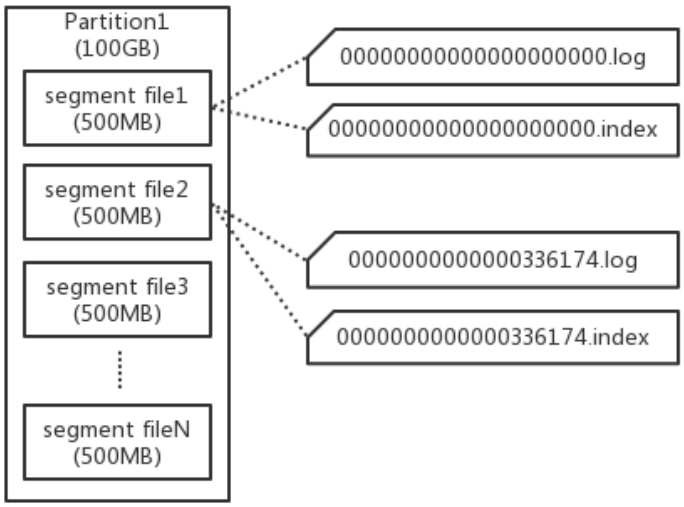

Figure 4.4: The file structure within a partition

If a producer has been producing messages for a period of time T_p, we can denote the used storage

of the Kafka cluster by matrix:

$$\boldsymbol{H} = (1 + \frac{8}{index.interval.bytes})T_p(\boldsymbol{R}_s \circ \boldsymbol{S}_r) \tag{4.2}$$

where each element in $\boldsymbol{H}$ denotes the size of stored segment files on the corresponding broker.

4.2.3 Consumer Throughput

Similar to the process in the producer, the messages fetched by a consumer are also effectively batched. More specifically, the message format on brokers is exactly the same as what the producer sends through the network, which means messages are still batched on brokers. The consumer sends fetch requests to the brokers in the Kafka cluster, and each broker responds with a packet that contains several batches of messages. The number of batches in a consumer-fetch packet is determined by the number of partitions on the broker, because each partition will provide at least k batches in the response packet, where k is a non-zero positive integer. Therefore we can denote the number of messages fetched per request as $kN_b\boldsymbol{P}$, and define the size of a packet that a consumer fetches from each broker through a single fetch request as the fetch size of the consumer $\boldsymbol{S}_f = S_b \cdot \boldsymbol{P} + packetHeaderSize \cdot \boldsymbol{I}_B^T$.

In the situation that there are multiple consumer instances in a consumer group, the elements in matrix $\boldsymbol{P}$ should be adjusted to the number of partitions that are assigned to the specific consumer instance. By default the consumer uses the $poll$ method to fetch messages, thus when there are plenty of messages to be consumed, the rate of the consumer-fetch packets depends on how fast the broker accumulates enough batches in a packet. We study the correlation between the size of the consumer-fetch packet and the time interval of sending these packets from a broker to the consumer. We use the matrix $\boldsymbol{G}_f = f_c(\boldsymbol{S}_f)$ to denote the interval on each broker, and $\boldsymbol{R}_f$ to denote the matrix of consumer-fetch packet rates which fits $\boldsymbol{R}_f \circ \boldsymbol{G}_f = \boldsymbol{e}_B^T$. Then the throughput of consumer-fetch packets is $X_{fp} = \boldsymbol{R}_f \cdot \boldsymbol{e}_B$ in number of messages and $X'_{fp} = (\boldsymbol{R}_f \circ \boldsymbol{S}_f) \cdot \boldsymbol{e}_B$ in bytes.

4.2.4 Packet Latency

Sometimes the users of Kafka as a service want to make a real-time analysis, and their metric of concern is the end-to-end latency of a packet, which is defined as the time between the sending of a packet from the producer to the moment the packet is received by the consumer.

In queueing theory the end-to-end latency is the response time of packets, and we denote the mean response time of packets through each broker by a $1 \times B$ matrix $\boldsymbol{R}$. According to the description in Section 4.2.1, over a certain period of time, the number of packets that are sent to each partition

is almost equal. Thus we use the number of partitions on a broker b_i as the weight of the mean response time on this broker, then we can take the weighted mean response time of all brokers as the mean response time of packets through the Kafka cluster, given by $(\boldsymbol{R} \circ \boldsymbol{P}) \cdot \boldsymbol{e}_B / \boldsymbol{P} \cdot \boldsymbol{e}_B$. We model one of the brokers as server in a queueing model, which deals with 3 types of request streams. The first request stream is exactly the packet sent from producer with arrival rate λ_s, and the service time is from when the broker receives a packet to when it is stored on disk, as illustrated in Fig. 4.5. The second request stream is the fetch request with arrival rate λ_r from the other brokers to replicate the messages on leader partitions, and the service time is the time spent to send replicated messages. The last request stream received by a broker is the fetch request with arrival rate λ_f from a consumer, and the service time is the time to send packets to the consumer.

According to the research in [46], we can use a Poisson process to approximate the arrival process of requests. Since any distribution with a strictly positive support in $(0, \infty)$ can be approximated arbitrarily close by a phase-type distribution [24], we assume the service time is phase-type (PH) distributed, represented by the three distributions $PH_s(\boldsymbol{\pi_s}, \boldsymbol{T_s})$, $PH_r(\boldsymbol{\pi_r}, \boldsymbol{T_r})$ and $PH_f(\boldsymbol{\pi_f}, \boldsymbol{T_f})$ respectively. Therefore, the process of a broker dealing with each job is an $M/PH/1$ queue. The derivation of the $M/PH/1$ queue is introduced in Section 3.1.3.

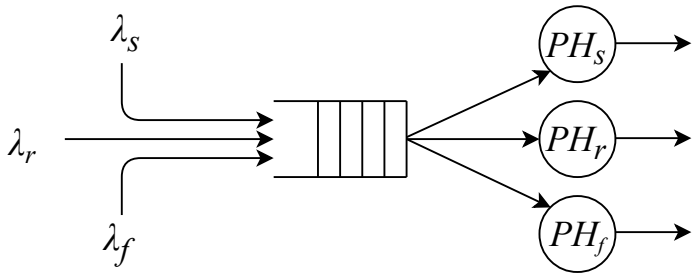

Figure 4.5: The queueing model of a broker

We use $E[R_s]$, $E[R_r]$ and $E[R_f]$, respectively, to denote the mean response time of the three jobs mentioned above. When a user chooses the synchronous replication mode, a message will be available to consumers only after it is replicated to all follower partitions, thus the average end-to-end latency of a packet equals $E[R_s] + E[R_r] + E[R_f]$. However, in the asynchronous replication mode the consumer can fetch a message as long as a it is stored in the leader partition, without waiting for the replication process. In this situation the average end-to-end latency is $E[R_s] + E[R_f]$.

4.3 Experimental evaluation

In this section we evaluate the proposed model from different perspectives for multiple scenarios.

Our Kafka system is built from the latest version 2.1.1, orchestrated by Zookeeper 3.4.12 and each broker is started on a single PC. We built the Kafka cluster with 3 brokers to simulate the environment of using Kafka as a service, and use two other PCs as the producer client and consumer client, respectively. Each PC is equipped with 8G memory and all have the same type of CPU: Intel(R) core(TM)2 Quad CPU Q9550@2.83GHz, running Debian 4.9.144-3.1, so our servers are homogeneous, which is common for many cloud platforms.

We study the use case where a user buys Kafka cloud service to use the message broker for a server log analysis application. The data source is web server access logs from NASA Kennedy Space Center web servers [46], and each record in these logs is a message in Kafka. Our experiments revolve around the most relevant metrics, such as the throughput of producer and the relative payload under heavy load, the utilization of network bandwidth, the disk space limits and Kafka data retention time.

4.3.1 Correlation Analysis

The performance of the Kafka system varies significantly depending on the message type and the compute power of machines (e.g., CPU speed, memory size), making it extremely difficult to build a general model for all cases. Referring to the method in [106], we collected execution traces using a small fraction of the source data, and obtained the essential parameters for our performance model. We analysed the correlation between producer-send packet size $\boldsymbol{S}_r$ and the send interval $\boldsymbol{G}_s$ to find function $f_p(x)$ through the metrics recorded in the producer throughput tests. All tests run for at least 60 seconds after setup and messages are sent with different batch sizes to disjoint topics. The number of messages in one batch N_b ranges from 1 to 50, and we create 10 topics with 3 to 12 partitions, therefore the packet size in our tests vary according to equation (4.1). Other configurations are all set to the default values. We illustrate part of our correlation analysis results in Fig. 4.6, including the scatter points for the mean packet size and send interval for one broker, as well as the lines of best fit.

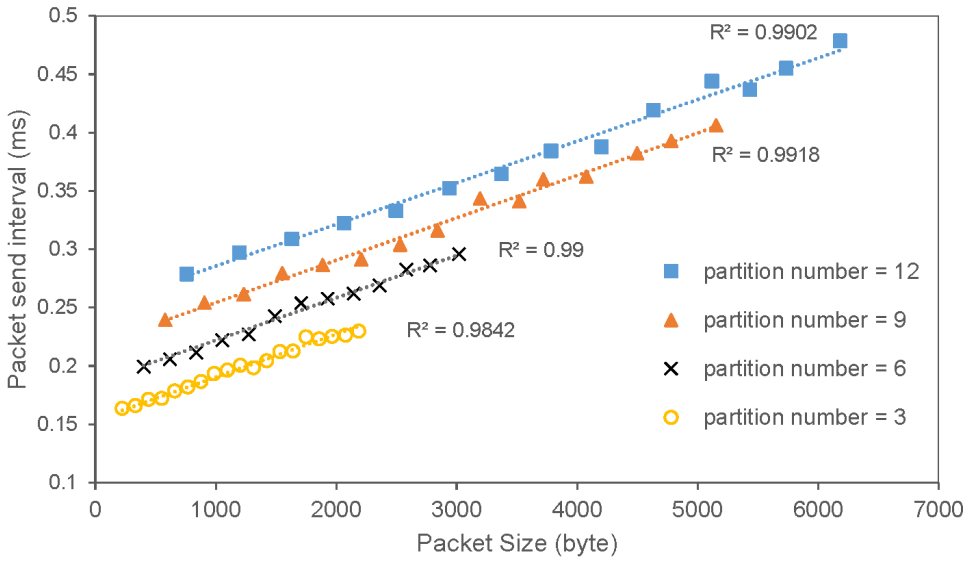

Figure 4.6: Correlation analysis of producer-send packet size and send interval

In Fig. 4.6 we can observe a strong, positive and linear correlation between packet size and send interval, grouped by different number of partitions. For the packets of almost the same size, a larger number of partitions causes certain increments on send interval, because the Kafka producer needs longer to arrange metadata information of each partition. The packet send interval in a certain topic increases linearly as the size of the packets grows, and the slopes of different topics are approximately equal. Every coefficient of determination in our correlation analysis results, denoted by R^2, is close to 0.99, as depicted in Fig. 4.6. Likewise, we study the correlation between the consumer-fetch packet size $\boldsymbol{S}_f$ and the interval $\boldsymbol{G}_f$ and the R^2 is closed to 0.95. This indicates that the correlation between packet size and send interval is positive and we can use the results to complete the function $f_p(x)$ and $f_c(x)$ in the performance model.

4.3.2 Network Bandwidth Evaluation

From [73] it is known that the network bandwidth provided by cloud vendors varies significantly, and the available bandwidth for a user is always limited. Moreover, a Kafka cluster has the ability to enforce network bandwidth quotas on packets to control the resources used by the clients. By default, each unique client group receives a fixed quota in bytes/sec as configured by the cluster, which is defined on a per-broker basis. When the upstream application generates many messages to the Kafka system, the user expects to better utilize the available network bandwidth aiming at larger throughput. Since the results of producer-send packets and consumer-fetch packets are similar, we

take the example of the former one to analyze and compare the numerical results and experimental results.

In Section 4.2.1 we stated the closed-form expression of packet throughput under heavy load, and the required network bandwidth for each broker can be obtained from the equations in the section. In Fig. 4.7 and Fig. 4.8 we show the experimental results under heavy load in comparison with the model prediction results, separated by the number of partitions in the topic. Through the experiments we intend to explore the impact of batch size and number of partitions on the throughput of packets from producer client in Megabytes per second, and the relative overhead φ_{os}.

The scatter points in Fig. 4.7 and Fig. 4.8 are the experimental results we collected while the curves are generated from our performance model, and the results of the model are close to the experimental results. We can see that accumulating more messages in one batch can indeed increase the packet throughput, but the improvement is less when N_b grows. E.g. when sending messages to the topic with 3 partitions, increasing N_b from 1 to 20 can boost the throughput by a factor of approximately 6.75, while configuring N_b from 20 to 40 only improves the throughput by 37%.

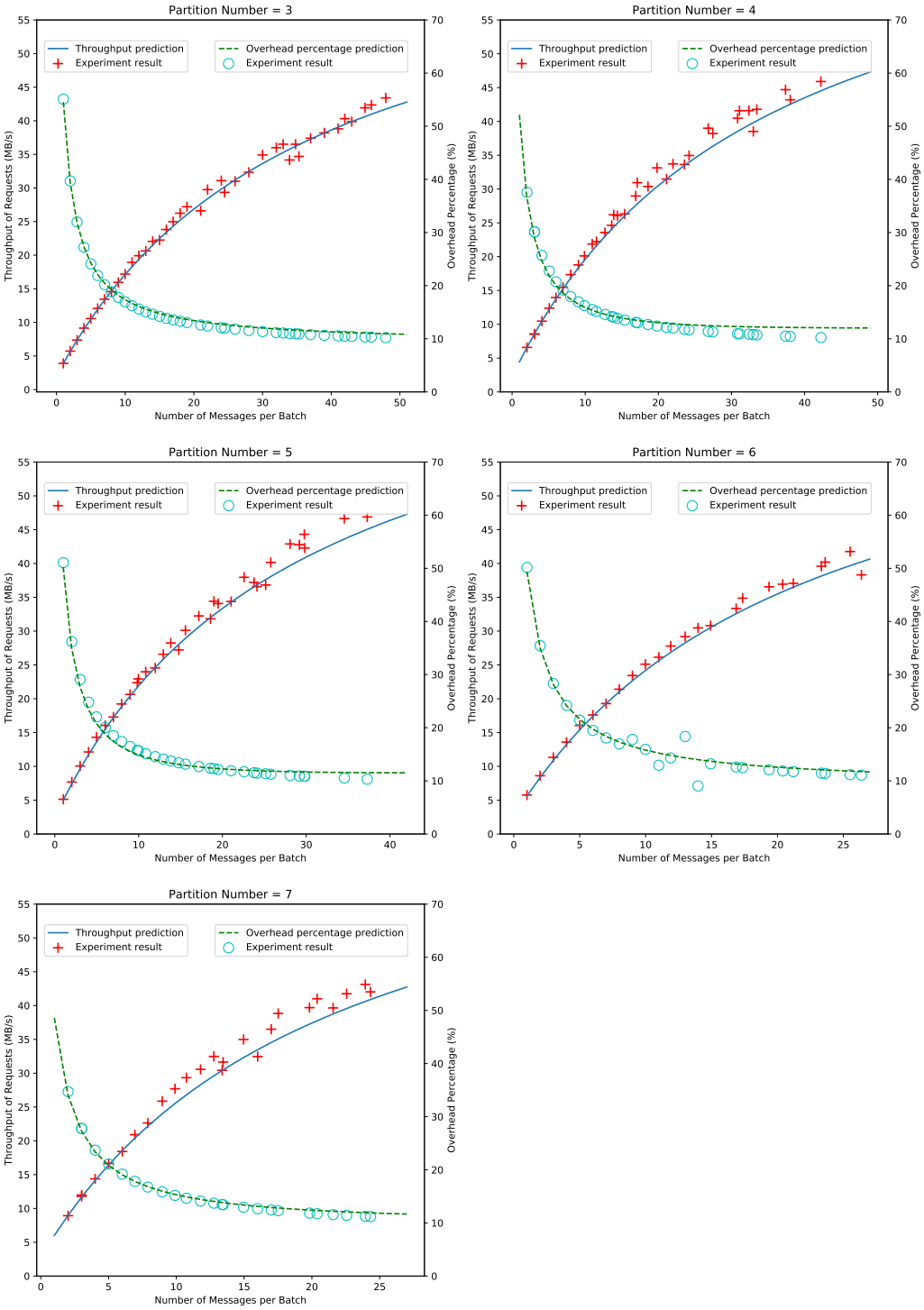

Figure 4.7: Experiment and prediction results of producer-send packet throughput with number of partitions from 3 to 7

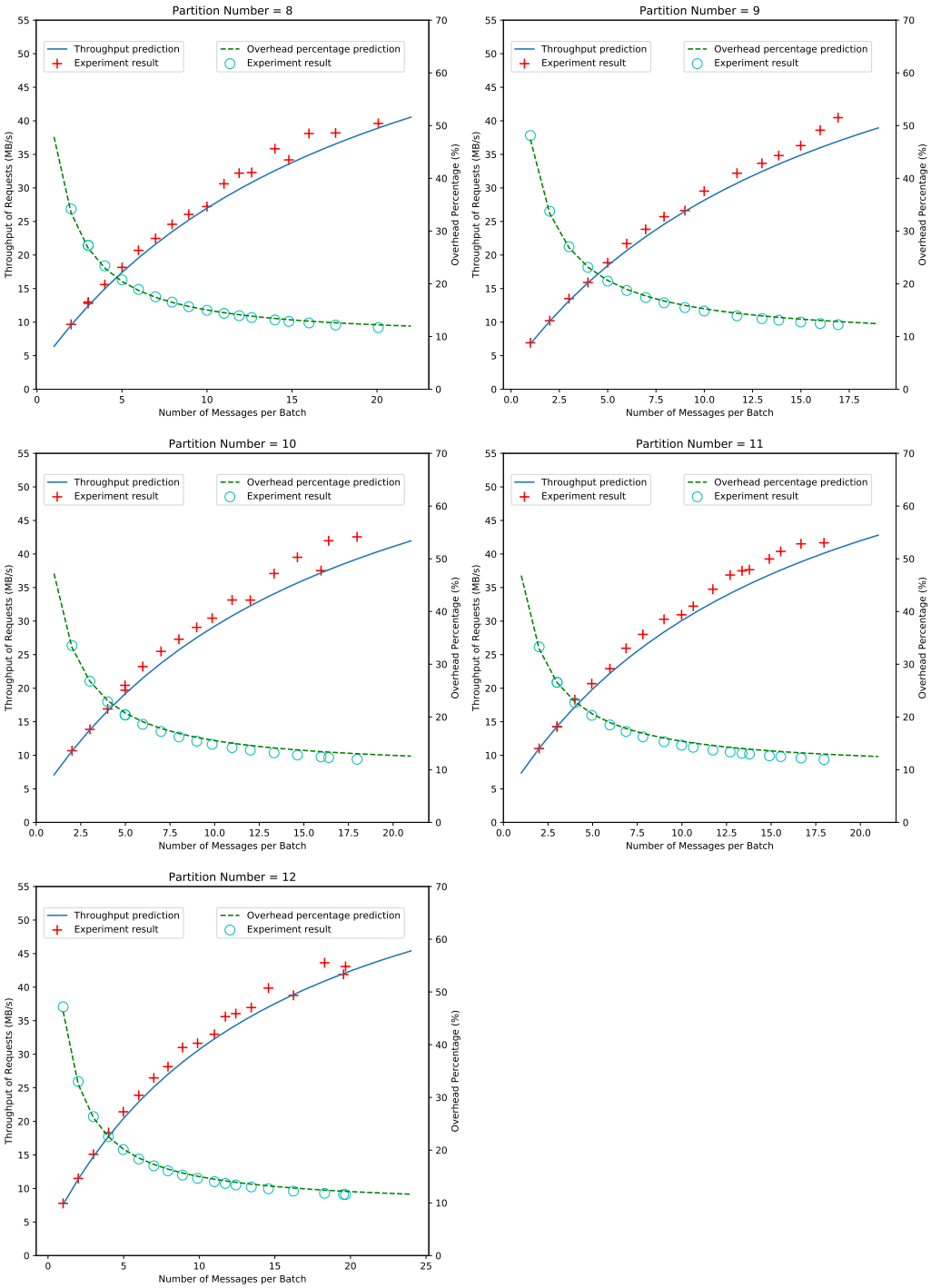

Figure 4.8: Experiment and prediction results of producer-send packet throughput with number of partitions from 8 to 12

We employ the mean absolute percentage error (MAPE) to measure the prediction accuracy of our model, which is given by:

$$MAPE = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{E_i - M_i}{E_i} \right| \tag{4.3}$$

where E_i is the experimental result and M_i is the prediction result from our model, and n is the number of fitted points. For the results of packet throughput, the MAPE is close to 2% when the throughput is below 25 MB/s, and with higher throughput the MAPE is still less than 10%. Compared to sending messages without batching (i.e. 1 message per batch), collecting 10 messages per batch can sharply decrease the relative overhead from over 50% to less than 15%, as illustrated in Fig. 4.7 and Fig. 4.8. However, the relative overhead remains stable around 10% after the throughput of packets reaches 30MB/s. The MAPE of the relative overhead also achieves 10%. The network bandwidth quota for a user of Kafka as a service is limited, and our performance model can help the user to choose a suitable number of partitions and batch size to achieve good throughput. E.g. if the partition number is set to 3 and the quota is 20 MB/s, our prediction results indicate that setting N_b to 12 can achieve a packet throughput of 19 MB/s, and adding more messages to the batch will risk exhaustion of the bandwidth and exceeding the available memory.

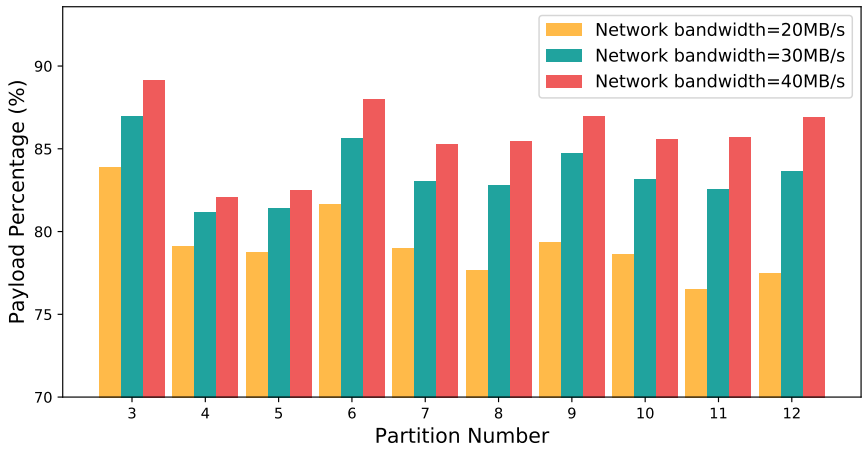

Figure 4.9: Partition number and relative payload of producer-send packets given different network bandwidth quotas

For the choice of a good number of partitions in a topic, we use our model to study the effect of partition number on the relative payload of producer-send packets, given network bandwidth quotas of 20 MB/s, 30 MB/s and 40 MB/s. The batch size in each test is configured to approach the highest

throughput at the quota limit, therefore the network bandwidth is almost fully utilized. As the results in Fig. 4.9 illustrate, we can observe that for any topic, assigning a higher network bandwidth quota achieves higher relative payload. Since the servers in our experiments are considered homogeneous, we define the topics with partitions evenly distributed among brokers as the load-balanced topic, and in our tests the load-balanced topics are those with the number of partitions equal to $3N$, where N denotes a non-zero positive integer. We can see from Fig. 4.9 that given the same network bandwidth quota, the relative payload of load-balanced topics decreases as the number of partitions increases. This happens due to the overhead of maintaining the extra partitions' metadata. Observing relative payload of the topics that are not load-balanced we find that the relative payload is always below their adjacent load-balanced topics. Thus we suggest the users try to avoid creating topics with the number of partitions that are not load-balanced among brokers, specifically when the servers are homogeneous.

4.3.3 Disk Space Evaluation

Limited disk space on cloud servers is an important concern of users, especially when the producer client runs a heavy load for a long time. Our performance model can predict the used storage of disk on each broker over time, and by observing the prediction results the user can set a proper value for the retention configuration parameters. Fig. 4.10 depicts the used disk space by messages on each broker, after the producer client runs for 24 hours with 20 MB/s network bandwidth quota. The line in this figure specifies the production time on each broker when the size of stored messages on disk reaches 500 GB.

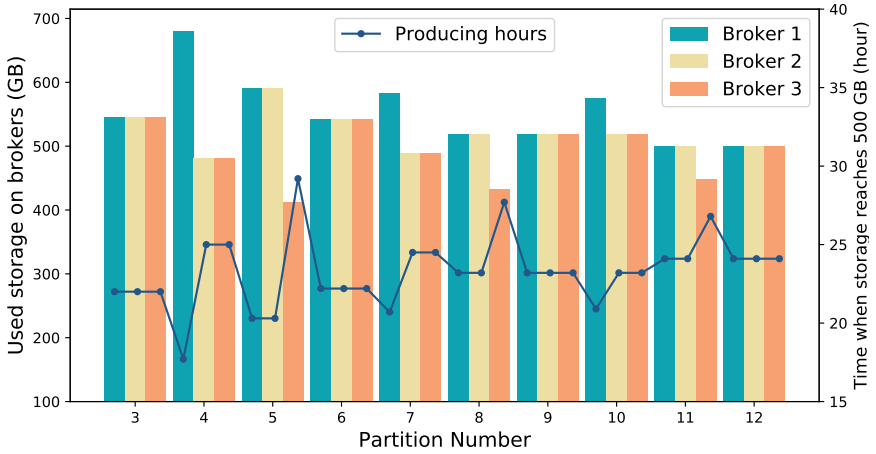

Figure 4.10: Used storage on brokers after producing messages for 24 hours

Apparently, the used disk space by messages is almost equal on each broker when the topic is load-balanced, while the other topics see imbalanced sizes of stored messages among brokers. The difference of used disk space among brokers is shrinking as the number of partitions in those imbalanced topics grows. E.g for the topic with 4 partitions the difference of storage between two brokers can be around 200 GB, yet the difference in the topic with 11 partitions is 50 GB. From observing the results of production time, the user can set appropriate configuration parameters to control the retention time of the messages on disks. For the topics that are not load-balanced, there will be some brokers that reach a certain size of stored messages much earlier than the other brokers, and the user has to specify the retention parameter on those brokers.

4.3.4 End-to-end Latency

In our latency experiments, we first used a small fraction of the web server log data to collect the service time data for the three types of requests mentioned in Section 4.2.4 to evaluate the end-to-end latency. The producers and consumers are run on two different PCs and we use the JMX (Java Management Extensions) tool to collect the total time to store a producer packet, the time to send replicas to other brokers, and the time to handle a fetch request from the consumer.

We use the fitting tool HyperStar2 [96] to obtain the phase-type distribution of service times including the seen correlation and use the fitted results in our queueing model introduced in Section 4.2.4. The probability density function of the service time and the fitted process are shown in Fig. 4.11.

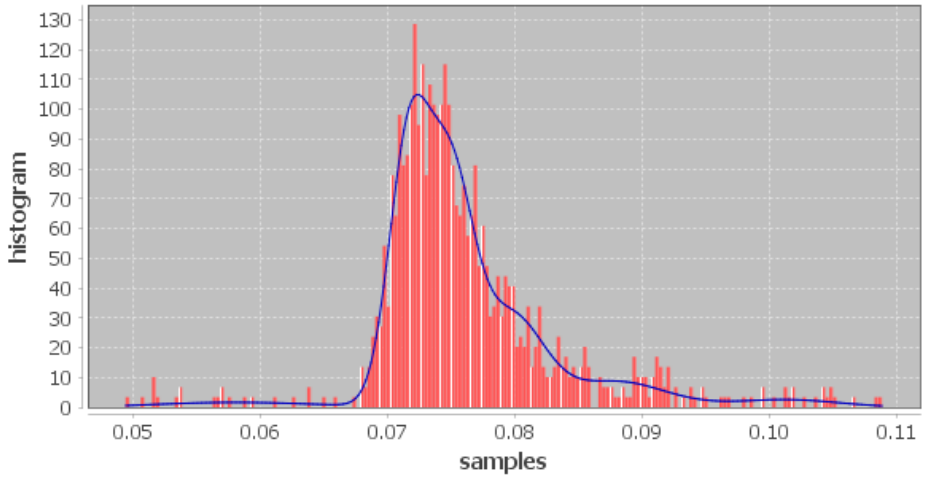

Figure 4.11: Probability density function of the service time distribution fitting

In our experiments, inspired by [46], packets are sent from producers with exponentially distributed inter-packet times. In Fig. 4.12 we compare the experimental results and the numerical results from the queueing model under different arrival rates.

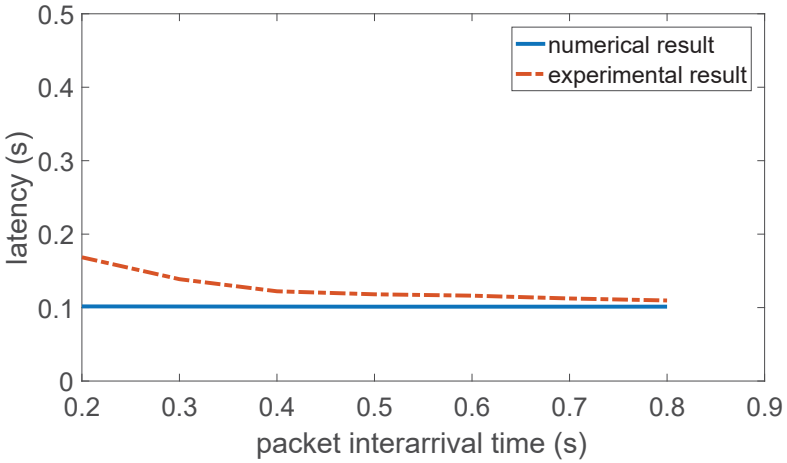

Figure 4.12: Latency results under different arrival rate

We can observe that the prediction model underestimates the mean end-to-end latency of packets, especially when the interarrival time is less than $0.3s$. The model works well with lower arrival rate, as shown in Fig.4.12, when the message interarrival time is larger than $0.4s$, the MAPE of the numerical result could be less than 10%. The difference has a decreasing trend with lower load.

4.4 Summary

In this chapter we propose a performance model to predict the performance of Kafka messaging system. Our model provides users an easy and reliable way to study the effects of the configuration parameters on the performance of their Kafka system. By tuning the batch size and the number of partitions in the model, users can observe the changes of producer throughput and relative payload under heavy load. The mean absolute percentage error is close to 2% when the network bandwidth quota does not exceed 25 MB/s, and this can help the user choose appropriate configuration parameters. The model also predicts the disk occupancy over time for changing number of partitions, through which users can adjust the retention parameters in Kafka to avoid using up disk space on cloud servers. We use $M/PH/1$ queueing model to evaluate the mean time a packet travels through a Kafka cluster, which is the main concern when collaborating with real-time applications.

Chapter 5

Reliable Data Delivery

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 chapter 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 [87]. We solve the above problems by building a prediction framework which can accurately predict the reliability metrics we define, given certain configuration parameters.

5.1 Reliability Metrics

In this section we 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.

5.1.1 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. 5.1. 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 5.1. 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.

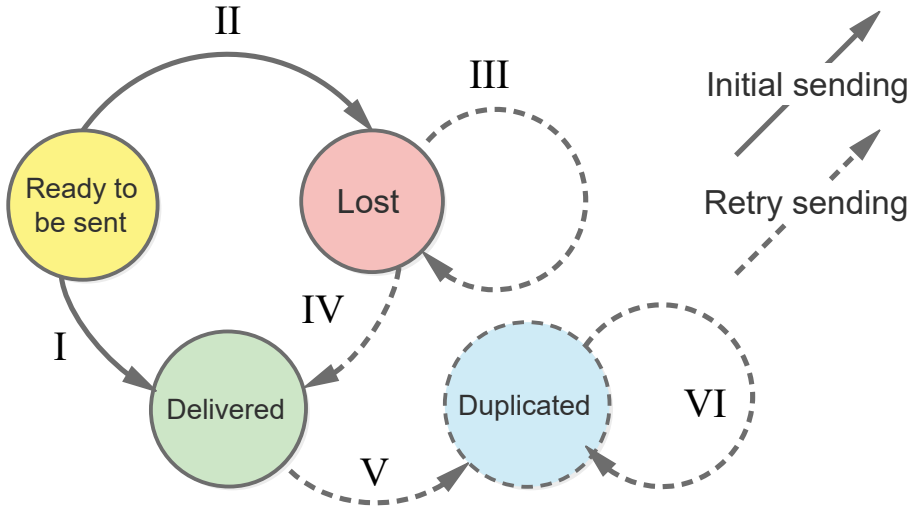

Figure 5.1: The states of a message in Kafka

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 $\tau_r \cdot$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).

It is worth noting that there exist other cases apart from all the cases listed in Table 5.1, e.g. a message might firstly be *Lost*, then it is *Delivered* through retrying mechanism, however, due to the failure of receiving response from brokers, the message is resend a few times and finally turns to *Duplicated*. However, the probability of such cases is rather low according to our experimental results, and in this research we focus only on the final state of each message.

Table 5.1: All Possible Cases of a Message Delivery in Kafka

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

5.1.2 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 cargo (messages) from upstream applications to the cluster efficiently and safely. Each cargo should be carried to its destination exactly-once. In Table 5.1 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 [62]. 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 $P_d = P(Case5)$.

5.1.3 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. 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. As introduced in Section 1.2, 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.

We mainly introduce the following features for the prediction model based on the results sensitivity analysis: (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 5.4 we introduce those main features through analysing their impact on the reliability metrics in various application scenarios.

5.2 Testbed Design

In Chapter 4 we develop a Kafka cluster on several PCs, however, it is insufficient for the reliability analysis. This is because the network condition is difficult to maintain and the fault injection is inconvenient. Therefore, in this section, we introduce the Kafka testbed built on Docker containers, which is used in this Chapter as well as the next Chapter.

We need to 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 remains unclear. In this section we introduce our testbed of Kafka built using Docker, an industrial-grade lightweight virtualization technology [21]. As depicted in Fig. 5.2, the testbed consists of four parts: the Kafka system based on Docker containers, the *message generator*,

the *network emulator* and the *statistical analysis tool*.

5.2.1 Testbed Components

All components of a Kafka system testbed use Docker containers. 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 [48]. 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.

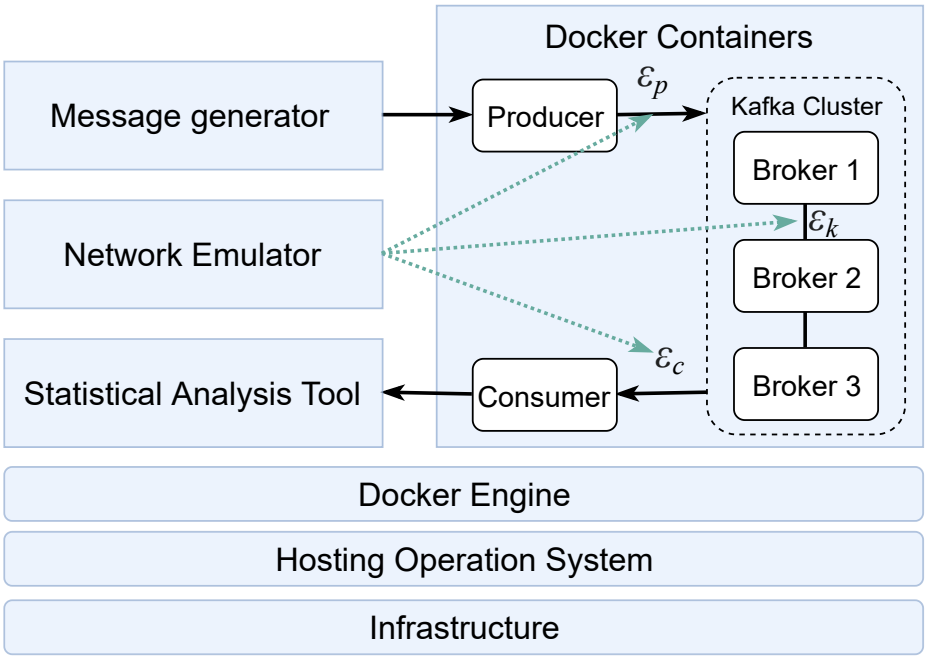

Figure 5.2: The architecture of the Kafka testbed based on Docker containers

5.2.2 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. To obtain the *produce time*, 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 the *commit* and *consume time*.

5.2.3 Network Emulator

As illustrated in Fig. 5.2, 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 [59]. 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 [87].

5.2.4 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, which is used in Chapter 6. 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.

5.3 Prediction Model

5.3.1 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.

$$\{\hat{P}_l, \hat{P}_d\} = f(M, S, D, L, Confs), \tag{5.1}$$

where M stands for the size of the message which the producer tries to send, and S represents the timeliness of the message. The value of timeliness S varies depending on the nature of the application, and it is usually defined by users. 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, including delivery semantics $acks$, Batch size B, Polling interval δ and Message timeout T_o. 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. 5.3. 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, we inject network faults using the *network emulator*, where in each test the network delay D and packet loss rate L are fixed. Then for those effective features in the left oval of Fig. 5.3, we choose the proper values studied in the normal network environment thus their impact can be neglected. The effective

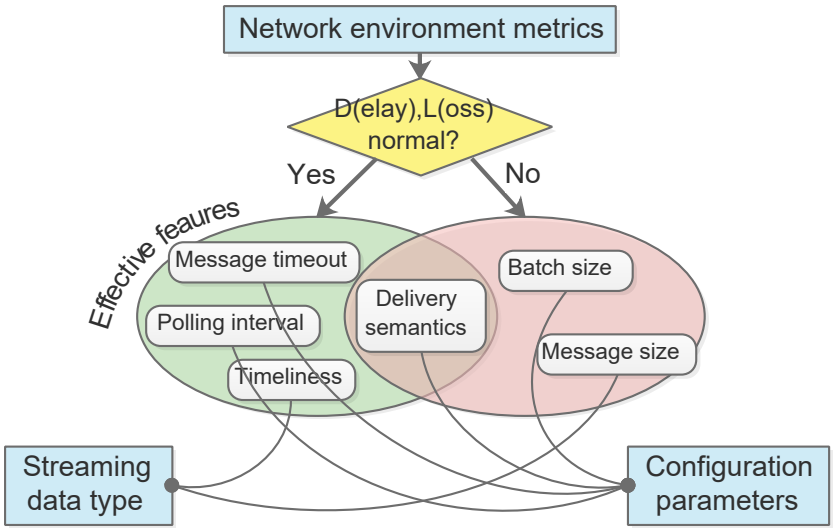

Figure 5.3: Training data collection design

features we studied in the abnormal cases are listed in the right oval of Fig. 5.3. For each feature we specify the range of possible variables according to real world systems.

5.3.2 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. 5.5 to Fig. 5.7, where we compare samples of the test data with the

predicted results.

The way we construct the ANN based prediction model is depicted in Fig. 5.4. The left part of the figure shows how we collect the training data using related modules. The right part of the figure illustrates the inputs and outputs of the prediction model.

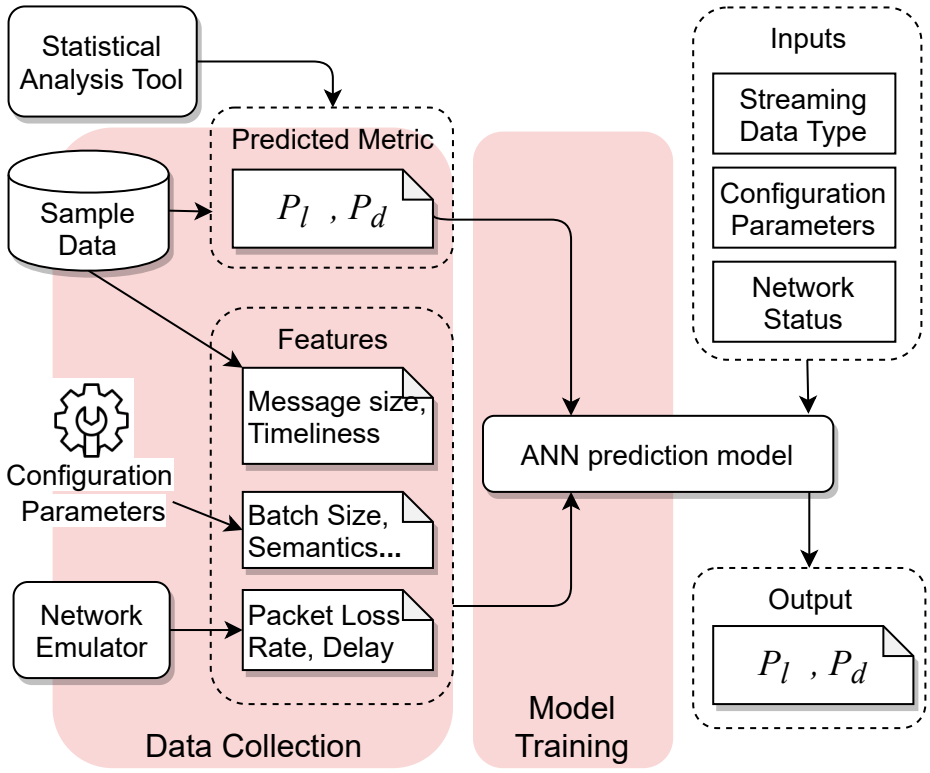

Figure 5.4: The process of data collection and ANN model training

5.4 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:

$$\gamma = \omega_1 \varphi + \omega_2 \mu + \omega_3 \left(1 - P_l\right) + \omega_4 \left(1 - P_d\right), \left(\sum_{i=1}^{4} \omega_i = 1\right). \tag{5.2}$$

Here $\omega_i (1 \leq i \leq 4)$ represents the weight assigned for each metric. The higher the weight is, the more important the metric is from the user's perspective. 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). These two performance metrics φ and μ can be predicted according to Section 4.2.1. In this chapter 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 $\omega_3 = 0.1$.

5.4.1 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. 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. 5.5 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 ($P_l \approx 63\%$). In other words, small messages ($M \leq 200bytes$) are more likely to be lost under at-least-once delivery. 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 be observed by Wireshark, an open-source packet analyser. 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. 5.5 in each experiment at-least-once can save approximately 3000 more messages

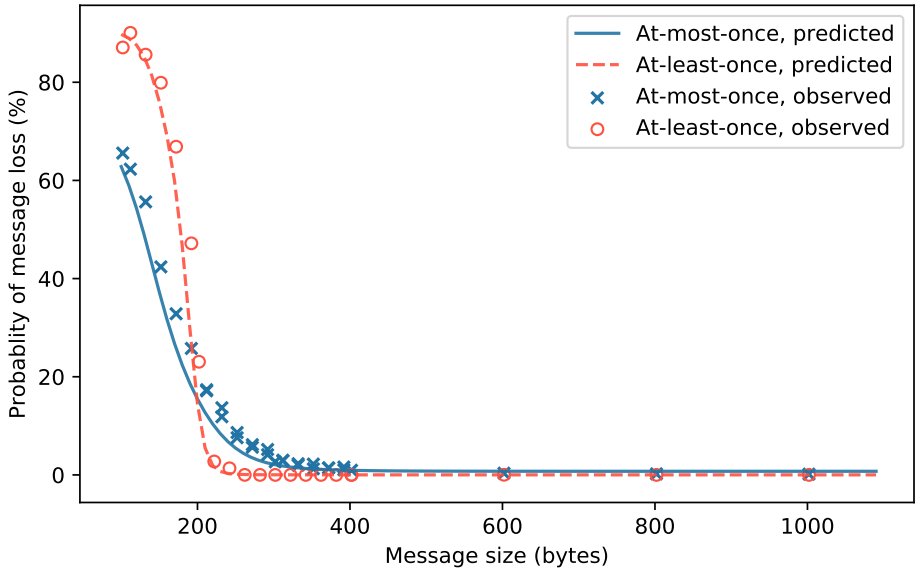

Figure 5.5: The message size M and its probability of loss P_l for network packet loss rate $L = 19\%$

than at-most-once.

5.4.2 Message timeliness

In real-time application scenarios 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 the message timeout 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. 5.6. In this case using the at-least-once delivery can significantly reduce P_l.

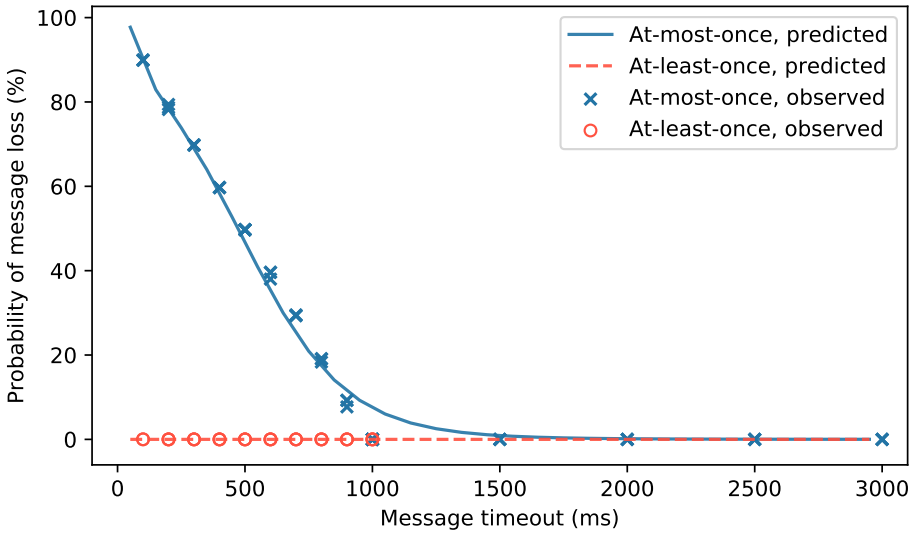

Figure 5.6: The configuration parameter message timeout T_o and the probability of message loss P_l, no network fault injected

5.4.3 Scalability of a producer

In the example of Fig. 5.6, 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 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. 5.7 indicate that increasing the polling interval δ can effectively avoid message loss, while T_o is fixed at $500ms$. If the polling interval δ is configured as 0, then the producer is fully loaded because it keeps polling messages without waiting. 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%.

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.

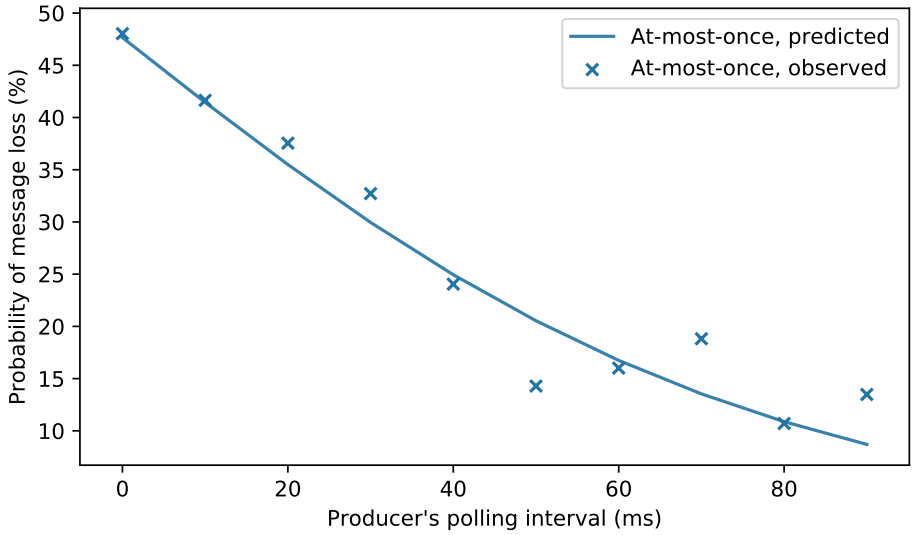

Figure 5.7: The configuration parameter polling interval δ and the probability of message loss P_l, no network fault injected, $T_o = 500ms$

5.4.4 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. The predicted effects under at-most-once and at-least-once delivery semantics are depicted in Fig. 5.8 and Fig. 5.9.

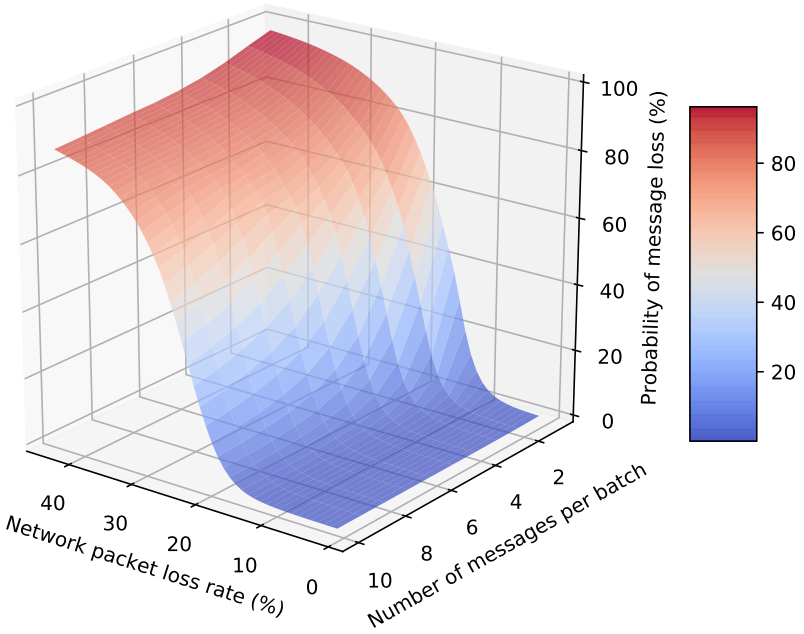

Figure 5.8: The configured batch size and the probability of message loss with at-most-once delivery, injected with various packet loss rate

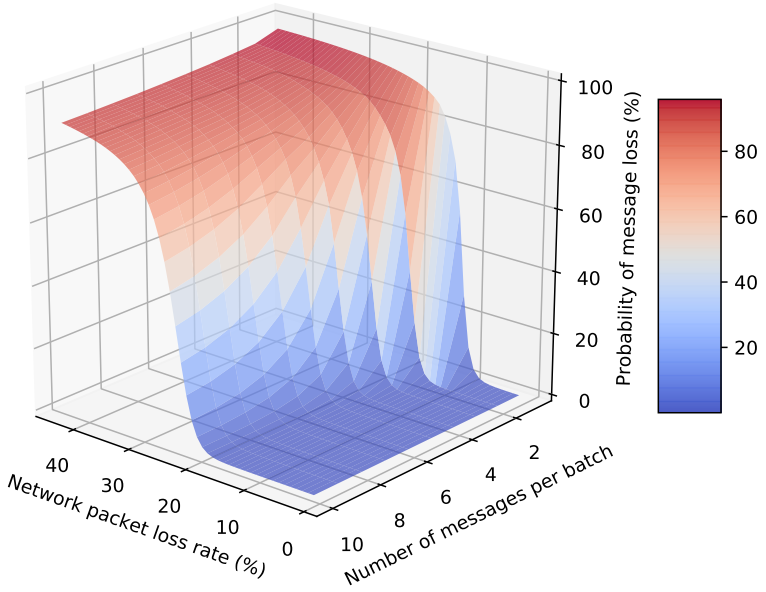

Figure 5.9: The configured batch size and the probability of message loss with at-least-once delivery, injected with various packet loss rate

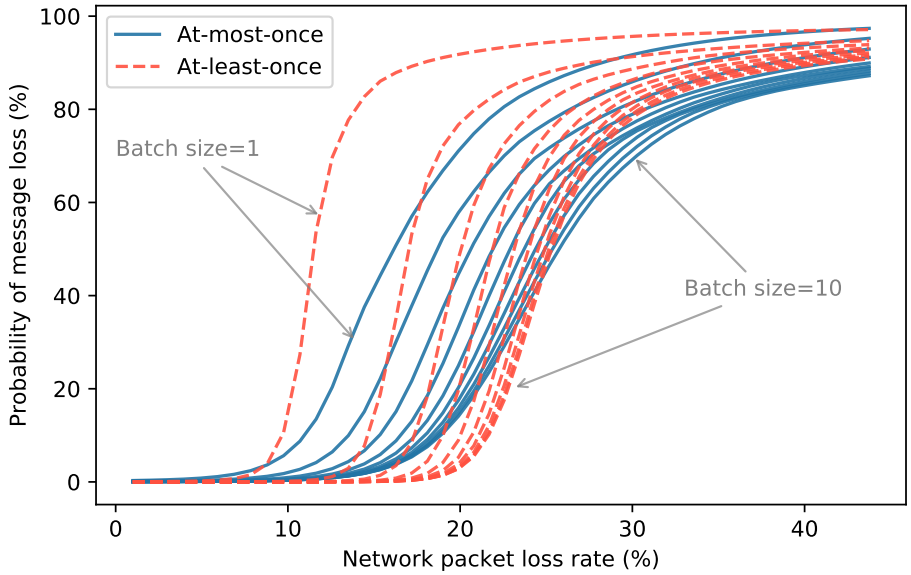

Figure 5.10: Comparison of different batch size and delivery semantics, injected with various network packet loss rate

From Fig. 5.10 we have a clearer view of 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.

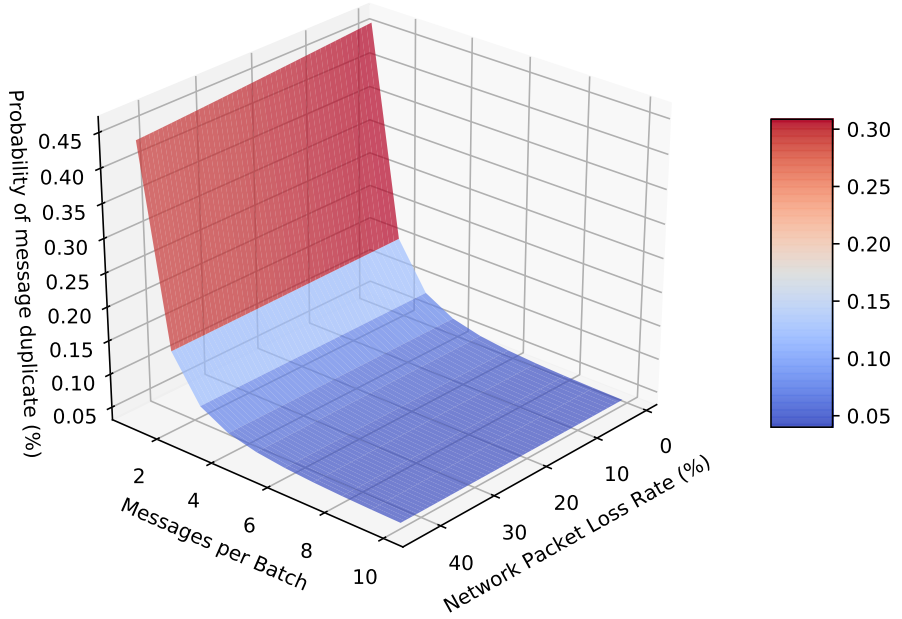

Figure 5.11: The configured batch size and the probability of message duplicate with at-least-once delivery, injected with various network packet loss rate

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 (5.2) all show strong correlation with batch size B. Larger B results in lower μ and also increases the end-to-end latency [35]. The impact of B on the probability of message duplicate P_d is shown in Fig. 5.11. It is obvious that P_d can be reduced by batching, while no strong correlation between P_d and L is observed.

5.5 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.

The predicted reliability metrics can be obtained according to Equation (5.1). Combining with the model in Chapter 4, we obtain the weighted KPI γ from Equation (5.2). 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 [44]. Thus in our experiments we check γ every other time interval (i.e. every 60 seconds). The dynamic

configuration process is illustrated by Algorithm 1 bellow.

Algorithm 1: Dynamic Configuration

Input: KPI weights, $\omega_i, 1 \leq i \leq 4$; streaming data type, M, S; network environment metrics, D, L; configuration parameters, $acks, B, T_o, \delta$

Output: New configuration parameters, $acks', B', T_o', \delta'$

```
calculate initial KPI γ;
specify user defined KPI γ';
while true do
    monitor current network condition;
    predict φ, μ, P̂_l, P̂_d;
    calculate initial KPI γ using Equation 5.2;
    for parameter ∈ {acks, B, T_o, δ} do
        while γ < γ_user do
            parameter' = parameter ± Δ;
            predict γ using parameter';
        end
    end
    Update current configuration to parameter';
    Wait 60 seconds;
end
```

The network status in our experiment is depicted in Fig. 5.12. To simulate a realistic network environment, the network delay follows a Pareto distribution [116], while the network packet loss rate is generated from the Gilbert-Elliot model using Linux netem tool [20]. 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 5.2. 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. Considering those related facts, our suggested values of weights ω_i are listed in Table 5.2.

According to Equation (5.1), since the features M, S, D and L are all some functions of the

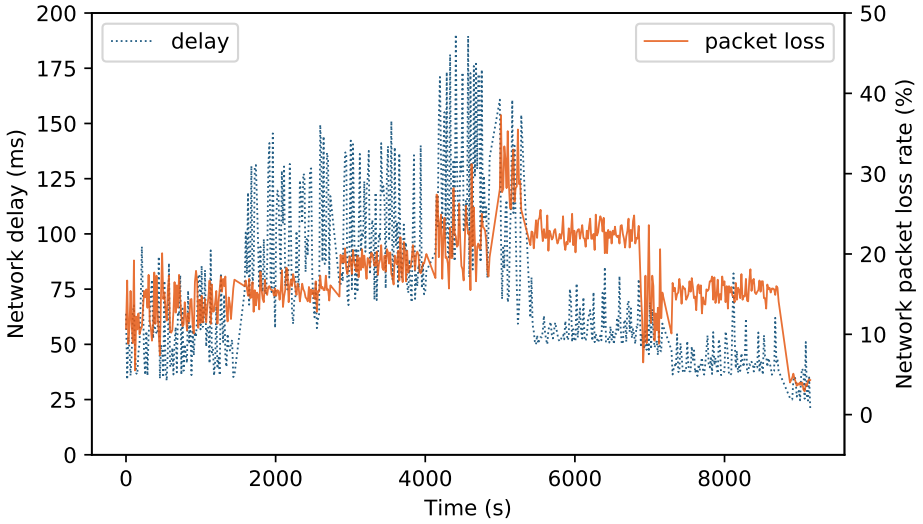

Figure 5.12: Network connection between Kafka producer and cluster in the dynamic configuration experiment

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:

$$R_l = \frac{\int \lambda(t)P_l(t)dt}{\int \lambda(t)dt}, \quad R_d = \frac{\int \lambda(t)P_d(t)dt}{\int \lambda(t)dt} \tag{5.3}$$

Table 5.2: 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 5.2 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 dupli-

cate 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.

5.6 Summary

In this chapter we utilize the Kafka testbed we created for exploring the approach of building a prediction model. 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. Then the Docker-based testbed of Apache Kafka is introduced, which helps us collect the training data. With the testbed it is easy and fast to build or restart a Kafka cluster, and it capable of various experiments with fault injection. 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).

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.

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 model in Chapter 4, 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 for adjusting these parameters is proposed in our experiments to evaluate the effectiveness of the prediction model.

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.

Chapter 6

Reactive Batching Strategy

From Chapter 4 and Chapter 5 we know that 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. Batch size also have impact on the reliability of data delivery under unstable network condition. 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 chapter, we address problems in the performance measurement of existing stream processing systems. Since in Chapter 4 we only 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.

6.1 Problems and challenges

In Chapter 4 we introduced the concept of packet latency, which measures the latency of an entire packet. However, this metric is insufficient for studying the real-time strategies in Apache Kafka. A proper measurement of each message's end-to-end latency is required for guaranteeing the message timeliness. In this chapter, we introduce the problems of current performance metrics and challenges in the batching strategy of Kafka.

6.1.1 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 [39,43]. In this chapter we give an explicit definition of the end-to-end latency, denoted by L_K, and introduce its major components in detail. Fig. 6.1 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.

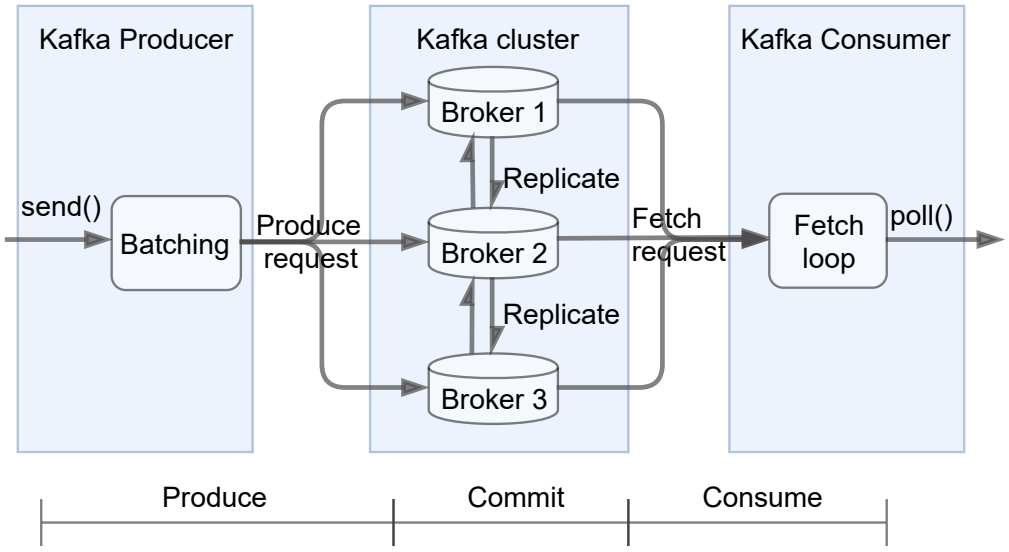

Figure 6.1: The components of Kafka's end-to-end latency

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.

6.1.2 Batching methods

From Fig. 6.1 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 [35]. 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, as we discussed in Chapter 4. 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 optimising 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 [35]. Kafka provides both configuration parameters to control batching. In this chapter, 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 analysed statistically. We use l_1 to denote the *produce time* in Fig. 6.1, and l_2 to denote the *commit* and *consume time*, thus $L_K = l_1 + l_2$. Fig. 6.2 illustrates the impact of B_S on the mean and standard deviation of L_K, l_1 and l_2.

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 2.2, 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 this thesis.

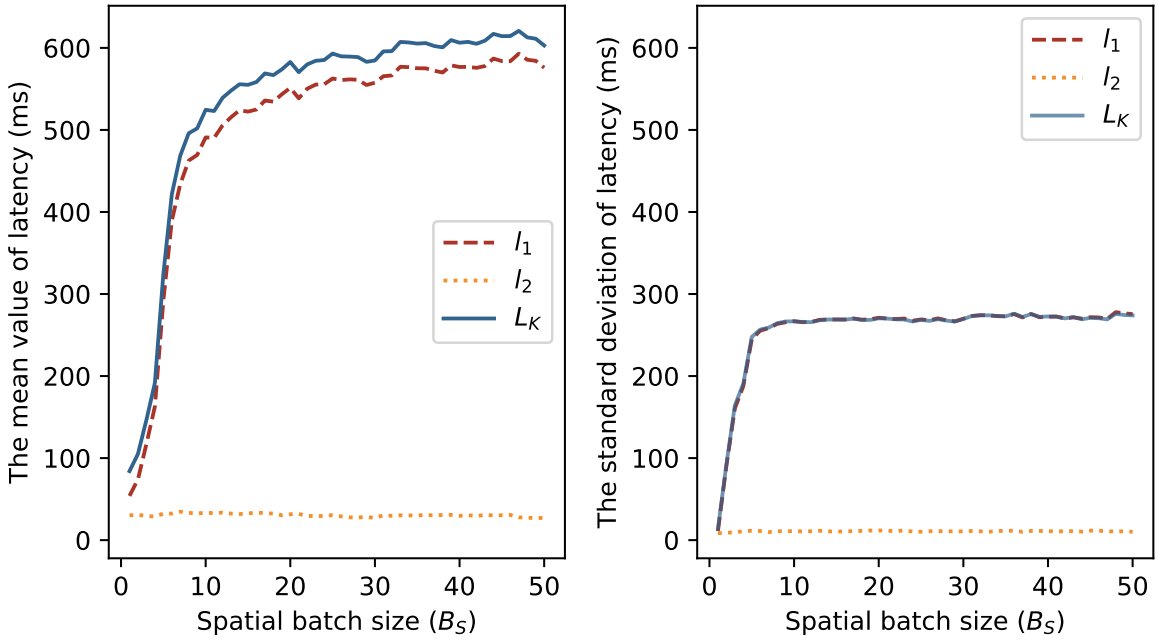

Figure 6.2: The impact of spatial batch size on latencies when temporal batch size $B_T = 1000ms$

6.1.3 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 [75]. 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 [39, 43, 109]. We use σ_K to denote the standard deviation of L_K, as depicted on the right of Fig. 6.2. 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. 6.3 shows the distributions of L_K in three selected individual tests, with B_S configured as $1, 4$ and 10, respectively.

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 distri-

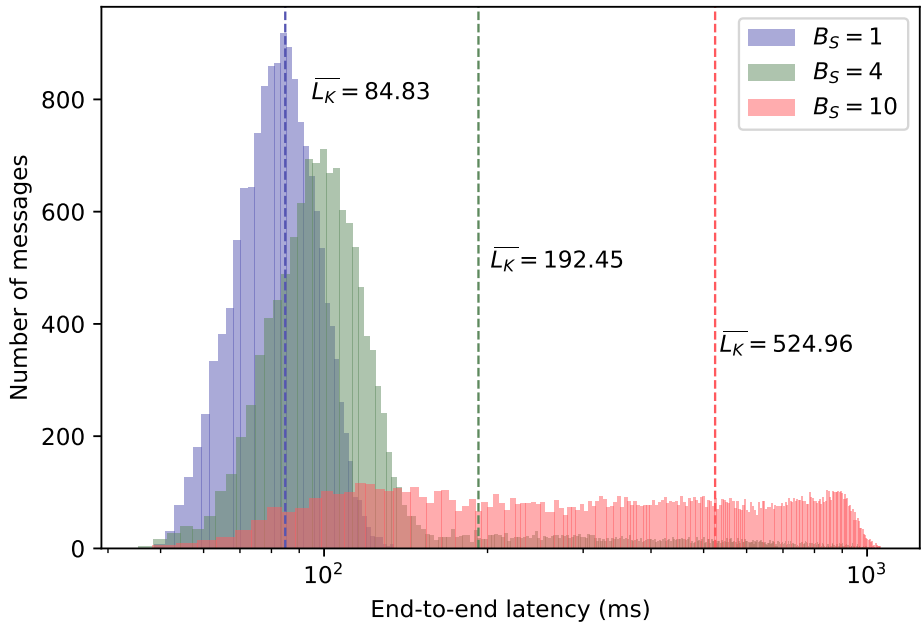

Figure 6.3: The distribution of end-to-end latency with different spatial batch size

bution 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. 6.3), 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.

6.2 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 6.1.3, 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 Chapter 2: 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.

6.2.1 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 6.1, 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. 6.4. 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.

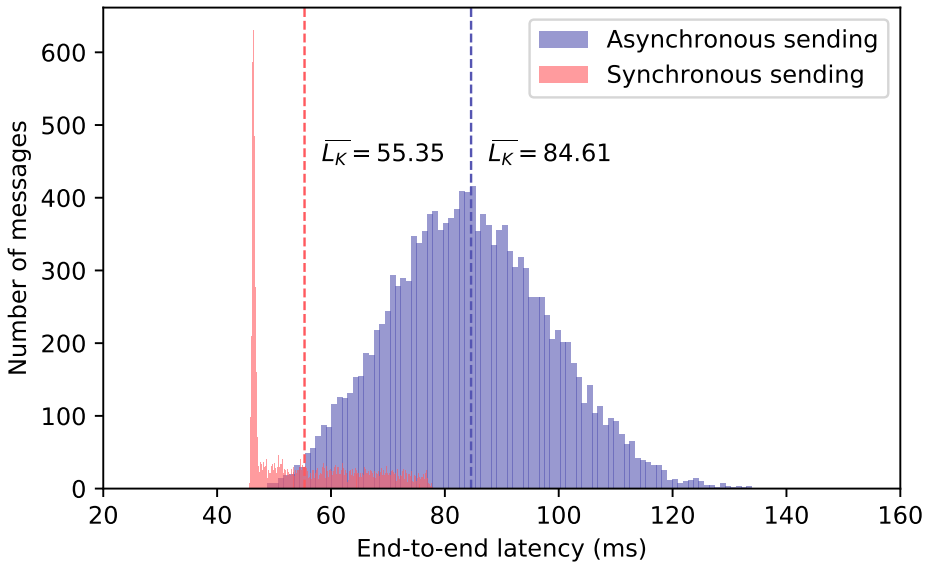

Figure 6.4: The distributions of L_K under synchronous and asynchronous sending

6.2.2 Violation rate prediction

Our hypothetical ideal solution to the problems introduced in Section 6.1 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.

$$\eta_v = f(\varepsilon_\star, B_S, B_T, \zeta_L) \tag{6.1}$$

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 5.2. 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 6.1.3 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) $F_\Omega(x)$:

$$\eta_v = P(L_K > \zeta_L) = 1 - P(L_K \leq \zeta_L) = 1 - F_\Omega(\zeta_L) \tag{6.2}$$

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.

6.2.3 Latency violation rate prediction

The workflow of building the above prediction model is depicted in Fig. 6.5. 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. 5.2 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 6.1.2.

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

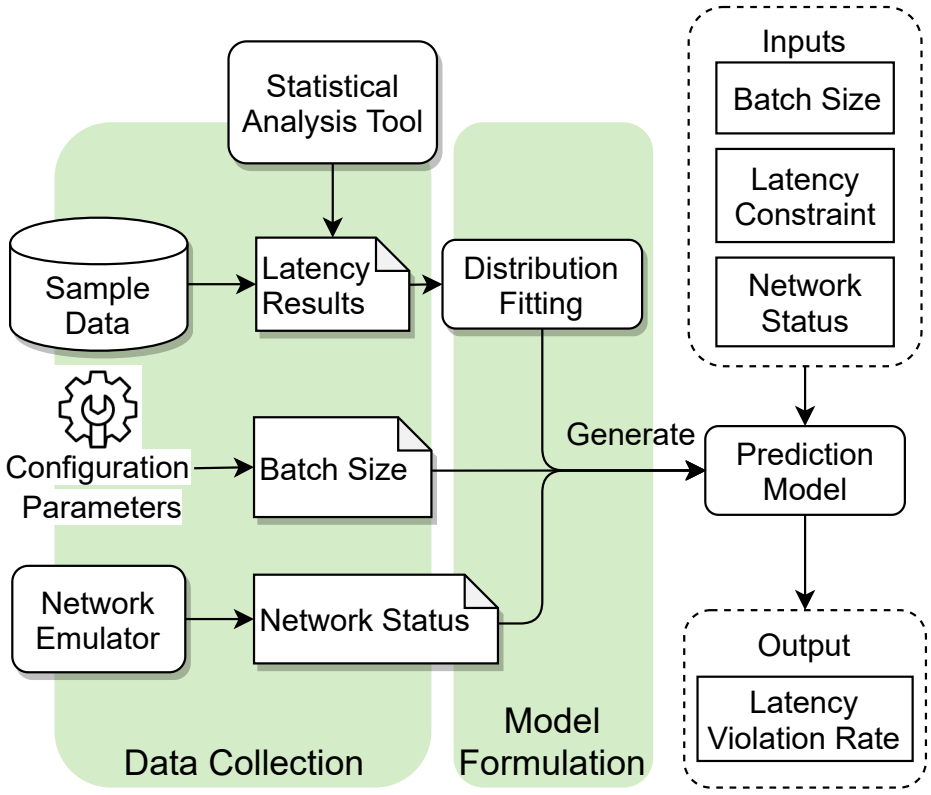

Figure 6.5: The prediction model of latency violation rate η_v

them in Fig. 6.6.

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 6.1.1:

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. 6.7.

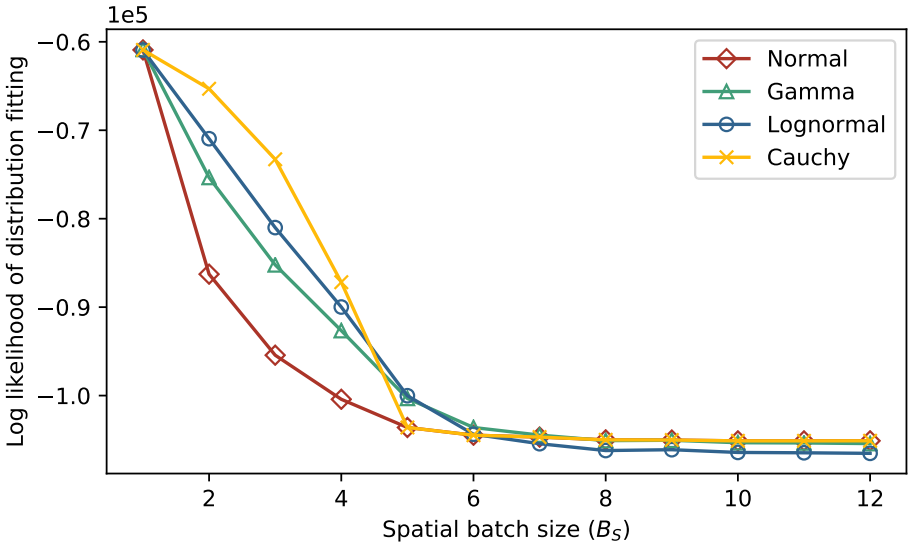

Figure 6.6: The log-likelihood θ of different types of distributions

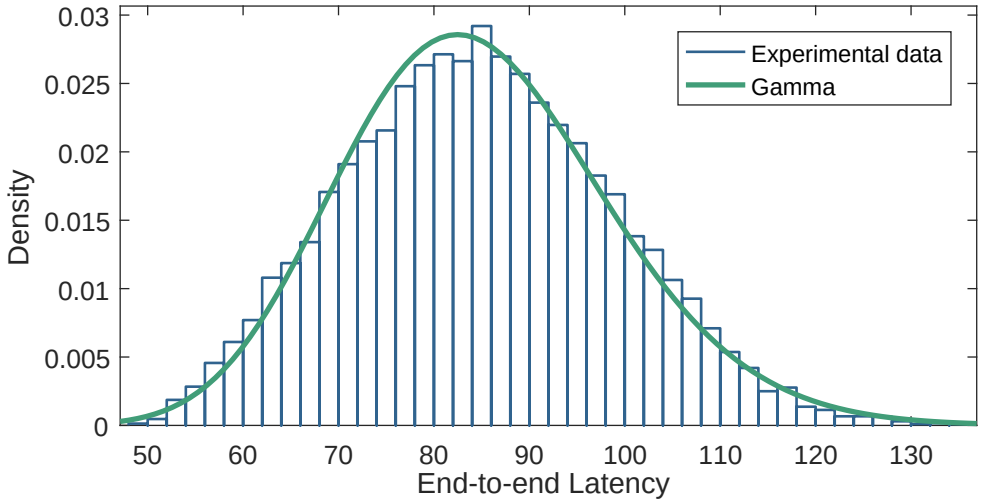

Figure 6.7: The PDF of distribution fitting result when $B_S = 1$

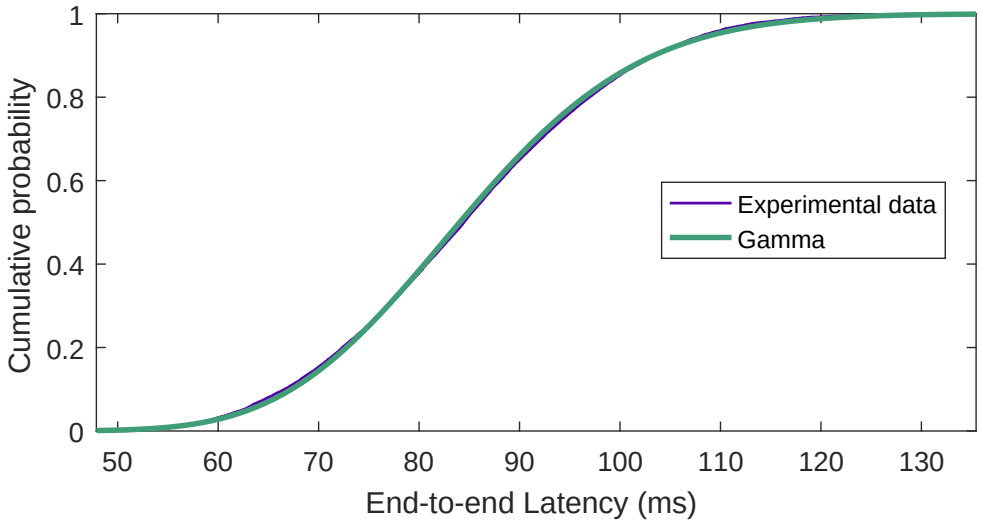

Figure 6.8: The CDF of distribution fitting result when $B_S = 1$

From Fig. 6.8 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:

$$MAE = \sum_{i=1}^{n} \frac{|\hat{\eta_v} - \eta_v|}{n} \tag{6.3}$$

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 (6.4).

$$\eta_v = 1 - F_\Omega(\zeta_L) = 1 - \frac{1}{b^a \Gamma(a)} \int_0^{\zeta_L} t^{a-1} e^{\frac{-t}{b}} dt \tag{6.4}$$

$\Gamma(a)$ 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. 6.6 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

Table 6.1: 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

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 [24, 96]. 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. 6.1. We can observe from Fig. 6.9 and Fig. 6.10 that the curves fit well.

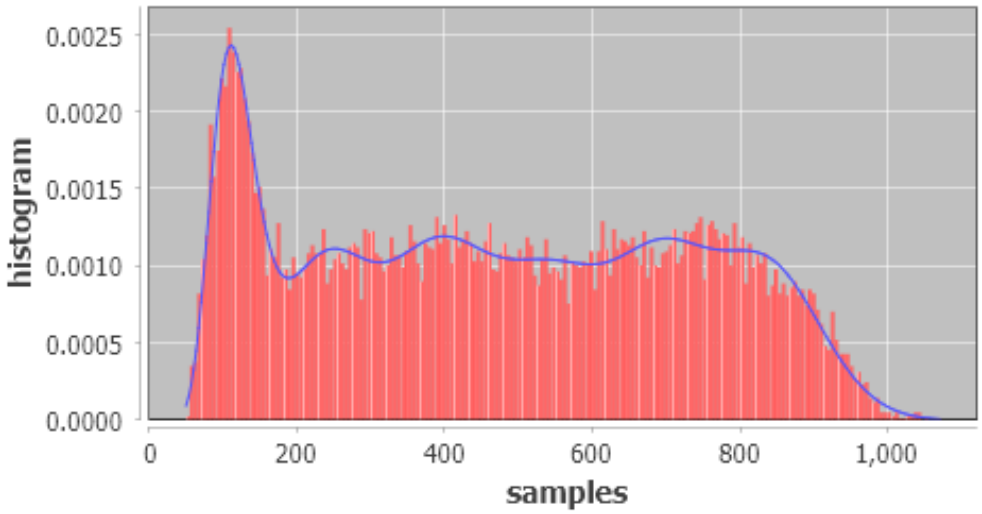

Figure 6.9: The PDF of distribution fitting with HyperStar2 when $B_S = 7$

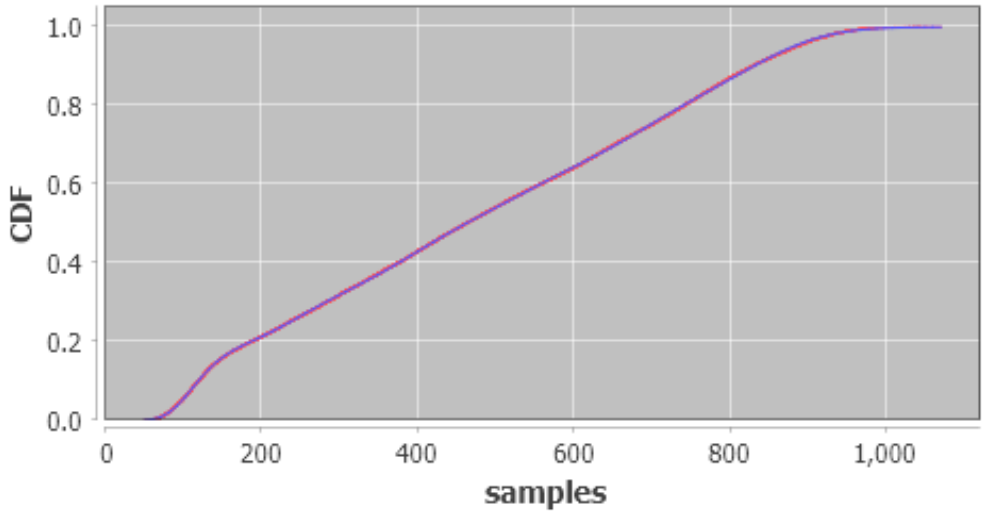

Figure 6.10: The CDF of distribution fitting with HyperStar2 when $B_S = 7$

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:

$$\hat{\eta_v} = 1 - F_\Omega(\zeta_L) = \boldsymbol{\pi} e^{\zeta_L \boldsymbol{T}} \boldsymbol{I} \tag{6.5}$$

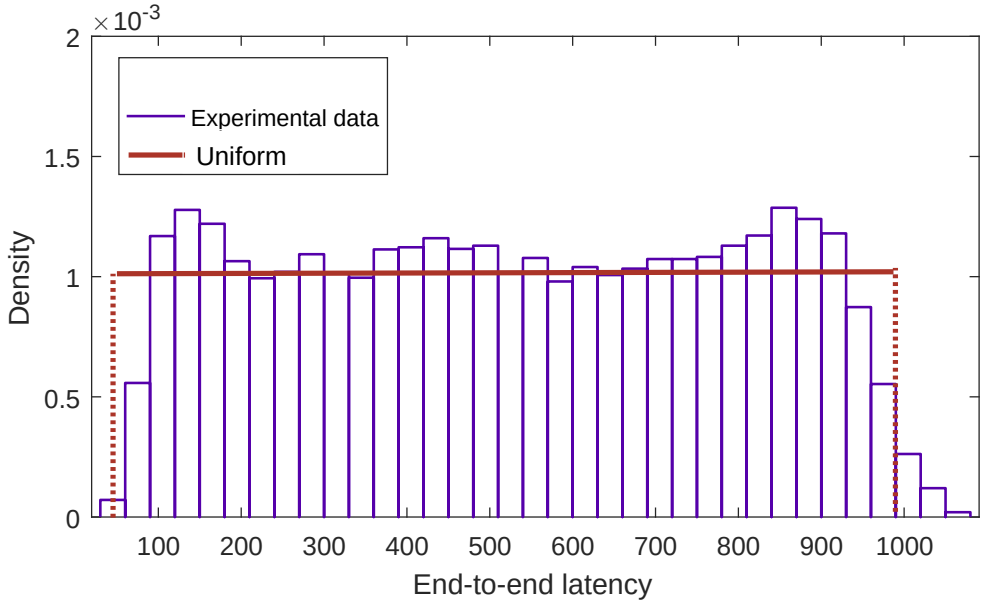

Figure 6.11: The PDF of distribution fitting result when $B_S = 11$

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 6.1.2. We can simply use uniform distribution to fit the the experimental data in this stage and obtain high accuracy, and the PDF and CDF fitting results of one example ($B_S = 11$) are illustrated in Fig. 6.11 and Fig. 6.12 respectively.

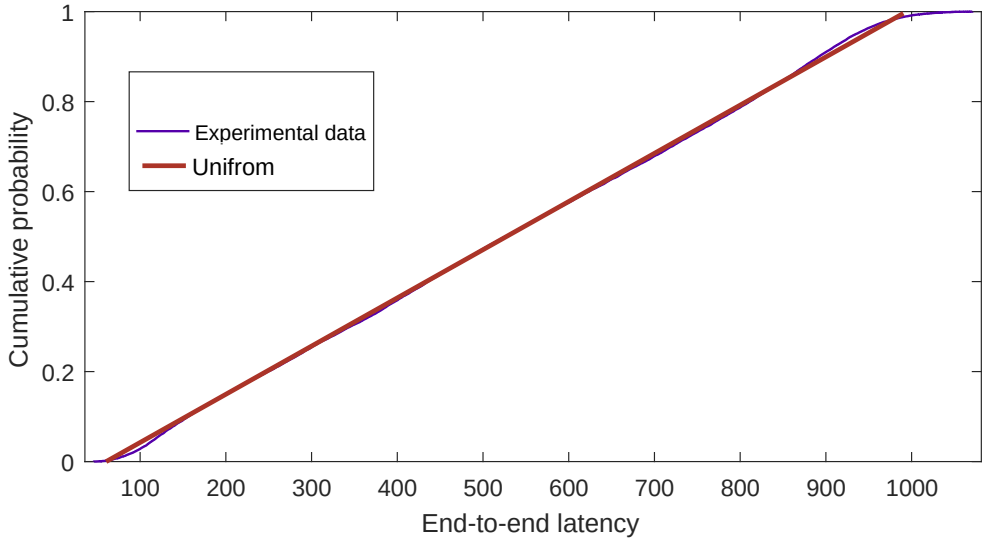

Figure 6.12: The CDF of distribution fitting result when $B_S = 11$

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:

$$\hat{\eta_v} = 1 - F_\Omega(\zeta_L) = \frac{\alpha L_{Kmax} - \zeta_L}{\alpha L_{Kmax} - L_{Kmin}} \tag{6.6}$$

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.

6.2.4 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 Chapter 4, thus in this chapter we predict χ_P according to the proposed model.

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:

$$\chi'_P = (1 - \eta_v)\chi_P \tag{6.7}$$

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 6.3 we evaluate this metric in the tradeoff between L_K and χ_P and compare it with other empirical methods.

Considering the case when network connections are unstable, we define the message loss rate η_l as the proportion of lost messages. From Section 5.4.4 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. The model proposed in Section 5.3.2 can be expressed as the equation:

$$\hat{\eta_l} = f(l_p, acks, B_S) \tag{6.8}$$

We choose to ignore message duplication because the impact is minor and most applications have an idempotence mechanism. 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 (6.7):

$$\chi'_P = (1 - \eta_v)(1 - \eta_l)\chi_P \tag{6.9}$$

6.3 Experimental evaluation

In order to evaluate the batching strategy in Section 6.2 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 [33]. The services of connected cars include traffic safety and cost efficiency and the demand for real-time processing is increasing [119]. 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 [47]. 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 [116]. 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 analyse measurements on wireless networks [20].

6.3.1 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 [75] 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. 6.13, 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.

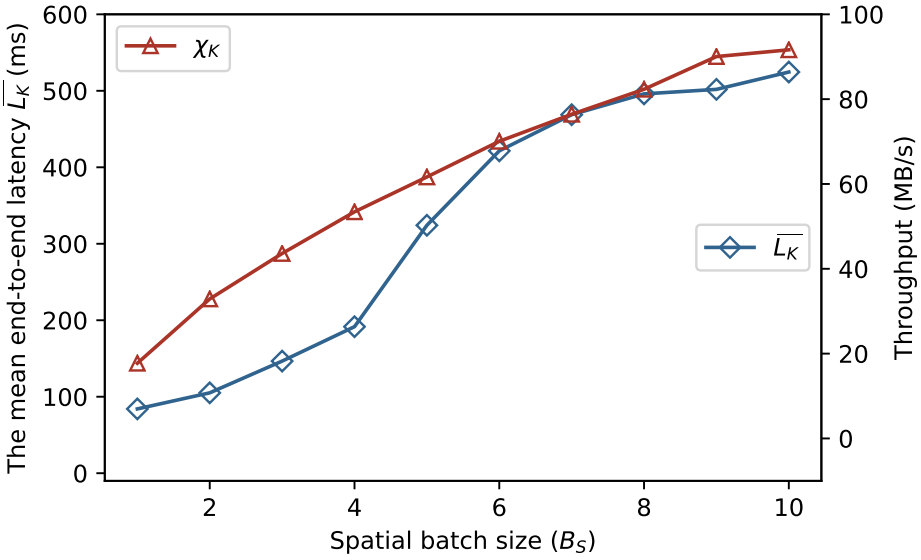

Figure 6.13: The impact of B_S on $\bar{L_K}$ and χ_P

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. 6.14. 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 $B_S = 5$.

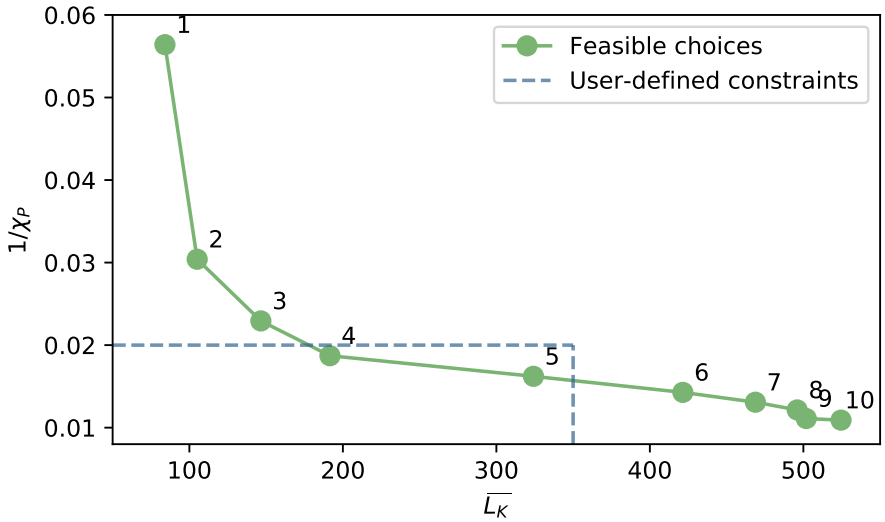

Figure 6.14: The Pareto font of throughput vs latency

With our batching strategy, the value of spatial batch size B_S that maximizes the timely throughput χ'_P is configured, and it is $B_S = 4$, according to the results illustrated in Fig. 6.15. This configuration improves the throughput of timely messages by 7.37MB/s, compared to the choice above.

6.3.2 Message success rate evaluation

In the experiment with network fault injection we assume that network status is known as Fig. 5.12 in Chapter 5. We check the network metrics including network delay and packet loss rate, $\varepsilon_p(d_p, l_p)$, 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 [44]. We substitute the operation and configuration parameters to Equation (6.9) and observe how the predicted result changes over different B_S. Then we reconfigure B_S to obtain the maximum χ'_P.

The overall message loss rate and timely throughput in this experiment are illustrated in the Table 6.2. We compare the results obtained via our reactive batching strategy with those using Pareto front based empirical methods, and fix the spatial batch size $B_S = 1$, $B_S = 10$. 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

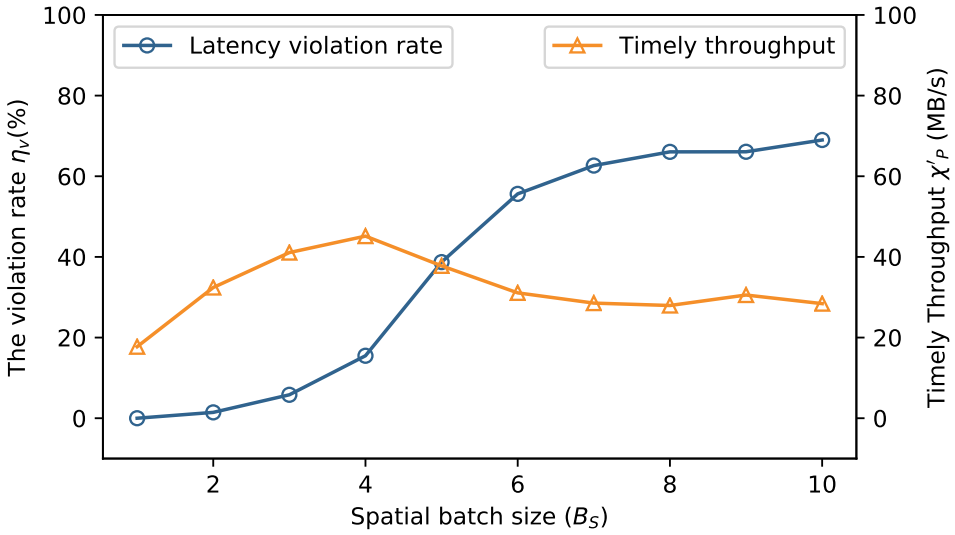

Figure 6.15: The impact of B_S on η_v and χ'_P

Table 6.2: 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

we use. Docker containers offer lower latency costs and lower variability than hardware virtualization [8]. 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 6.2, 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 optimisation 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.

6.4 Summary

In this Chapter we discuss the drawbacks of current measurement methods of Kafka's performance. We introduce the importance and difficulties in the fine tuning of Kafka's batch size for real-time requirements. 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 most messages within the user-defined latency constraint. The Kafka users may refer to the workflows in this chapter to obtain the corresponding features in their real-time systems, thus they optimise their batching strategy in their operation environment.

Chapter 7

Conclusions and Outlook

7.1 Conclusions

In this thesis, we study the real-time streaming methodologies and strategies based on the distributed publish/subscribe messaging pattern. The main goal is to guarantee the real-time requirements of stream processing by properly tuning of Apache Kafka, a popular distributed messaging system. Kafka is capable of flexible configuration parameters, thus we are able to implement different messaging patterns for various application scenarios. We propose a queueing based model to predict the performance of Kafka, which provides the users a simple way to study the effects of important configuration parameters. From the experimental results we observe that batching has a significant effect on producer throughput and relative payload. This helps the Kafka users with the decision on the batch size and partition number configuration. The disk occupancy over time can be predicted through the model and this reminds the users of proper adjustment of partition numbers.

We learn how to reliably deliver messages using Kafka using a Docker-based Kafka testbed, which is built for exploring the effects of failure scenarios including unstable network conditions. In the testbed we can easily and quickly deploy a Kafka cluster, and restore different failure scenarios on it. This flexible testbed is available on GitHub: https://github.com/woohan/kafka_start_up.git.

Based on the analysis of possible message states in Kafka, we introduce two metrics, the probability of message loss and the probability of message duplicate for the reliability evaluation. A machine learning model is proposed to predict those metrics, while the application scenarios are known. We run experiments on the 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 (e.g. batch size) dynamically is the ideal solution for guaranteeing reliability in complex and changing scenarios.

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 performance measurement, then use experimental data to illustrate their unreliability. We find that given a certain latency constraint, measuring the mean value of end-to-end latency is insufficient when batching method is applied. We analyse 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. 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.

7.2 Outlook

Our research can be extended in several directions, which call for further research. The main ideas are summarized as follows:

- More failure scenarios including the failure of brokers should be studied to complete the prediction model.
- The Docker-based Kafka testbed can be improved by integrating more functionalities. E.g. fault injection module of a broker failure.
- We only consider packet loss rate and network delay to represent the network status. More metrics, including jitter can be introduced to the model.
- We do not make a deep dive into the retry strategy in the exactly-once delivery semantics, 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.
- The cost of end-to-end latency distribution analysis can be expensive in real-time decision

making. Thus the overhead of the prediction model formulation process can be reduced by using moment matching to fit a distribution.

- More variable real-world workloads with multiple client scenarios will be evaluated with our batching strategy.

Bibliography

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

[2] H. H. Aghdam and E. J. Heravi. Guide to convolutional neural networks. *New York, NY: Springer*, 10:978–973, 2017.

[3] A. Ahmad, M. Hassan, M. Abdullah, H. Rahman, F. Hussin, H. Abdullah, and R. Saidur. A review on applications of ann and svm for building electrical energy consumption forecasting. *Renewable and Sustainable Energy Reviews*, 33:102–109, 2014.

[4] 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*, 6(11):1033–1044, 2013.

[5] K. Al-Aubidy, A. Derbas, and A. Al-Mutairi. Real-time healthcare monitoring system using wireless sensor network. *International Journal of Digital Signals and Smart Systems*, 1(1):26–42, 2017.

[6] B. K. Al-Shammari, N. Al-Aboody, and H. S. Al-Raweshidy. Iot traffic management and integration in the qos supported network. *IEEE Internet of Things Journal*, 5(1):352–370, 2017.

[7] AmazonWebServices. Amazon managed streaming for kafka (msk). https://aws.amazon.com/msk/, 2021. [Online].

[8] 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)*, pages 1–7. IEEE, 2016.

[9] ApacheFlink. Apache flink - stateful computations over data streams. https://flink.apache.org/, 2021. [Online].

[10] ApacheHadoop. Apache hadoop. https://hadoop.apache.org/, 2021. [Online].

[11] ApacheKafka. Powered by - apache kafka - the apache software foundation. https:

//kafka.apache.org/powered-by, 2021. [Online].

[12] ApacheSamza. Apache samza - a distributed stream processing framework. http://samza.apache.org/, 2021. [Online].

[13] ApacheSpark. Apache spark: Lightning-fast unified analytics engine. https://spark.apache.org/, 2021. [Online].

[14] ApacheStorm. Apache storm - a free and open source distributed realtime computation system. https://storm.apache.org/, 2021. [Online].

[15] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613, 2018.

[16] L. Arras, G. Montavon, K.-R. Müller, and W. Samek. Explaining recurrent neural network predictions in sentiment analysis. *arXiv preprint arXiv:1706.07206*, 2017.

[17] A. Bahrammirzaee. A comparative survey of artificial intelligence applications in finance: artificial neural networks, expert system and hybrid intelligent systems. *Neural Computing and Applications*, 19(8):1165–1195, 2010.

[18] I. A. Basheer and M. Hajmeer. Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, 43(1):3–31, 2000.

[19] J. K. Basu, D. Bhattacharyya, and T.-h. Kim. Use of artificial neural network in pattern recognition. *International journal of software engineering and its applications*, 4(2), 2010.

[20] 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)*, pages 2049–2054. IEEE, 2015.

[21] C. Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[22] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[23] E. Brewer. A certain freedom: thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 335–335, 2010.

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

[25] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit

for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[26] 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*, 10(12):1718–1729, 2017.

[27] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736, 2013.

[28] D. Chen. Research on traffic flow prediction in the big data environment based on the improved rbf neural network. *IEEE Transactions on Industrial Informatics*, 13(4):2000–2008, 2017.

[29] 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*, pages 1087–1098. ACM, 2016.

[30] E. Choi, M. T. Bahadori, A. Schuetz, W. F. Stewart, and J. Sun. Doctor ai: Predicting clinical events via recurrent neural networks. In *Machine Learning for Healthcare Conference*, pages 301–318, 2016.

[31] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

[32] Confluent. Confluent cloud: Apache kafka re-engineered for the cloud. https://www.confluent.io/confluent-cloud/, 2021. [Online].

[33] R. Coppola and M. Morisio. Connected car: technologies, issues, future trends. *ACM Computing Surveys (CSUR)*, 49(3):1–36, 2016.

[34] H.-N. Dai, R. C.-W. Wong, H. Wang, Z. Zheng, and A. V. Vasilakos. Big data analytics for large-scale wireless networks: Challenges and opportunities. *ACM Computing Surveys (CSUR)*, 52(5):1–36, 2019.

[35] 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*, pages 1–13. ACM, 2014.

[36] M. D. de Assuncao, A. da Silva Veith, and R. Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018.

[37] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[38] 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*, pages 145–156. ACM, 2015.

[39] 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*, pages 227–238. ACM, 2017.

[40] C. Doulkeridis and K. NØrvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, 2014.

[41] M. Ge, H. Bangui, and B. Buhnova. Big data for internet of things: A survey. *Future generation computer systems*, 87:601–614, 2018.

[42] A. Goel, S. Aghajanzadeh, C. Tung, S.-H. Chen, G. K. Thiruvathukal, and Y.-H. Lu. Modular neural networks for low-power image classification on embedded devices. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(1):1–35, 2020.

[43] 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.*, 35(2):33–45, 2012.

[44] 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*, 28(7):1891–1904, 2017.

[45] N. Gruber and A. Jockisch. Are gru cells more specific and lstm cells more sensitive in motive classification of text. *J. Front. Artif. Intell*, 3:1–6, 2020.

[46] S. Hagihara, Y. Fushihara, M. Shimakawa, M. Tomoishi, and N. Yonezaki. Web server access trend analysis based on the poisson distribution. In *Proceedings of the 6th International Conference on Software and Computer Applications*, pages 256–261. ACM, 2017.

[47] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz. Meeting iot platform requirements with open pub/sub solutions. *Annals of Telecommunications*, 72(1-2):41–52, 2017.

[48] 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*, pages 124–133, 2017.

[49] 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*, pages 361–373. ACM, 2016.

[50] 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*, pages 13–22, 2014.

[51] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, I. Simon, C. Hawthorne, N. Shazeer, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck. Music transformer: Generating music with long-term structure. In *International Conference on Learning Representations*, 2018.

[52] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

[53] IBM. Ibm cloud: Event streams. https://console.bluemix.net/catalog/services/event-streams, 2021. [Online].

[54] C. V. N. Index. Forecast and methodology, 2016–2021. *White Paper, June*, 2017.

[55] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019.

[56] S. Jaeger, S. Manke, J. Reichert, and A. Waibel. Online handwriting recognition: the npen++ recognizer. *International Journal on Document Analysis and Recognition*, 3(3):169–180, 2001.

[57] D. Jaramillo, D. V. Nguyen, and R. Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5. IEEE, 2016.

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

[59] 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)*, pages 1–6. IEEE, 2011.

[60] D. T. Kanapram, F. Patrone, P. Marin-Plaza, M. Marchese, E. L. Bodanese, L. Marcenaro, D. M. Gómez, and C. Regazzoni. Collective awareness for abnormality detection in connected autonomous vehicles. *IEEE Internet of Things Journal*, 7(5):3774–3789, 2020.

[61] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE, 2018.

[62] G. Kaur and K. Bahl. Software reliability, metrics, reliability improvement using agile process. *International Journal of Innovative Science, Engineering & Technology*, 1(3):143–147, 2014.

[63] H. Khazaei, J. Misic, and V. B. Misic. Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems. *IEEE Transactions on parallel and distributed systems*, 23(5):936–943, 2011.

[64] I. K. Kim, W. Wang, and M. Humphrey. Pics: A public iaas cloud simulator. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 211–220. IEEE, 2015.

[65] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design'96*, pages 151–170. Springer, 1996.

[66] L. Kozma-Spytek, P. Tucker, and C. Vogler. Voice telephony for individuals with hearing loss: The effects of audio bandwidth, bit rate and packet loss. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*, pages 3–15, 2019.

[67] J. Kreps. Benchmarking apache kafka: 2 million writes per second (on three cheap machines). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines, 2021. [Online].

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

[69] 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*, pages 239–250. ACM, 2015.

[70] M. Lam. Neural network techniques for financial performance prediction: integrating fundamental and technical analysis. *Decision support systems*, 37(4):567–581, 2004.

[71] A. ledenev. Pumba-chaos testing and network emulation tool for docker. https://github.com/alexei-led/pumba, 2021. [Online].

[72] I. Lee and K. Lee. The internet of things (iot): Applications, investments, and challenges for enterprises. *Business Horizons*, 58(4):431–440, 2015.

[73] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010.

[74] B. Li, Y. Diao, and P. Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

[75] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees.

In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 399–410. IEEE, 2015.

[76] B. Lohrmann, D. Warneke, and O. Kao. Nephele streaming: stream processing under qos constraints at scale. *Cluster computing*, 17(1):61–78, 2014.

[77] S. Lokesh, P. M. Kumar, M. R. Devi, P. Parthasarathy, and C. Gokulnath. An automatic tamil speech recognition system by using bidirectional recurrent neural network with self-organizing map. *Neural Computing and Applications*, 31(5):1521–1531, 2019.

[78] A. P. Marugán, F. P. G. Márquez, J. M. P. Perez, and D. Ruiz-Hernández. A survey of artificial neural network in wind energy systems. *Applied energy*, 228:1822–1836, 2018.

[79] H. Mayer, F. Gomez, D. Wierstra, I. Nagy, A. Knoll, and J. Schmidhuber. A system for robotic heart surgery that learns to tie knots using recurrent neural networks. *Advanced Robotics*, 22(13-14):1521–1537, 2008.

[80] MicrosoftAzure. Hdinsight: Easy, cost-effective, enterprise-grade service for open source analytics. https://azure.microsoft.com/en-us/services/hdinsight/, 2021. [Online].

[81] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018.

[82] M. F. Neuts. *Matrix-geometric solutions in stochastic models: an algorithmic approach*. Courier Corporation, 1994.

[83] D. Niu, Y. Liang, and W.-C. Hong. Wind speed forecasting based on emd and grnn optimized by foa. *Energies*, 10(12):2001, 2017.

[84] 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*, 10(12):1634–1645, 2017.

[85] A. F. Novelo, E. Q. Cucarella, E. G. Moreno, and F. M. Anglada. Fault diagnosis of electric transmission lines using modular neural networks. *IEEE Latin America Transactions*, 14(8):3663–3668, 2016.

[86] J. Qi, G. Jiang, G. Li, Y. Sun, and B. Tao. Surface emg hand gesture recognition system based on pca and grnn. *Neural Computing and Applications*, 32(10):6343–6351, 2020.

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

[88] P. Romano and M. Leonetti. Self-tuning batching in total order broadcast protocols via analyt-

ical modelling and reinforcement learning. In *2012 International Conference on Computing, Networking and Communications (ICNC)*, pages 786–792. IEEE, 2012.

[89] D. Saad. Online algorithms and stochastic approximations. *Online Learning*, 5:6–3, 1998.

[90] I. Sadeghkhani, A. Ketabi, and R. Feuillet. Radial basis function neural network application to power system restoration studies. *Computational Intelligence and Neuroscience*, 2012, 2012.

[91] R. Sahal, J. G. Breslin, and M. I. Ali. Big data and stream processing platforms for industry 4.0 requirements mapping for a predictive maintenance use case. *Journal of Manufacturing Systems*, 54:138–151, 2020.

[92] S. Sakr, A. Liu, and A. G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Computing Surveys (CSUR)*, 46(1):1–44, 2013.

[93] K. Salah and R. Boutaba. Estimating service response time for elastic cloud applications. In *Cloud Networking (CLOUDNET), 2012 IEEE 1st International Conference on*, pages 12–16. IEEE, 2012.

[94] K. Salah and T. R. Sheltami. Performance modeling of cloud apps using message queueing as a service (maas). In *Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on*, pages 65–71. IEEE, 2017.

[95] O. B. Sezer, E. Dogdu, and A. M. Ozbayoglu. Context-aware computing, learning, and big data in internet of things: a survey. *IEEE Internet of Things Journal*, 5(1):1–27, 2017.

[96] 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*, pages 139–142, 2017.

[97] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[98] M. Sundermeyer, T. Alkhouli, J. Wuebker, and H. Ney. Translation modeling with bidirectional recurrent neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 14–25, 2014.

[99] E. Thereska and G. R. Ganger. Ironmodel: Robust performance models in the wild. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):253–264, 2008.

[100] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[101] Q.-C. To, J. Soto, and V. Markl. A survey of state management in big data processing systems.

The VLDB Journal, 27(6):847–872, 2018.

[102] M. Vejdannik, A. Sadr, et al. Modular neural networks for quality of transmission prediction in low-margin optical networks. *Journal of Intelligent Manufacturing*, pages 1–15, 2020.

[103] 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)*, pages 363–378, 2016.

[104] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[105] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein. Building a replicated logging system with apache kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, 2015.

[106] K. Wang and M. M. H. Khan. Performance prediction for apache spark platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 166–173. IEEE, 2015.

[107] B. Widrow, A. Greenblatt, Y. Kim, and D. Park. The no-prop algorithm: A new learning algorithm for multilayer neural networks. *Neural Networks*, 37:182–188, 2013.

[108] C. Wohler and J. K. Anlauf. An adaptable time-delay neural-network algorithm for image sequence analysis. *IEEE Transactions on Neural Networks*, 10(6):1531–1536, 1999.

[109] 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)*, pages 154–161. IEEE, 2019.

[110] H. Wu, Z. Shang, and K. Wolter. Trak: A testing tool for studying the reliability of data delivery in apache kafka. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 394–397. IEEE, 2019.

[111] H. Wu, Z. Shang, and K. Wolter. Learning to reliably deliver streaming data with apache kafka. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 564–571. IEEE, 2020.

[112] H. Wu, Z. Shang, and K. Wolter. A reactive batching strategy of apache kafka for reliable stream processing in real-time. In *2020 IEEE International Symposium on Software Reliabil-*

ity Engineering (ISSRE). IEEE, 2020.

[113] Y. Wu, G. Min, and L. T. Yang. Performance analysis of hybrid wireless networks under bursty and correlated traffic. *IEEE Transactions on Vehicular Technology*, 62(1):449–454, 2012.

[114] Y. Yin, W. Zhang, Y. Xu, H. Zhang, Z. Mai, and L. Yu. Qos prediction for mobile edge service recommendation with auto-encoder. *IEEE Access*, 7:62312–62324, 2019.

[115] Q. Zhang, L. T. Yang, Z. Chen, and P. Li. A survey on deep learning for big data. *Information Fusion*, 42:146–157, 2018.

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

[117] Y. Zhang, L. Wu, N. Neggaz, S. Wang, and G. Wei. Remote-sensing image classification based on an improved probabilistic neural network. *Sensors*, 9(9):7516–7539, 2009.

[118] G. Zhao, C. Zhang, and L. Zheng. Intrusion detection using deep belief network and probabilistic neural network. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 1, pages 639–642. IEEE, 2017.

[119] 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*, 20(1):383–398, 2018.

List of Figures

List of Tables

About the Author

For reasons of data protection, the curriculum vitae is not published in the electronic version.