Standardizing Data For Kafka Consumers

Wesley Jones *Iowa State University*
CPRE 550: Distributed Systems and Middleware
Ames, US
wesleyj@iastate.edu

***Abstract*—Scaling publish and subscribe systems face a variety of issues. When these systems are integrated into a tiered data pipeline data reliability can become one of the culprits. Different systems that rely on data from a pipeline will either have to rely on a local parser to assert or correct the data field types for a contained record or rely on the client to correctly format a record before sending it. The alternative to these two options is a separate service to define data types and reform data into a standardized format. These service create complexity and add latency to a streaming pipeline.**

I propose a more focused approach on transforming data as it enters a pipeline. Using a data serialization service, such as Apache Avro *before* arriving into a data pipeline, such as Apache Kafka, means that withdrawing centralized data will be more fluid across applications.

I achieved my objectives with the system design and software I built, but the performance issues are likely insurmountable. The average performance of the adjustment process is around 25% of the total incoming messages. The most costly step is the parsing the log data. There is plenty of potential for improvement, but likely these will rely on constructing a method that is outside of my objectives.

***Index Terms*—Apache Kafka, Apache Avro, logs, log parsing, data pipeline, data serialization, data processing**

I. Introduction

FOR SYSTEMS THAT GENERATE high volume sequential messages, such as logging, efficient storage and searching infrastructure is needed. Essential components of this infrastructure rely on publish and subscribe systems. These systems handle unbounded data input and typically provide records for a separate indexing service. This indexing service offers the niceties required for handling large data: searching, analysis, and data mining, among other varieties of application focused utilities.

Scaling publish and subscribe systems face a variety of issues. When these systems are integrated into a tiered data pipeline data reliability can become one of the culprits. Different systems that rely on data from a pipeline will either have to rely on a local parser to assert or correct the data field types for a contained record or rely on the client to correctly format a record before sending it. The alternative to these two options is a separate service to define data types and reform data into a standardized format. These service create complexity and add latency to a streaming pipeline.

I propose a more focused approach on transforming data as it enters a pipeline. Using a data serialization service, such as Apache Avro *before* arriving into a data pipeline, such as Apache Kafka, means that withdrawing centralized data will be more fluid across applications. This methodology is geared to increase data extraction simplicity, not to improve computation time per record (although that could be a natural benefit). A typical pipeline for Apache Kafka without data serialization built in can be described as this:

1) Producer creates a record
2) Kafka receives the data and stores it within a topic
3) A consumer receives the message from a topic
4) An application parses message for effective usage
    a) This repeats for *each* application

The steps of 4 and 4.a can create additional overhead for consuming data. Even in instances where infrastructure includes a record parsing application at step 5 — one that can be extended to additional applications in an environment — this invalidates the functionality offered by a data pipeline. To date, there are various custom codebases on GitHub that work to achieve this type of functionality, but they are tied to client-based services and offer specific domain solutions to the authors.

Kafka brings a lot to the table when it comes to designing a distributed system. Of the core components that distributed systems are expected to offer, Kafka provides all of them at a basic configuration [1], [2] (with caveats [3]. While it checks all of the boxes of transparency, reliability, interoperability, scalability, dependability, and security, it has no way of enforcing that to the systems implemented on top of it.

A. Objectives

1) Design a system that runs on top of Kafka and ensures data offered to consumers is standardized based on the content of data ingested.
2) Analyze each message so that all data within a defined Kafka topic can be consumed with the same technique.
3) Construct this to happen within the Kafka cluster so that no external services (aside from producers and consumers) are required.

In this model the parsing will happen within the Kafka cluster instead of in an external environment and the standardization templates are both automated and self-contained[1].

I evaluate processes for pre-pending Apache Kafka[2] with a data schema or data serialization process that can ensure extracted records from Kafka are fully useful at the time of extraction. This process involves interpreting incoming data and programmatically defining a data schema. I utilize

[1]See the code repository here: https://github.com/iamwpj/kafka-types
[2]https://kafka.apache.org/

Apache Avro[3] for schema templating. Once this is implemented, records consumed from Kafka should be filed into a standardized schema.

II. Background

Apache Kafka is a message queue that supports asynchronous, buffered inputs (called producers) and reliable real-time output by subscribers (called consumers). The design of Kafka was based around short term log management — a shared system that would allow for offline processing by some consumers and rapid real-time responses by other consumers [4]. The service is built on Java and utilizes a series of abstraction layers to provide transparency and durability of the incoming messages.

There are two main roles that a node in the cluster can operate as: *broker* and *controller*. These roles are assigned in the configuration. During cluster startup any nodes eligible for controller will participate in a vote. There have been multiple iterations of Kafka controller election formats. Apache ZooKeeper was required as a standalone service alongside Kafka until newer (post-2020) releases. This system operates as a general purpose coordination layer in distributed systems [5]. Because of some of the guarantees that ZooKeeper provides it is still popular in highly performant and distributed Kafka set ups. Newer releases of Kafka implement a bespoke coordination service: KRaft, based on the Raft consensus algorithm [6], [7]. The consensus system maintains metadata is appropriately positioned to host cluster controller elections. KRaft deviates from Raft by relying on a pull-based system of voting to establish consensus, but shares a similar group-based approach to maintaining metadata confidence.

Brokers in a Kafka cluster handle data. They support the necessary bucketing required by the data ingesting processes and server to abstract the data storage process away from the file system and network interface simultaneously. The broker system offers a transparent method of subdividing larger sets of data (more than would fit on a single cluster node) across multiple nodes. It is through the subdividing that the distribution can achieve redundancy. Each broker coordinates which *topics* and *partitions* it holds. This coordination (performed through metadata consensus) ensures that required *replicas* of the data are stored on different brokers [4]

Brokers are the underlying data storage denotation. They are independent of the logical categorization that is applied to the overall storage system — that of topics. When data is produced and written to the cluster it is inserted into a topic. The topic holds security, organization, and sometimes data definition policies. When data is read from the cluster, it is consumed out of a topic [2].

Apache Avro is a utility that runs will conform data to a specified standard (defined in Avro IDL). The data is serialized using a JSON defined notation, and the underlying system processor can be built into a variety of applications as needed. A common technique to implement Avro into a Kafka cluster is to have a intermediary service running that provides pre-built templates to Kafka producers to perform encoding. Once the brokers make the message available to the consumers they will require the schema to decode the message. This process results in some "back and forth" communication that, aside from adding latency to real-time streaming, can increase a systems' complexity, thus reducing reliability and potentially recover-ability.

[3]https://avro.apache.org/

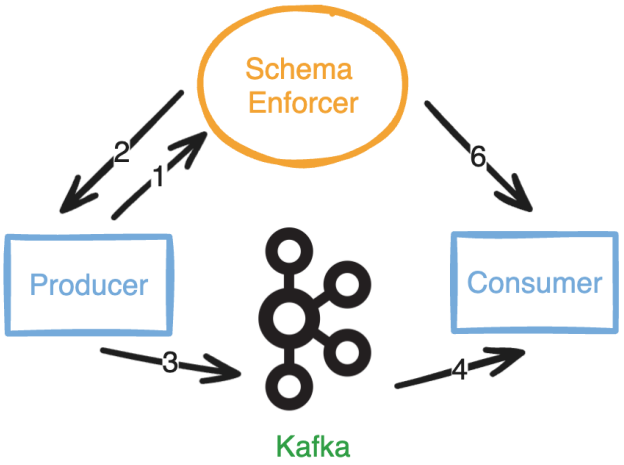

Fig. 1: A common setup for a data pipeline. This involves six steps with the assumption being that the Avro templates lay outside of the Kafka system.

III. Related Works

Kreps et al. introduced Kafka in 2011 as a dedicated distributed messaging application for log handling in 2011 [4]. They specifically designed the service to work with Apache Hadoop. Their efforts focused on provide a reliable distributed system with transparent data storage. The software is augmented by an API that offers easy extensibility. The authors provide benchmarking and break down the technology that allows Kafka to maintain performance in high throughput scenarios.

Zhang et al. confirmed the performance capabilities of Kafka and offered insight into it's viability based on extensibility and community support [8]. For the case here, Zhang et al. provide a standard expectation of performance for a Kafka pipeline. I found that Kul and Sayar provided useful discussion on applications of Kafka. They describe the importance of pub/sub systems in microservices, an efficiency focused field [2]. It is likely that extending the data pipeline process with better serialization will help to reduce latency in microservices.

There is some important discussion in this space when it comes to handling streaming inconsistencies. The common methods of addressing collisions in messaging applications can also provide a reference for other in-stream adjustments. Sax et al. offered a method for addressing inconsistencies by creating a dual stream approach: one to handle physical ordering of messages, another to handle logical ordering [9]. Their methods offer integrated features such as `agg` or `flatmap` which allow transformations during the reconciliation process. Their approach does not apply directly to my goals since their primary focus is to address variances in timestamp and ordering data.

Offering some explanation for "missing" features for Kafka as a reliable commit log service, Rooney et la. describe Kafka processes in depth [3]. The researchers describe the processes of Kafka in regards to a distributed replication system. Their discussion shows that Kafka is a piece the data storage and pipeline system — not to be mistaken for something more, specifically calling out its shortcomings with transactional consistency, coherence in the contents of a Kafka topic, and compliance for tracked metadata.

Narrowing my focus to data serialization I found that Viotti et al. provide an evaluation of available serialization applications [10]. This survey includes notes on both Protocol Buffers and Apache Avro. When it comes to processes of serialization within a distributed system I found that Munonye et al. offered a walk through of various implementation techniques for serialization systems [11]. For my research, I drew from their concept of buffered serialization — where the serialization process is a component of the distributed system (including the schema).

Finally, since the implemented testing of this proposed system design is focused on logs — primarily system logs, there is research on performance and reliability capabilities of various log parsing service or techniques. Zhu et al. present an assessment of 13 tools with efficiency and accuracy rankings [12]. Their analysis would also point to research resulting in the Drain method which utilizes fixed depth tree-based to progressively match components of log messages until a reliable result is found [13]. In certain implementations this method would get faster over running time.

IV. Methodology

My goal is not to benchmark the performance capabilities of a framework data pipeline — for this reason I am relying on an "ad-hoc" setup. I have implemented a cluster within a set of small containers. I can create a baseline latency check and measure against that to ensure additional processing does not cause delays down the chain. Should my desired architecture come to fruition more intensive capacity, throughput, and deeper latency testing should be done on "industry" hardware.

Kafka offers many performance advantages, but it also offers a simple way to implement a distribution of data to many disparate systems. In scenarios where it merely extends a traditional database or other data storage distribution the performance advantages are secondary to the reliable distribution infrastructure — this is the area where my evaluation lies.

Modern systems provide nearly complete coverage of "text-based" management through tooling such as YAML configurations, continuous integration/continuous deployment, and hosted code. You can find much of the detail and process within my GitHub repository.[4]

A. System Design

Kafka offers natural horizontal scaling and through the use of containers (and subsequently container orchestrates) systems engineers can take advantage of large, vertically scaled servers. The efficiency of a cluster can vary by the size of messages and quantity per buffer [14]. The container-based system I have designed encompasses the following:

1) Container Host: I have a RedHat 8.9 server with the RedHat provided package Podman[5] installed. This operates using the Open Container Initiative[6] standard — ensuring that it is fully compatible with other container solutions, such as Docker. In my notes and configuration setup I often refer to Docker, this is partly an oversight and partly because on the server the commands for `podman` and `docker` are synonymous.

2) Configuration: The container build process is controlled via Podman Compose.[7] This means that the entire container infrastructure can defined in a YAML file for an automated launch (which I have implemented). The containers enumerated are:

- Three Kafka nodes, broker and controller eligible. Each is with a unique address for accessing within the cluster and cluster support systems, as well as an port exposed to the container host for access from outside the container defined environment. The containers are pre-built by Bitnami, a popular packaging service for open source projects.[8]
- An instance of Kafka-UI [9], an open source web interface for viewing cluster interactions. This interface allows for some minor cluster management as well.
- Three containers to facilitate metrics collection. These are:
  - kafka-exporter[10], a utility to collect Java, system, and Kafka statistics and expose them to a metrics aggregator. This container is prebuilt by Bitnami as well.
  - Prometheus[11], a popular time-series databases (metrics aggregator). This container is built by Prometheus.
  - Grafana[12], a popular graphing web application. This services works well with the time-series database to provide views of the collected metrics data. This container is built by Grafana.

These containers will allow for both the efficient ingestion of data, but enough performance analysis for ensuring that data serialization will not affect processing.

B. Data Serialization

Typically schema conformation occurs outside a linear pipeline. This can be helpful in situations where large data messages could cause significant stream delays. In these cases, a serialization process would have to remain external to the pipeline stream until processing latency can be reduced

[4]https://github.com/iamwpj/kafka-types

[5]https://podman.io/
[6]https://opencontainers.org/
[7]https://docs.podman.io/en/latest/markdown/podman-compose.1.html
[8]https://bitnami.com/
[9]https://github.com/provectus/kafka-ui
[10]https://bitnami.com/stack/kafka-exporter/containers
[11]https://prometheus.io/
[12]https://grafana.com/

to the point that incoming messages are faced with delays due to processing limitations. Common pipeline patterns set ups involve several steps — often "doubling back" along a trajectory, see 1.

Data serialization is performed by Apache Avro running within a Python environment. This provides a low barrier of entry for modelling the setup and offers enough performance. On a test run during proof of concept the processing for of a schema with two small samples relied primarily on CPU resources (92%), this aligns well with vertical scaling of Kafka clusters where the constraining scaling resource is typical input/output (often, disk).

Listing 1: Log from a fake log generator. This is part of Phase 1.

```
<15>April 12 19:32:44 powlowski2164 quas[5150]: The
    PNG pixel is down, synthesize the back-end
    bandwidth so we can transmit the HTTP firewall!
```

Listing 2: Log file after applying grok pattern. Notice the string formatting for the "pid" field.

```
{
  "timestamp": "April 12 19:32:44",
  "timestamp8601": null,
  "facility": null,
  "priority": null,
  "logsource": "powlowski2164",
  "program": "quas",
  "pid": "5150",
  "message": "The PNG pixel is down, synthesize the
      back-end bandwidth so we can transmit the HTTP
       firewall!"
}
```

Listing 3: Data deserialized from the Avro bytecode. Notice the correctly formatted integer element for "pid".

```
{
  "timestamp": "April 12 19:32:44",
  "logsource": "powlowski2164",
  "program": "quas",
  "pid": 5150,
  "message": "The PNG pixel is down, synthesize the
      back-end bandwidth so we can transmit the HTTP
       firewall!"
}
```

To properly serialize data I had to build a two step phases:

1) Data is pulled from a Kafka topic and fields and values are itemized. I chose to utilize the grok patterns method and enabled in Python with a supporting library[13]. This is essentially a sequence of regex patterns that help to define field values into a JSON format. Adding this layer can cause significant latency increases if the amount of adjustments grows. For my part I limited the processing three patterns for common Linux syslog outputs. For production approaches a designed list of formatters limited per Kafka topic would provide low-latency feed back from this first stage. You can see the raw log in Listing 1 and then the follow up result of grokking in Listing 2.
2) Once there is a standard set of fields my script will handle the extra fields, standardize the data types, and serialize the schema. The results are shown in Listing 7. This layer will provide any filtering, relabelling, or other small manipulations. This content is provided as the message field to a new Kafka topic where it can be consumed as a standardized format. I have provided a deserialization example relying on the auto-defined schema in Listing 3.

[13]https://github.com/garyelephant/pygrok

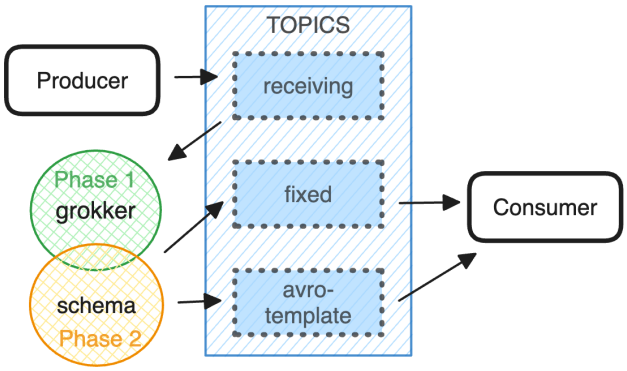

Fig. 2: System design for this research.

Additional details on Phase 1 and Phase 2 are provided in the next sections, respectively.

C. Phase 1: Identifying fields with grok

An issue with handling various data inputs is identifying fields and values. Disparate consuming systems attached to a shared pipeline require a transparent should be able to consume from a topic with complete confidence in the offered schema. Since I am controlling the input format to represent a design system, I can utilize a few grok statements with just a small coverage of common system logs (RFC 3164) in this case. For this pattern I utilized pygrok[14] with a pre-configured pattern covering the standard format.

My tool begins by running a consumer on an ingest topic, `receiving`. The receiving topic can contain any variety of data so long as the correct grok patterns for it are defined. An example of these builtin patters, plus a catch all, is shown in Listing 4. The Python library will return log files organized by the defined fields in the RFC specification [15]. The grok patterns in this case a combined list of regex patterns that combine to fulfill the standard.

Listing 4: Log grok patterns

```
# RFC 3164
"%{SYSLOGLINE}"
# RFC 5424
"%{SYSLOG5424LINE}",
# Catch All
".*%{TIMESTAMP_ISO8601:timestamp}.%{SPACE}%{
    GREEDYDATA:message}",
```

To demonstrate a system where there are a variety of mismatching messages in a receiving topic, I apply each listed grok pattern from Listing 4 to the message. I then count the

[14]https://github.com/garyelephant/pygrok
[15]https://www.rfc-editor.org/rfc/rfc3164.html

fields returned by applied pattern and select for the highest match, thus ensuring the most verbose log parsing is selected to create a schema from. This code is shown in Listing 5.

Listing 5: Method to iterate grok patterns and match data. There are three iterations here, the first is for the grok patterns, the next two are in the process of calculating the maximum and finding it within lists.

```
for pattern in self.all_patterns:
    grok = Grok(pattern=pattern)
    result[pattern] = grok.match(data)

# Return the "best" result.
# This is just the pattern with
# the most field matches.

counts = [
    len(result[i].keys())
    if result[i] else 0
    for i in result
]
idx, _ = max(
    enumerate(counts),
    key=lambda x: x[1]
)
return result[self.all_patterns[idx]]
```

D. Phase 2: Applying changes to a topic

Once the log is parsed, Phase 2 will handle generating a dynamic Avro schema. This involves iterating the newly grokked fields and defining them as a supported Avro data type. There are a variety of options here, but each step will cause CPU time and log flow latency. I kept my example simple and iterated based on Python's `isdigit()` function (see this in Listing 6). If that field is found to match — the data is re-encoded and a schema template is built. This process is streamlined by removing "null" fields such as those seen between Listing 2 to 3.

Listing 6: The process of identifying and relabeling fields in preparation for the Avro schema application. If this is not done then the data will not fit into the Avro schema by it's designed settings.

```
for i in data:
    field_name = i
    if data[i].isdigit():
        field_type = ["int"]
        data[i] = int(data[i])
    else:
        field_type = ["string"]
```

To speed up this process I only generated a new schema when there was a variance in the field data. Before the template would be constructed an identifying hash is constructed from the JSON key values of the grokked message. The hash is used to identify the schema template, and in the case where it already exists, the tool can proceed to submitting to a final topic. Generated schema are saved to a separate topic in Kafka and the log messages are identified with the Avro template name to ensure remote consumers can always identify the proper schema.

Saving a to a separate topic is borrowed from the discussion by [11]. By creating a more transparent pipe, I need to ensure that consumers are reliably able to perform deserialization of provided data.

Metric	*Value*
Kafka Ingest Rate	8,000 msg/s per processor
Delay on Kafka Broker Commitment	*instantaneous**
Adjustment Processing Rate	1,750 msg/s per processor

TABLE I: Averages for various performance metrics.
**The instantaneous result for broker commitment has to do with test throughput. Because of how Kafka commits messages to topics (batches) [4], [15] my testing never resulted in a delay that would have been caused by this process.*

Matching grok pattern	1.97×10^{-3} seconds (1970μ)
Schema application	3.57×10^{-5} seconds (35μ)
Producer	2.45×10^{-5} seconds (24μ)

TABLE II: Performance metrics for each stage from 50,000 message iterations. Notice the significantly poorer performance for the grok stage.

Listing 7: The encoded data from the Avro schema.

```
b'\x00"April 12 19:32:44\x00\x1apowlowski2164\
    x00\x08quas\x00\xbcP\x00\xbc\x01The PNG
    pixel is down, synthesize the back-end
    bandwidth so we can transmit the HTTP
    firewall!'
```

Once the schema is set, the data message is encoded (see Listing 7 and the message is submitted to the `fixed` topic. This is where consumers will access assimilated data.

V. Performance

Performance is sub-optimal. The container cluster of Kafka brokers (3 instances) appeared to be performant up to the maximum test amount of 20,000 messages per second. This performance was without any log processing set up and merely a verification of the throughput capabilities of the system. The designed system with two phases poses a significant performance cost and (more importantly) imposes a significant latency cost. This results in a maximum message rate of around 20% the total consumable messages. The breakdown of these results are in Table I.

Digging into the processing to see where the source of latency is occurring provides some positive news. The major source of latency occurs during the grokking phase — something that shouldn't come as a surprise given the "hammer-like" methodology of regex patter matching. The required cycle time for this step is about 100 times that of the other two.

Similar to the aforementioned producing and committing process — during testing of consuming the messages from the fixed topic, there was never any delay. Message consumption was able to happen at nearly real time, providing my tool was not running. During runtime the consuming capabilities from the fixed topic was the same as the Adjustment Processing Rate average.

VI. Discussion

A. Performance Improvements

While the results are disappointing, this run was a "first effort". The process converting logs inline with a standard

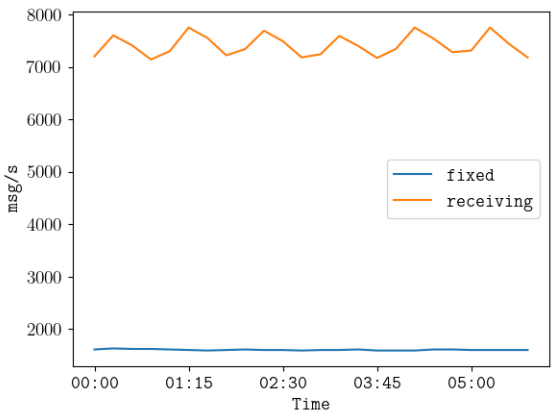

Fig. 3: A graph of the messages received by the Kafka brokers for each topic, fixed and receiving. The skew indicates the amount of deviation caused by the adjustment process.

Kafka pipeline in the "real world" is still a reasonable goal. Here are some straightforward improvements that would likely have significant effects:

1) Convert the data processing (if not all of the) code base to a more performant language. Currently the Python implementation faces limitations based on it's single process limitations and general performance degradation from runtime. The design of the code would allow for threading of each phase and ending in an asynchronous submission to the Kafka topic making threading especially easy. There is no issue with ordering since the messages either already contain a timestamp or the field could be enforced with the provided programming layer and the Avro schema.
2) Message parsing efficiency is an issue. There are two instances in the current process that could result in message parsing, the first is guaranteed: at the grok stage. The second is partial, in the enumeration of fields for the Avro schema auto-generation. In some cases, when this is the first time the schema is encountered all fields and values will be enumerated for generating a new schema. These two enumerations are costly at scale. It's possible to interlace them with a rework of phase 1, potentially saving cycles.
3) Grok parsing is likely a poor choice — no matter how it is implemented. I reviewed some research by Zhu et al. [12] and He et al. [13] that indicates the Drain, an online log sequencer (capable of parsing logs from a stream or pipeline), would be more reliable and likely more performant. Extending beyond Grok or a similar pattern matching technique has the downside of typically requiring a standalone service. This could be designed to allow for microservice-like processing, thus allowing the code base and system design to remain nearly identical to what is presented here, but the increased complexity is inescapable.

VII. Conclusion

Message processing can be an arduous task for CPUs, but also for humans involved with identifying and enumerating all the various forms. For a controlled system no additional help is needed: the application developer provides a standard message and the human designs the matching schema, Apache Kafka does the rest of the work. Too often this is no the case. The full potential of Kafka is in the convergence of a variety of data sources. Topics can partition the results, but at some point a system or human requires data be organized and structured for effective consumption. The ultimate goal is to reduce the amount of computer cycles collecting, parsing, and distributing this data requires.

I achieved my objectives with the system design and software I built, but the performance issues are likely insurmountable. The average performance of the adjustment process is around 25% of the total incoming messages. The most costly step is the parsing the log data. There is plenty of potential for improvement, but likely these will rely on constructing a method that is outside of my objectives.

A. Future Work

- More analysis is needed of effective log parsing during streaming to Kafka. Putting the parsing as soon as possible enables earlier schema applications — which provides more reliable and concise data for every system down the line. There is an advantage additionally in putting this parsing centralized, thus the log cluster.
- A generalized pre-processing framework for Apache Kafka would ensure effective transparency for application and system developers working with Kafka. This type of system would centralize the processing and reduce the effort on producers and consumers. It would also ensure more performance tuning capabilities for the cluster administrators.
- Efficient Apache Avro schema auto-generation tools can be created or improved. In some cases it would make sense to handle this type of design alongside log parsing techniques.

References

[1] M. v. Steen and A. S. Tanenbaum, *Distributed Systems*, fourth edition, version 4.01 (january 2023) ed. Erscheinungsort nicht ermittelbar: Maarten van Steen, 2023.

[2] S. Kul and A. Sayar, "A Survey of Publish/Subscribe Middleware Systems for Microservice Communication," in *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. Ankara, Turkey: IEEE, Oct. 2021, pp. 781–785. [Online]. Available: https://ieeexplore.ieee.org/document/9604746/

[3] S. Rooney, P. Urbanetz, C. Giblin, D. Bauer, F. Froese, L. Garces-Erice, and S. Tomic, "Kafka: the Database Inverted, but Not Garbled or Compromised," in *2019 IEEE International Conference on Big Data (Big Data)*. Los Angeles, CA, USA: IEEE, Dec. 2019, pp. 3874–3880. [Online]. Available: https://ieeexplore.ieee.org/document/9005583/

[4] J. Kreps, N. Narkhede, J. Rao, and others, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11. Athens, Greece, 2011, pp. 1–7, number: 2011.

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper}: Wait-free coordination for internet-scale systems," in *2010 USENIX annual technical conference (USENIX ATC 10)*, 2010.

[6] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[7] C. McCabe, "KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum - Apache Kafka - Apache Software Foundation," Jul. 2020. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum

[8] S. Zhang and R. Shen, "Subscription merging in filter-based publish/subscribe systems," Z. Zhu, Ed., Singapore, Singapore, Mar. 2013, p. 87687O. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.2010511

[9] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and Tables: Two Sides of the Same Coin," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. Rio de Janeiro Brazil: ACM, Aug. 2018, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3242153.3242155

[10] J. C. Viotti and M. Kinderkhedia, "A Survey of JSON-compatible Binary Serialization Specifications," 2022, publisher: arXiv Version Number: 2. [Online]. Available: https://arxiv.org/abs/2201.02089

[11] K. Munonye and P. Martinek, "Enhancing Performance of Distributed Transactions in Microservices via Buffered Serialization," *Journal of Web Engineering*, Oct. 2020. [Online]. Available: https://journals.riverpublishers.com/index.php/JWE/article/view/5725

[12] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and Benchmarks for Automated Log Parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Montreal, QC, Canada: IEEE, May 2019, pp. 121–130. [Online]. Available: https://ieeexplore.ieee.org/document/8804456/

[13] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *2017 IEEE International Conference on Web Services (ICWS)*. Honolulu, HI, USA: IEEE, Jun. 2017, pp. 33–40. [Online]. Available: http://ieeexplore.ieee.org/document/8029742/

[14] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang, "Data Infrastructure at LinkedIn," in *2012 IEEE 28th International Conference on Data Engineering*. Arlington, VA, USA: IEEE, Apr. 2012, pp. 1370–1381. [Online]. Available: http://ieeexplore.ieee.org/document/6228206/

[15] G. Fu, Y. Zhang, and G. Yu, "A Fair Comparison of Message Queuing Systems," *IEEE Access*, vol. 9, pp. 421–432, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9303425/