Realtime Visualization of Kafka Architecture

Matthew Jensen, Miro Manestar, Germán H. Alférez

***Abstract*—Apache Kafka specializes in the transfer of incredibly large amounts of data in real-time between devices. However, it can be difficult to comprehend the inner workings of Kafka. Often, to get real-time data, a user must run complicated commands from within the Kafka CLI. Currently, most tools used to visualize data streams over Kafka are commercially licensed and complicated to use. These tools also lack corresponding research to explain their implementations. Our contribution is a tool that monitors Kafka consumers, producers, and topics, and displays the flow of events between them in a web-based dashboard. This dashboard can be used to facilitate the comprehension of the complexity of a Kafka architecture, specifically for practitioners unfamiliar with the platform.**

***Index Terms*—Kafka Architecture, Data Visualization, Dashboard, KafkaJS**

I. Introduction

As the expectations placed upon a piece of data-heavy software increase, so does the scope of the tools used to support those expectations. To that end, numerous tools have been developed over the years with the aim of supporting complex data-driven tasks easy, especially in today's heavily cloud-dependent environment. Apache Kafka is one such tool, specializing in delivering high-performance data streams in a distributed environment [1]. However, as project requirements increase and the complexity of infrastructure grows to accommodate those requirements, the flow of data through Kafka becomes more convoluted. Subsequently, it can become increasingly difficult to understand its configuration and the path of the data as it flows throughout the application. In order to grasp the Kafka architecture, practitioners are often forced to run convoluted commands from within the Kafka CLI.

Thus, the contribution proposed here is a dashboard that can quickly be integrated into an existing Kafka architecture to gather data. This would enable an intuitive display of a Kafka architecture and the flow of data throughout it.

This paper is organized as follows: Section II presents the state of the art. Section III provides the theoretical framework upon which our tool is built. Section IV covers the methodology used to create this tool. Finally, Section V discusses the implications of our tool and outlines opportunities for further research.

II. State of the Art

Apache Kafka is widely used in applications requiring a messaging system with high throughput and scalability while maintaining low latency [2]. It is used in major companies like Adidas, The New York Times, and Pinterest to handle real-time predictions, monitoring, and content distribution [3].

The developers at SoftwareMill created a visualization that shows the designation between nodes, partitions, and replication factor [4]. This website operates off of meaningless data, and does not actually connect to a Kafka broker. However, this type of educational visualization accomplishes a similar goal to our tool. It displays a Kafka architecture and its data flows in an easy-to-understand, web-based, and data-visualizing dashboard.

OffsetExplorer is an application that enables a connection to an Apache Zookeeper and Apache Kafka Broker to display consumers, messages, and brokers. However, it does not display producers, and it does not appear to support later versions of the Kafka broker software [5].

Aurisano et al. designed a visualization to present data over a spatiotemporal axis, including both space and time, to show the movement of tracked animals within a region [6]. While the animal movements analyzed do not perfectly match the Kafka events, this paper is interested in, both share a spatiotemporal nature (Geographical vs. Kafka Locations).

III. Methodology

The steps of our methodology are laid out as follows.

A. Gathering Information and Planning

There are many dimensions a Kafka visualizer could expand into. In order to decide what should be present in the tool, an examination was made into existing Kafka architecture visualizers. A specific focus was the methods and applications by which a user could query the state of various nodes throughout that architecture and also examine how data was flowing through it. The SoftwareMill visualization was found during this stage. Its simplicity and direct focus were noticed [4].

The scope was narrowed to visually describe the connection between the components of Kafka that users can control, specifically the producers, topics, and consumers. Aside from some configuration tweaks, once the Kafka architecture is set up, developers are mostly coordinating the connections between producers and consumers. Because this is the area where Kafka developers spend their time, it became the focus of our visualizer.

It was determined that the key way to link together the various pieces of the Kafka architecture are via topics. Producers produce events that are stored in topics and consumers consume events based on the topic they reside in.

Matthew Jensen - matthewljensen@southern.edu
Miro Manestar - mirom@southern.edu
Germán H. Alférez - harveya@southern.edu
Miro Manestar, Matthew Jensen, and Germán H. Alférez are with the Department of Computer Science, Southern Adventist University, Collegedale, TN, USA

B. Connecting to a Kafka Broker

There are many ways to connect to Kafka. The developers at Confluent have created a tool called Kafka Connect that provides integrations to RDBMS' like Oracle, document stores like MongoDB, cloud object stores like Amazon S3, and cloud data warehouses like Amazon Redshift [7].

However, since the goal is to create a platform-agnostic visualizer, an API for javascript/node.js was preferred. That led to KafkaJS, a library that provides an Apache Kafka client that runs within Node.js. There are a multitude of visualizer libraries available for Node.js which made it perfect for the tool's design requirements. Not only does KafkaJS work within the chosen environment, it also provides ample documentation and community support.

C. Setup a Testing Environment

The simplest way to create a Kafka architecture is to set it up within Docker. Preconfigured docker images are available, and they are likely the starting place for new Kafka developers. This is perfect for this research work because the developers most likely to use a Docker instance to run Kafka are likely unfamiliar with how it works. They are our target audience.

However, a Kafka testing environment is only as good as the data passing through it. Scripts were created that start up a variety of producers and consumers which create and consume on specific topics. These nodes are then available for retrieval via the Kafka client to be displayed within the visualization.

D. Retrieving the Pertinent Information

KafkaJS makes the retrieval of most of the displayed information relatively easy. Topics can be queried and are returned as a list. Consumers can also be easily retrieved, along with a list of the topics each consumer is subscribed to.

Unfortunately, the information relating to the producers is not as readily available. Because producers do not maintain an active connection to the broker, their information is not available to simply query. There are multiple approaches that can be taken to still retrieve it, but all of them increase the complexity and overhead of our tool. The tool must know what producers exist, how to identify them, and how many events they are creating. The approach chosen to retrieve this information was to create secondary producers that tally up the number of messages being sent by their primary producer, and then periodically send this tally, along with a producer identifier, to a separate topic being consumed by the monitoring server.

E. Displaying the data

There are two different visualization methods the tool implements in order to communicate the architecture to users: a graphical representation of the Kafka architecture and an informational display which provides details about each node.

The JavaScript graphing library, Vis.js was used to visualize the producers, topics, and consumers as nodes in a directed graph. Both the producers and consumers connect to topics. The edges show underlying "data pathways" that connect producers and consumers.

An interface that is more focused on displaying in-depth information about each producer and consumer is also included. This is where the volume of data being produced can be seen and examined in more detail.

IV. Tool Architecture

Underpinning the research presented here is a union of several technologies, all working together to support our approach to the problem of visualizing a Kafka architecture. In this section, the design of the tool, the technologies used, and how they all work together is described. Figure 1 shows a visual map with the relationships between the technologies.

Apache Kafka is an open-source event streaming platform. It is primarily used for data pipelines requiring high performance, analytical data streaming, data integration, and high-availability applications[1]. It utilizes distributed servers to receive and store data. Another piece of the Kafka architecture are clients. There are two types of clients: producers and consumers. Producers create events which are then streamed to the servers and categorized by topic. Clients subscribe to these topics and events within these topics are streamed from the servers to them.

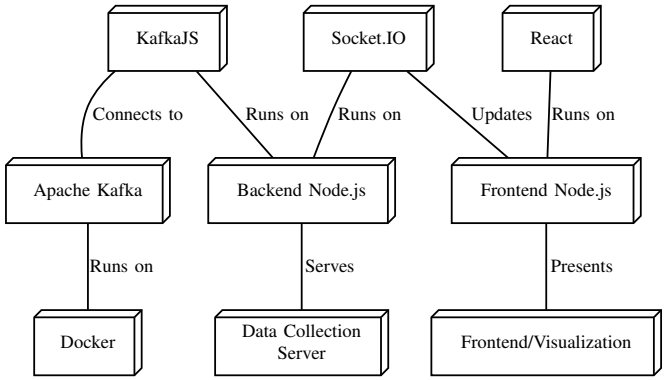

Fig. 1. Technology Stack. Read labels from top to bottom.

Docker enables the containerization of applications. It is useful in development and testing environments[2]. Docker runs a containerized version of Apache Kafka. This Apache Kafka container is the core piece of our testing architecture. JavaScript scripts are utilized to start producers and consumers, but all of the traffic is passing through this Kafka instance.

The Data Collection Server handles the collection of data from the Apache Kafka instance. It passively listens to secondary producers running within each producer. It can also dynamically start up consumers to get details information on specific topics and producers. This data is then stored and made available for access by the front-end. In order for our data collection server to access information from Kafka, KafkaJS is used. KafkaJS is an Apache Kafka client

[1]https://kafka.apache.org/

[2]https://docs.docker.com/get- started/overview/

for Node.js[3]. It runs in a Node environment which fits into this tool's tech stack because the data collection server is already running a Node environment. In order to send real-time updates to the front-end, to display statistics and message flows, Socket.io is employed. Socket.io is a Node.js library that enables real-time event-based communication between the browser and the server[4].

The front-end is where users can access real-time updates about their Kafka architecture, including currently active producers, topics, consumers, the way they all connect, and finally the data flowing through each topic. To this end, React was used for building a dynamic, JavaScript front-end for the tool. React makes it simple to create interactive and dynamic user interfaces[5], so it was determined it would be a good fit for the Kafka visualizer.

Both the front-end and backend require a number of libraries. The backend needs to run as a server, connecting to Kafka via the Kafkajs library and send updates to the front-end via the Socket.io library. Node.js is the best tool for this job. It is free, open-sourced, and creates a JavaScript run-time environment that enables server-side scripting[6]. It runs the various libraries and frameworks needed for both the client and server.

V. Results

This section describes the findings that resulted from this research. First, an examination is made of the class used to store and interact with the producer data. Next, the method of producer data retrieval is displayed. The user-facing aspect of this tool is the dashboard, comprised of the informational and graph visualizations. The source code of this tool is freely available on GitHub[7].

A. Data Storage

Data is collected by the Data Collection Server from two different locations. The consumer and topic information is collected via the KafkaJS Admin API. The producer information, which is not stored within Kafka, is collected from secondary producers.

The KafkaJS Admin API is relatively simple to work with and utilize to gather data. The data gathered from the Admin API is also relatively simple, specifically regarding the consumers and topics. Because of its simplicity and relatively static state, the data for those objects is simply stored in an array. This array is passed to the client and iterated through as necessary.

However, managing the producer information is more complex than the topic and consumer data. Listing 1 shows how the data is stored in a specific class with a couple of helpful methods. Lines 3-7 show the attributes associated with this class. Some have default initial values, like the produced attribute (line 7). Others are passed in at the time of the objects creation, like the topic and *createdAt* attributes (lines 5 and 6 respectively). Lines 15-17 show the setter method that updates the *lastUpdated* attribute to the current time, as well as increases the produced attribute's value according to the incoming data.

```
1 class Producer {
2   constructor(id, createdAt, topic) {
3     this.id = id
4     this.createdAt = createdAt
5     this.topic = topic
6     this.lastUpdated = createdAt
7     this.produced = 0
8   }
9
10  getProduced() {
11    return this.produced
12  }
13
14  updateProduced(produced) {
15    this.produced += parseInt(produced)
16    if (produced > 0) {
17      this.lastUpdated = new Date()
18    }
19  }
20 }
```

Listing 1. The structure and properties of a producer.

B. Secondary Producers

Secondary producers work to provide amortized data to the Data Collection Server. Specifically, they provide data that is sent in regular intervals as a summary of what occurred during the interval to provide an overview of exactly what has been occurring within the network in a robust and efficient manner.

This data is first recorded and updated whenever a producer sends a message. This is accomplished by incrementing a global variable that holds the number of messages produced by that producer. Then, on an interval, a secondary producer sends a trace message containing this value to the Data Collection Server via an unmonitored topic. This topic is not monitored by the tool because it is a component of the tool. Displaying it to the user would likely only cause confusion.

Listing 2 presents the function that generates the aforementioned trace messages before they are sent. Specifically, line 6 shows how the number of produced messages is passed in the body of the trace message. When received by the Data Collection Server, this will be added to the associated producer object. Lines 9-12 show that the *producerId, startTime*, and *topic* are all passed in the Kafka header. These fields are used to create the producer if it does not exist. Finally, on line 18, the global variable holding the number of produced messages since the last update is reset to 0.

Listing 3 shows how trace messages are received from within the data collection server and stored within the producer

[3]https://kafka.js.org/
[4]https://socket.io/docs/v4/
[5]https://reactjs.org/
[6]https://nodejs.dev/
[7]https://github.com/MatthewLJensen/kafka-visualization

```
async function generateTraceMessage(index, topic) {
  const message = [
    {
      key: index.toString(),
      value: JSON.stringify({
        numProduced: produced,
      }),

      headers: {
        producerId: clientId,
        startTime: createdAt,
        topic: topic,
      }
    }
  ]

  // Reset global produced variable
  produced = 0

  return message
}
```

Listing 2. Secondary producer trace message output.

class (see Listing 1). Specifically, this listing presents the code to create a consumer that receives metadata on a given topic. It can be seen that a consumer is created (line 2), connected to Kafka (line 4), subscribed to a topic (line 5), and a function is defined that is called every time a message is received (lines 6-24). Lines 8, 9, 13, and 17 show how the data from the message (specifically the *producerId, createdAt, topic*, and *numProduced* fields) are retrieved from the message body and headers. They are then converted to a string format within the same lines. Next, they are used to update the current state of the producer in lines 18 and 19. Finally, line 22 emits the change to the client, alerting them about a change in the producers. This emit event is fired by the Socket.io library.

```
async function consumeMetadata(topic, groupId, id) {
  const consumer = kafka.consumer({ groupId: groupId, clientId: id })

  await consumer.connect()
  await consumer.subscribe({ topic })
  await consumer.run({
    eachMessage: ({ message }) => {
      const producerId = message.headers.producerId.toString()
      const createdAt = message.headers.startTime.toString()

      let topic = null
      if (message.headers.hasOwnProperty('topic'))
        topic = message.headers.topic.toString()

      add_producer(producerId, createdAt, topic)

      const numProduced = JSON.parse(message.value.toString()).numProduced
      producers[producerId].updateProduced(numProduced)
      producers[producerId].topic = topic

      // Update frontend
      io.emit('producers', producers)
    }
  })
}
```

Listing 3. Creation a consumer that receives metadata on a given topic.

However, this arrangement presents a limitation. In order for the tool to function properly, the Kafka architecture itself must first be modified to provide the proper “hooks” for the server. While not the most optimal solution, this approach served as the most economical way to better understand the practical aspects of building such a tool.

C. Dashboard

Two different visualization methods were combined to create this dashboard. They have been linked together so that information in one can easily be used to reference the other:

1) Informational visualization: Figure 2 shows the information visualization that contains text information about each producer, topic, and consumer. Each producer displays its corresponding name, creation date, update date, number of produced messages, and topic. Next, each topic is listed. Finally, each consumer displays its corresponding name, group id, and the topics it is subscribed to.

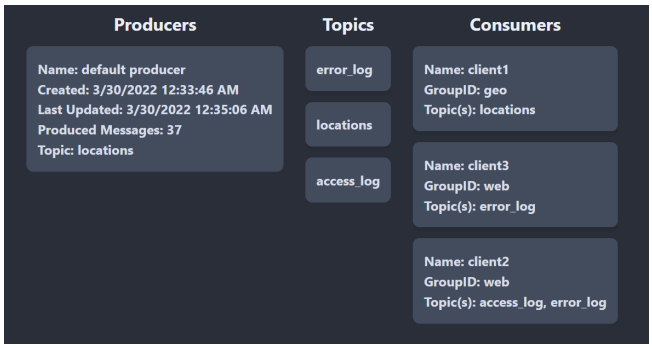

Fig. 2. The detail portion of the dashboard.

2) Graph visualization: The informational graph is useful when a user is looking at a specific node. However, it lacks the ability to provide a general understanding of the Kafka architecture. The graph visualization in Figure 4 is designed to display how the various parts of the specified Kafka architecture relate to each other. The graph updates in real time with the incoming data and is color coded to allow the user to more easily differentiate between topics, producers, and consumers. The producers are reddish pink, the topics are yellow, and the consumers are green. The direction of the arrows indicate what topics producers and consumers are either producing on or subscribed too.

The two distinct visualizations are directly interconnected within the tool. That is to say, selecting a node in either the graph or dashboard will select the equivalent node within the other visualization. This is to enable the user to easily highlight nodes of interest regardless of whether they are looking at a node in the detail view and are wanting to see how the node relates to the rest of the network or whether the user is looking at a node in the graph and wants to get more information about it. Furthermore, when a user selects a topic node, the tool displays the incoming messages from that topic live, as shown in the bottom right of Figure 3. Each message will only appear for a short amount of time before disappearing to make space for newer incoming messages. There is currently no way to view historical records of all received messages within the tool. Figure 3 shows how the two visualizations are connected and also demonstrates how incoming messages are

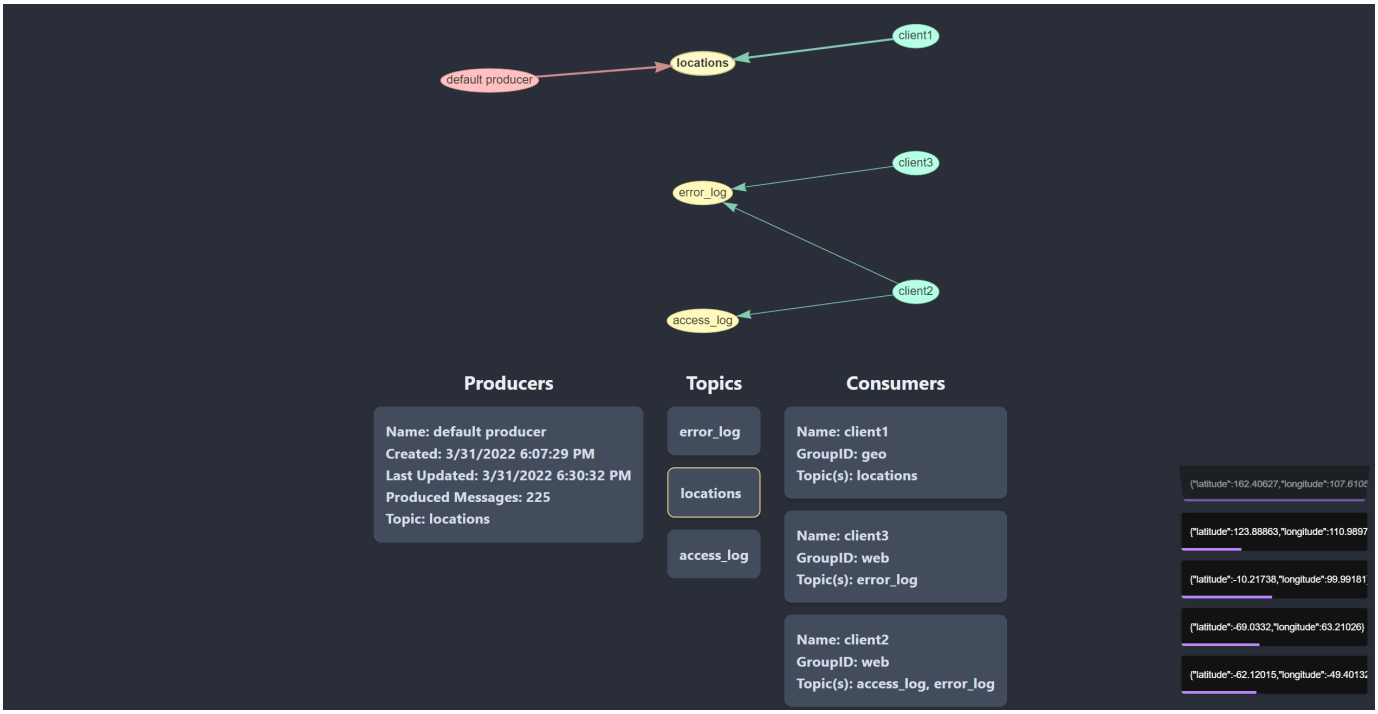

Fig. 3. The full view of the dashboard as presented to the user, including a preview of incoming messages on the right.

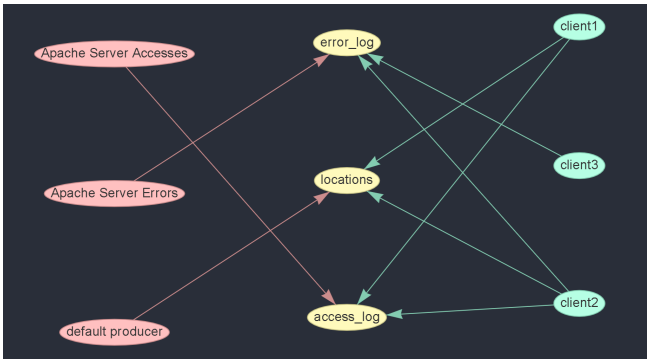

Fig. 4. Graph of an example Kafka architecture.

displayed in the bottom right. In this example the "locations" topic is shown to be selected within the graph by the bolding of its label and any connected arrows. Within the informational visualization it is show to be selected by its yellow border.

VI. Conclusions and Future Work

This paper presented a Kafka architecture visualization tool. Kafka is complex and difficult to properly configure. It can be confusing to users unfamiliar with its design. Given this complexity, this tool facilitates the visualization of Kafka architecture to practitioners.

However, the complexity of Kafka's APIs and the underlying implementation itself made the development of the dashboard difficult. The approach outlined in this research work came with some limitations. First and foremost, the tool developed is not "plug and play" for Kafka architectures. It requires modification of the Kafka architecture to include the aforementioned "secondary producers". Future work in this area would involve researching alternative methods for retrieving information about producers. For architectures sending less data, or where efficiency is not a priority, the Data Collection Server could itself subscribe to all topics and retrieve most of the pertinent information.

Another limitation involves gathering the flow of data throughout the application. As it stands, there currently exists no efficient implemented manner of watching how data flows from producers to consumers through the intermediary nodes and topics. Future work could visualize data in more dimensions that might show the user at a glance how data is actively flowing through the application, rather than just the architecture of the application and the status of its nodes. While it appears possible to implement such functionality, it would require careful research into the capabilities of the KafkaJS Admin API. More specifically, it would require the ability to carefully measure the flow of data through Kafka nodes. This is a task which the API makes non-trivial.

This data visualization tool could be improved by adding historical data. Future work could be invested in allowing a user to browse through data collected in the past. This would allow for better insights into how the architecture has behaved over time. It could possibly help track down long-running bottlenecks or other issues.

References

[1] "Apache Kafka. Homepage." [Online]. Available: https://kafka.apache.org/

[2] H. Wu, "Research proposal: Reliability evaluation of the apache kafka streaming system," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2019, pp. 112–113. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISSREW.2019.00055

[3] "Apache Kafka. Powered By." [Online]. Available: https://kafka.apache.org/powered-by
[4] "SoftwareMill. Kafka Visualization." [Online]. Available: https://softwaremill.com/kafka-visualisation/
[5] DB Solo, LLC, "Offset explorer." [Online]. Available: https://kafkatool.com/index.html
[6] J. Aurisano, J. Hwang, A. Johnson, L. Long, M. Crofoot, and T. Berger-Wolf, "Bringing the field into the lab: Large-scale visualization of animal movement trajectories within a virtual island," in *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, 2019, pp. 83–84.
[7] R. Moffatt, T. Berglund, "Kafka connect." [Online]. Available: https://developer.confluent.io/learn-kafka/kafka-connect/intro/