Collecting and Exploiting Performance Metrics of Kafka Streams Applications

David Benedikt Wetzel

Department of Computer Science, Kiel University stu126940@mail.uni-kiel.de

Abstract. Stream processing engines are increasingly used in huge and distributed systems, especially when a continuous flow of data is to be processed. Benchmarking can be used to analyze and evaluate the performance of a system and to make it comparable with other systems by measuring various indicators. To execute a benchmark, it is necessary to use performance metrics that give information about the analyzed system characteristics. In this paper, we present a monitoring approach that allows acquiring extensive metric data which can be used for executing benchmarks. Visualization can be helpful for controlling and monitoring systems and it is particularly useful for showing benchmark results. For that reason, our approach provides a dashboard solution that allows to visualize the monitored data. Based on this background, we focus on the collection of metric data, which are particularly relevant for benchmarking and monitoring Kafka Streams. In an experimental overhead evaluation, we analyze if the usage of the monitoring solution causes a relevant performance overhead. The evaluation shows that exposing debug-level metrics in Kafka Streams causes a noticeable overhead of 30 - 50% in comparison to exposing info-level metrics.

1 Introduction

In times of distributed, parallel, highly scalable digital systems, the relevance of monitoring and control possibilities is an important topic. In this context it is possible to use monitoring and benchmarking mechanisms to measure and control performance characteristics of a system. Monitoring and visualization can be used to show the relationship of different performance metrics at runtime or to map correlations. Another example for using monitoring is the controlling and visualization of benchmarks. Collecting metrics of a distributed platform can be a challenge, for example while a single system property is scaled within a benchmark. To differentiate between benchmarking and monitoring, we follow the definition of Bermbach et al. [1]: Both, benchmarking and monitoring, are used for the purpose of measuring quality and collecting information. Monitoring is the passive surveillance of a system with the smallest possible intervention. Benchmarking, on contrast, serves to answer specific questions, such as how scalable a system is. However, monitoring can be understood as a basis for benchmarking. In this paper we present a monitoring approach that also provides performance metrics for benchmarking. In the second step, we use benchmarking
methods to evaluate the degree to which monitoring influences the system and whether there is a monitoring overhead [12].

In this paper, we show a way to collect metrics of the stream processing engine Kafka Streams. For this purpose, we access provided metric data and store them centrally. Kafka and Kafka Streams provide metrics via the Java Management Extension (JMX). To process and store metrics we use the monitoring toolkit Prometheus. Prometheus is based on the concept of a pull-based time series database and enables various operations on stored metric data by using its own query language PromQL. For visualization we use the analysis platform Grafana. Grafana provides a highly customizable dashboard solution.

We structured this paper as follows: In section 2 we introduce the most important basics of Kafka and Kafka Streams. In section 3 we present the architecture of our monitoring solution. In section 4 we describe Kafka Streams performance metrics. In section 5 and section 6 we explain in detail the different kinds of metrics Kafka Streams offers, which we collect and how we can combine metrics for monitoring and benchmarking. In section 7 we present the results. In section 8 we summarize this work.

2 The Messaging Framework Apache Kafka and the Stream Processing Framework Kafka Streams

Apache Kafka [8] is a distributed platform for sending, caching and receiving messages. Messages correspond to key-value pairs, can be categorized in topics, and divided into partitions.

Kafka Streams [11] is a stream processing framework based on Apache Kafka. In Kafka Streams, data flow architectures are defined in topologies, where a topology consists of input and output topics and different operations (such as map or foreach). Each Kafka Streams Application is based on one or multiple topologies. Operations are performed by stream processor nodes which are connected by topics. A topology can be executed in a single task or in multiple tasks for parallel processing. In one Kafka Streams thread one or more tasks can be executed. In addition, the number of threads used within an application instance to execute Kafka Streams is variable. Instances are JVM processes, potentially running in different JVMs (e.g., when running in isolated Docker containers[1]).

Figure 1 shows the schematic structure of Kafka Streams.

3 Monitoring solution

For our monitoring solution we use various software components. The software stack can collect, store, analyze, and display the metrics. We propose an architecture processing metric data of a distributed microservices-based[9] platform. The microservices are implemented in Java.

[1] https://www.docker.com/
3.1 Architecture

Our approach starts with accessing performance metrics of each Kafka Streams application (in our case each Kafka Streams application is a microservice), storing the resulting metrics, and making them available. These metrics are visualized using a single-page web-application that can be used to monitor the system. The architecture can be divided into four sub-areas, which are described below.

Deployment The individual microservices, Apache Kafka, and Zookeeper are executed in Docker containers. The configuration enables the containers to interact with each other. We orchestrate the containers which Docker-Compose and Kubernetes.

Acquisition Kafka Streams provides various metrics by using the API of the Java Management Extension (JMX). This metric data is queried via Prometheus JMX exporters, which are integrated as Java agents. The JMX exporter provides metrics by a REST API via HTTP. This REST API is accessed by the monitoring toolkit Prometheus.

Storaging We apply the monitoring toolkit Prometheus as persistence of metric data. Prometheus accesses various JMX exporters via HTTP, which provide metric data of the related applications. Prometheus provides time series metric data by means of a HTTP API.

Visualization The analysis platform Grafana is used as visualization solution. We provide a single-page web-application via Grafana which enables data exploration and system monitoring.

Fig. 1. Schematic representation of a Kafka Streams application.
Fig. 2. Grafana variables. The user can interactively choose the metric and filter the results by selecting the label name and corresponding value.

Scalability At start time, our approach automatically detects the number of executed instances of a service and present Kafka Brokers in the cluster. Prometheus then collects automatically metrics from all endpoints.

3.2 Vizualisation

We use the Grafana analysis and visualization platform to display the metric values. For this purpose we divide the user interface into different Grafana ROWs. ROWs can serve for group and fade in or out several panels dynamically. In addition, we use Grafana variables, a concept that allows the dynamic creation of database queries by the user. Figure 2 shows the assigned Grafana variables.

4 Performance Metrics

Kafka Streams exposes various metrics using managed beans (MBeans) via JMX. A distinction is made between the two kinds of log levels: info- and debug-level.

Metrics are exposed at the following hierarchical layers: thread, task, processor, state store (for statful operations), record cache and suppression buffer. Depending on the log level Kafka Streams exposed different metrics for each layer. We cumulatively collect metrics about Kafka. Kafka also exposes metrics on several layers, in particular on server, producer, and consumer.

The info-level provides metrics on the thread layer, for more detailed metrics the debug-level must be activated. In the following, we introduce three important metric categories as examples:

Latency The latency of a system is an important measure to analyze, whether the system can handle the amount of data. Latency can be defined as the time interval between the input and the output of data. Kafka Streams exposes the metrics about latency at both thread and processor layer. Figure 3 represents the latency at both layers.

Throughput Throughput can be defined as the number of messages a streaming application is able to process in a predefined time interval. Kafka Streams provides throughput metrics at both levels by accessing the process-rate metrics on the thread and processor layer.

6 https://docs.confluent.io/current/streams/monitoring.html
## 5 Using Performance Metrics for Benchmarking

We can use collected performance metrics for benchmarking. The relevance of Kafka Streams metrics differs depending on the specific application. In the case of benchmarking, metrics that provide information about throughput and latency are of particular interest\[6,7\]. The Kafka Streams performance metrics make it possible to access measured values that are required for a benchmark.

**Relevant Performance Metrics for benchmarking** In section 4 we present some important Kafka Streams performance metrics. These metrics make it possible to determine the latency and throughput of a system. Another important metric is the consumer lag. The consumer lag describes the difference between produced and read messages. It can provide information about whether a consumer can process the messages at the same speed as a producer. Using the metrics of the type \textit{FetchManager}, Kafka provides direct information about the consumer lag and calls this \textit{record lag}. In \[2\] Ehrenstein discusses more relevant performance metrics.

**Correlations of Performance Metrics for Benchmarking** The combination of performance metrics can provide information about the reliability of a system and the quality of its scalability. For example, incoming message frequency can be related to processing latency, consumer lag or throughput. Especially in an isolated benchmark situation it is possible to test, for example, the maximum incoming frequency of messages without the latency exceeding a certain threshold.

## 6 Using Performance Metrics for Monitoring

Monitoring depends on collecting and analyzing various data and it is helpful to get an overview of the monitored system. In the following we monitor Kafka and Kafka Streams applications of the distributed Industrial DevOps \[3\] platform Titan\[4\]. Based on the method presented in section 3 we measure the microservices communicating with the Kafka messaging system. Furthermore we collect metrics of the Kafka Brokers. **Figure 4** shows the structure of the system.

**Structure of Titan Control Center** In Titan the microservices communicate via Apache Kafka. Individual microservices represent Kafka Streams applications. We analyze the Titan microservices aggregation, history, configuration, and stats. The aggregation microservice reads messages from a topic and calculates new records based on these values (create aggregatedActivePowerRecords
Fig. 3. Latency average by job label on thread (in ms) and processor (in ns) layer. The history microservice (yellow) stores the information in a Cassandra database and has the largest latency.

from incoming ActivePowerRecords). The new records are published to another topic. The other microservices only subscribe or publish certain topics.

**Dashboard solution** Our approach consists of a prototypical dashboard divided into four sections. In the upper part, the user can dynamically select the presentation of individual metrics as presented in [subsection 3.2](#). The other sections of the dashboard show information about latency, throughput and topic metrics.

**Boundary** In this work we focus on monitoring Kafka Streams applications. Monitoring can also be used to monitor Kafka performance characteristics and to detect bottlenecks or error conditions at the cluster layer [10].

**Findings** We visualized metrics on thread and processor layer. We found the highest latency at the History Service. Overall, the latency per job was more volatile than the throughput, which values were relatively constant.

### 7 Experimental Overhead Evaluation

Monitoring overhead can be defined as the additional resource consumption of a system while it is being monitored, compared to the same resource assumption of the system without being monitored [12]. In the following, we analyze the monitoring overhead of our system.

As explained in [section 5](#), we set the log level of the Kafka Streams application to debug. In this way we provide measurements of single processor and task layer in addition to metrics on thread layer. This approach is usually suitable for benchmarking if there is no relevant overhead by switching on the debug level or using the JMX exporters as Java agents.

For this reason, we use the Kafka Streams topology presented by S. Henning and W. Hasselbring [5]. We evaluate the impact of the activation of the debug
Fig. 4. Benchmarking- and Monitoringsystem. For the overhead evaluation we particularly use the lower area. For monitoring, we use the whole system without the workload generator and analyzer. In this case, the Kafka Streams applications are the microservices of the Titan Control Center.

level and the integration of the Prometheus JMX exporter on the performance of a Kafka Streams application. Henning and Hasselbring have performed a scalability evaluation to show which message frequency leads to which degree of capacity utilization of the stream application. Based on this analysis, we select two different message loads so that the streaming application will have in one case high and in the other case a low load. Similarly to the evaluation of Henning and Hasselbring, we measure the latency between the time the record is created and the time the record is read by an analysis component. Figure 4 shows the experimental setup.

Environment The overhead evaluation setup is deployed in a Kubernetes cluster with 4 nodes. Each node is hosted on the following hardware: CPU: 2 x Intel Xeon Gold 6130 (2 x 16 Cores, 64 threads), RAM: 384 GB, Cloud Network: 10GBase-T. In total we have 256 parallel threads.

Execution To analyze possible overhead factors, we examine four different scenarios: 1. No Java agent and info-level recording. 2. No Java agent and debug-level recording. 3. With Java agent and info-level recording. 4. With Java agent and debug-level recording. In the cases where JMX exporters are included as Java agents, Prometheus accesses them for data request.

We perform these four described scenarios in two different settings. In the first one, we fully utilize the streaming application (3 instances of the aggregation service), in the second one, we scale-up the streaming application, so that there is a low utilization (6 instances of the aggregation service). We repeat the benchmark 10 times for each configuration option. Each benchmark is run for 6 minutes, where we measure the last 5 minutes each time to avoid inaccuracies due to a warm-up phase.
Results We were not able to find a significant difference between the execution with and without JMX exporters as Java agent. However, the debug level increases the average latency noticeably. Figure 5 and Figure 6 show on the left side the average latencies and on the right side the percentage differences between the execution with different log levels. We have calculated the percentage deviation based on [12] as follows: Let $t_{debug}$ be the total time of an execution with log level on debug and let $t_{info}$ be the total time of an execution with log level on info. Then we get the monitoring percentage of the resource usage by $m = \frac{(t_{debug} - t_{info})}{t_{debug}} \times 100$. Our test shows that the latency increases by 35 – 55% using the debug level. In comparison we also found that the average latency decreases when we increase the instances of the stream application. In our test, the percentage overhead is about 10 – 20% lower when we create a low utilization of the topology.

The average percentage increasing of the overhead for three instances was 53%. When we scale up to 6 instances, the percentage increasing of the overhead decreases to only 36% percent. In both cases we calculated the average values of the scenarios with and without JMX Exporter, since there was no difference between these scenarios. This means, that in our evaluation the percentage overhead depends on the degree of capacity utilization of the Kafka Streams applications.
Collecting & Exploiting Performance Metrics of Kafka Streams Applications

Fig. 6. Execution with six instances of the Kafka Streams application. In this case the instances have a low utilization.

Table 1 shows the measured latencies, averaging all iterations. At this point, further overhead evaluation should research the correlations between the overhead and the workload of the Kafka Streams application.

8 Summary and Conclusion

In this paper we showed a way to collect Kafka Streams performance metrics using an open source software stack. Subsequently, we demonstrated the usage of performance metrics for benchmarking and monitoring Kafka Streams applications. Finally, in an experimental overhead evaluation, we evaluated whether the presented approach to data collection is associated with a high monitoring overhead. We found that the Kafka Streams log level debug provides a noticeable overhead, but the use of a JMX exporter as Java agent does not. We also

<table>
<thead>
<tr>
<th></th>
<th>debug + Agent</th>
<th>debug</th>
<th>info + Agent</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Instances</td>
<td>26190.35</td>
<td>26551.99</td>
<td>12996.68</td>
<td>11782.459</td>
</tr>
<tr>
<td>6 Instances</td>
<td>9109.714</td>
<td>9844.089</td>
<td>5649.162</td>
<td>6449.393</td>
</tr>
</tbody>
</table>

Table 1. Average latency for each benchmark scenario (all values in milliseconds).
found that the monitoring overhead correlates with increasing load on the Kafka Streams application. We provide a replication package and experimental results, to make it possible to replicate our work [13].

Finally, we conclude that the approach presented can be used for monitoring and for providing performance metrics for benchmarking. However, it should be noted that there is a noticeable overhead which should be avoided for productive systems in most cases. Future work could study the degree to which the overhead depends on the load.

References

2. Ehrenstein, S.: Scalability Benchmarking of Kafka Streams Applications. Studienarbeit, Institut für Informatik (February 2020)