Toward Efficient Scalability Benchmarking of Event-Driven Microservice Architectures at Large Scale

Sören Henning, Wilhelm Hasselbring
{soeren.henning,hasselbring}@email.uni-kiel.de
Software Engineering Group, Kiel University, Germany

Abstract

Over the past years, an increase in software architectures containing microservices, which process data streams of a messaging system, can be observed. We present Theodolite, a method accompanied by an open source implementation for benchmarking the scalability of such microservices as well as their employed stream processing frameworks and deployment options. According to common scalability definitions, Theodolite provides detailed insights into how resource demands evolve with increasing load intensity. However, accurate and statistically rigorous insights come at the cost of long execution times, making it impracticable to execute benchmarks for large sets of systems under test. To overcome this limitation, we raise three research questions and propose a research agenda for executing scalability benchmarks more time-efficiently and, thus, for running scalability benchmarks at large scale.

1 Introduction

A common practice in microservice-based architectures is letting microservices communicate in a publish-subscribe manner via a dedicated messaging system such as Apache Kafka. Regarding scalability, this means microservices may not only scale with the load at their REST endpoints, but also with the load of incoming messages. To meet scalability requirements, microservices employ tools and frameworks, which adopt streaming processing techniques, such as Apache Kafka Streams. An essential idea here is that individual microservices process (and potentially store) only a certain portion of the data stream.

While a lot of research goes into advancing the underlying principles for scalable stream processing [3], there is only little research on developing metrics and benchmarks for thoroughly evaluating the scalability of stream processing microservices. Good benchmarks have to fulfill requirements for relevance, reproducibility, fairness, verifiability, and usability [5]. In this paper, we present our Theodolite scalability benchmarking method, which is designed with special focus on fulfilling these requirements. Further, we point out necessary steps toward scalability benchmarks with Theodolite at large scale in a time-efficient manner.

2 Theodolite Benchmarking Method

Our Theodolite benchmarking method [9] allows to thoroughly benchmark the scalability of stream processing frameworks for microservices as well as of deployment options for such microservices.

2.1 Scalability Metric

Scalability describes the ability of a system to continue processing an increasing load with additional resources provided [2]. To compare the scalability of different systems under test (SUT), we therefore compare how their resource demand evolves with increasing load. Thus, our benchmarking method determines a *demand(intensity)* function [4], mapping load intensities to the resources which are required to handle these load intensities. In the following, we specify the terms "load" and "resources" in the context of event-driven microservice architectures.

Load Theodolite focuses on microservices, which are subject to a load of messages coming from a central messaging system. This load can have multiple dimensions, such as number of messages per unit time or size of messages. Even though Theodolite supports different load dimensions, we focus on scaling with the amount of distinct message keys per unit of time[1]. This key is used for data partitioning and, thus, the major means for parallelizing stream processing tasks.

Resources Microservices are typically deployed in (e.g., Docker) containers. Stream processing frameworks support scaling microservices by simply deploying more containers and thus multiple microservice instances. All necessary coordination and assignment of data portions is automatically performed by the framework. In this paper, we focus on horizontal scalability. Thus, the amount of required resources corresponds to the number of instances, which are required to process all incoming messages in time.

2.2 Measurement Method

Our benchmarking method approximates the resource demand function by creating a mapping of load intensities to required instances for a given set of load

[1] The key typically refers to the domain object an event refers to, such as the user ID when tracking user activities.

intensities. To do so, it also takes a set of number of instances and experimentally tests for each load intensity L and each number of instances I whether I instances are able to handle load L. These test are performed in isolated experiments as scalability does not contain any temporal aspect [2]. The number of necessary instances finally corresponds to the smallest number of instances, which is able to handle load intensity L.

The isolated experiments for assessing whether a given amount of instances is able to handle a certain load intensity are called *lag experiments*. In these experiments, we continuously monitor the *record lag*, which is the number of messages added to the messaging system but not being consumed yet by any instance. As the record lag is subject to high fluctuations, we apply linear regression to the time series and, thus, fit a trend line. We consider the number of instances as sufficient if the trend line's slope does not exceed a certain threshold.

The results of lag experiments may be significantly influenced by external factors, such as container placement or utilization of cloud servers. To increase statistical rigor [1], we thus allow repeating lag experiments several times and check their summary statistics against the configured threshold.

2.3 Provided Benchmarks

We provide four specification-based [5] benchmarks, which represent typical use cases for stream processing within microservices. These use cases are derived from an Internet of Things analytics platform [6, 8] and perform the tasks of storing messages to a database, aggregating messages in a hierarchical manner, downsampling the message frequency, and aggregating messages based on temporal attributes. The use cases are of different complexity and include typical stream processing operations such as joins, aggregations, and different types of windowing.

2.4 Benchmarking Framework

We provide implementations of all benchmarks for Kafka Streams as well as our Theodolite benchmarking framework as open source[2]. Our framework executes the proposed benchmarking method in a cloud infrastructure, operated by Kubernetes. It handles deployment, scaling, and monitoring of all benchmark components. This includes a distributed load generator, Apache Kafka as messaging system, and the actual SUTs, implementing our benchmarks.

3 Benchmarking at Large Scale

Theodolite allows to accurately assess the scalability of microservices that apply stream processing. Specifically, accuracy can be increased by testing more load intensities and number of instances and by increasing the number of repetitions. Increased benchmark accuracy, however, comes at the cost of significantly increasing execution time. For example, evaluating a single SUT for a single benchmark with 6 load intensities, 10 numbers of instances, 5 minutes execution time per experiment, and only 3 repetitions already requires 15 hours. This makes scalability evaluations for different benchmarks, stream processing frameworks, and deployment options expensive in time or resources. Therefore, an important challenge is to reduce the execution time for scalability benchmarks while preserving the presented accuracy. In the following, we raise three research questions and propose a research agenda to tackle this challenge.

3.1 Open Research Questions

RQ1: *How can the scalability metric be measured more efficiently?* Our current measurement method executes lag experiments for each combination of load intensity and tested number of processing instances. In particular, this includes experiments, whose results are unlikely to have any influence on the resource demand function. For example, if a certain load requires many instances, then it is unlikely that a higher load can be handled by only a few instances. We are thus looking for a way to reduce the number of lag experiments to be executed without significantly distorting the resulting demand function.

RQ2: *For how long should a single lag experiment be executed?* As we observe that the monitored record lag fluctuates strongly, we compute a trend line over a certain time period. However, to minimize the overall benchmark execution time, also the lag experiment's execution time should be kept as short as possible.

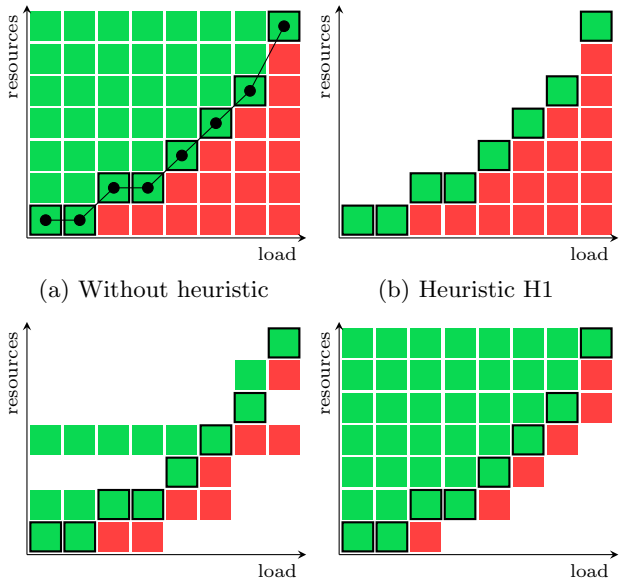
RQ3: *How many experiment repetitions are required?* Theodolite executes benchmarks in a cloud environment, which potentially causes significant variations among executions [7]. However, hundreds or thousands of repetitions are impracticable due to the execution time of single lag experiments as well as due to the number of different lag experiments, which are required to obtain the demand function.

3.2 Research Agenda

For RQ1: We are working on three heuristics, which already evaluate results during a benchmark's execution. Instead of running lag experiments for each load intensity with each number of instances, these heuristics apply a search strategy to find the number of required instances for each load intensity.

Heuristic H1 and H2 are based on the assumption that with additional instances, processing capabilities only improve. In other words, if I instances are able to handle a certain load intensity, then also $I + 1$ instances are able to handle that load. With H1, we test instances in an increasing order and stop when we find the first number of instances, which can handle the tested load. Heuristic H2 applies binary search to

[2] `https://github.com/cau-se/theodolite`



(a) Without heuristic (b) Heuristic H1

(c) Heuristic H2 (d) Heuristic H3

Figure 1: Comparison of suggested heuristics

find the number of required instances. This method is particularly advantageous if many numbers of instances should be evaluated.

Heuristic H3 is based on the assumption that with increasing load intensity the number of required instances never decreases. Thus, if we find that load L_x requires at least I instances, we can start testing next larger load L_{x+1} with I instances.

Figure 1 compares our suggested heuristics using an exemplary benchmark execution. Each cell corresponds to a lag experiment for a certain load intensity and amount of resources, which is executed by the respective heuristic. Green cells represent that the tested resources are sufficient to handle the respective load, whereas red cells represent that the resources are not sufficient. Framed cells indicate the lowest sufficient resources per load intensity. The resulting resource demand function is plotted in Figure 1a.

All heuristics are based on assumptions regarding the scaling behavior of SUTs. We plan to evaluate the individual assumptions for different SUTs to decide whether the respective heuristics are applicable. If this is the case, we also plan to combine heuristics to further reduce the number of lag experiments.

For RQ2: We plan to conduct dedicated experiments with different execution times. We expect that with increasing execution time, the trend line's slope stabilizes. We therefore identify the smallest duration, a lag experiment needs to obtain a trend line, which is sufficiently close to this stable value. By determining the necessary execution time for different resources, load intensities, benchmarks, and SUTs, we expect to assess the influence of parameters on the necessary execution time. Thus, our benchmarking method could potentially adjust the execution time per experiment.

For RQ3: With dedicated experiments, we plan to quantify the scattering of lag trends among multiple repetitions. Based on these results, we can also quantify the statistical error when only conducting a few repetitions. Again, these experiments should be performed for different resources, load intensities, benchmarks, and SUTs to increase the statistical validity and to assess the influence of such parameters on scattering.

4 Conclusions

With Theodolite, we present a metric, a measurement method, and corresponding implementations for benchmarking scalability of event-driven microservice architectures. In order to increase benchmark accuracy as well as simply executing more benchmarks, efforts must be made to reduce the benchmark execution time. Promising approaches in this regard are reducing the number of lag experiments, the execution time of single lag experiments, and the number of repetitions. Once these approaches are successfully implemented, different stream processing frameworks and deployment options can be benchmarked and compared to each other.

Acknowledgments This research is funded by the German Federal Ministry of Education and Research (BMBF) under grand no. 01IS17084B.

References

[1] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *SIGPLAN Not.* 42.10 (2007).

[2] A. Weber et al. "Towards a Resource Elasticity Benchmark for Cloud Environments". In: *Proc. Int. Workshop on Hot Topics in Cloud Service Scalability.* 2014.

[3] T. Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proc. VLDB Endow.* 8.12 (2015).

[4] N. R. Herbst et al. "BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments". In: *Proc. IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* 2015.

[5] J. v. Kistowski et al. "How to Build a Benchmark". In: *Proc. ACM/SPEC International Conference on Performance Engineering.* 2015.

[6] S. Henning, W. Hasselbring, and A. Möbius. "A Scalable Architecture for Power Consumption Monitoring in Industrial Production Environments". In: *Proc. IEEE International Conference on Fog Computing.* 2019.

[7] A. Papadopoulos et al. "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Trans. on Software Engineering* 01 (2019).

[8] S. Henning and W. Hasselbring. "Scalable and Reliable Multi-Dimensional Sensor Data Aggregation in Data-Streaming Architectures". In: *Data-Enabled Discovery and Applications* 4.1 (2020).

[9] S. Henning and W. Hasselbring. "Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines". In: *arXiv preprints* (2020). arXiv: 2009.00304.