Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 136 (2018) 99–108

Procedia Computer Science

www.elsevier.com/locate/procedia

7th International Young Scientist Conference on Computational Science

Optimising Kafka for stream processing in latency sensitive systems

Roman Wiatr[a,b,*], Renata Słota[b], Jacek Kitowski[b]

[a]*Codewise, ul. Lubicz 17G, 31-503 Cracow, Poland*

[b]*AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. Adama Mickiewicza 30, 30-059 Cracow, Poland*

Abstract

Many problems, like recommendation services, sensor networks, anti-crime protection, sophisticated AI services, need online data processing coming from the environment in the form of data streams consisting of events. The novelty of the approach in the field of stream processing lies in a synergistic effort toward optimization of such systems and additionally needed client components working as a whole. Building a message passing system for gathering information from mission-critical systems can be beneficial, but it is required to pay close attention to the impact it has on these systems. In this paper, we present the Apache Kafka optimization process for usage Kafka as a messaging system in latency sensitive systems. We propose a set of performance tests that can be used to measure Kafka impact on the system and performance test results of KafkaProducer Java API. KafkaProducer has almost no impact on system overall latency and it has a severe impact on resource consumption in terms of CPU. Optimising Kafka for stream processing in latency sensitive systems we reduce KafkaProducer negative impact by 75%. The tests are performed on an isolated production system.

© 2018 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/3.0/)
Peer-review under responsibility of the scientific committee of the 7th International Young Scientist Conference on Computational Science.

Keywords: Performance Optimisation; Stream Processing; BigData; Kafka

1. Introduction

Distributed computing and Big Data analysis are still ones of important paradigms at present both in science and industry, so there is a need to adopt scalable solutions for message transport systems. Typically, the computations are arranged with many components, often implementing master-worker or producer-consumer design patterns. While adopting cloud computing the cost becomes the key factor being influenced by utilization of computing resources, hence optimization of their usage is worth to be studied. We have seen a shift in many fields of software development. It has changed the way in which the products are shipped to the clients. We can identify two paradigms of interests:

*Corresponding author. Tel.: +48 889 608 925
E-mail address: rwiatr@gmail.com (R. Wiatr), renata.slota@agh.edu.pl (R. Słota), jacek.kitowski@agh.edu.pl (J. Kitowski)

1877-0509 © 2018 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/3.0/)
Peer-review under responsibility of the scientific committee of the 7th International Young Scientist Conference on Computational Science.
10.1016/j.procs.2018.08.242

- SaaS or software as a service. SaaS in principle is delivering programs over the internet using company infrastructure to host functionality as well as storing data. A well-known example of SaaS is Gmail.
- IaaS or infrastructure as a service. IaaS is a form of cloud computing providing virtualized computing resources over the Internet. Amazon AWS, Microsoft Azure, and Google Cloud are examples of IaaS.

Building or shipping software using the SaaS paradigm requires virtual infrastructure costs to be taken into considerations. They depend on the type of resources delivered by an IaaS provider. For example AWS [1] ships virtual machines as:

- On-Demand instances. Pay at least for 60 seconds. No long-term commitments.
- Reserved instances. Discounts for a long-term commitment.
- Spot instances. Can be terminated with a two-minute notification.

Many problems, like recommendation services, sensor networks, anti-crime protection, sophisticated AI services, need online data processing coming from the environment in the form of data streams [2] consisting of events or messages. The events can be used to determine the system state in real time [3] [4], or stored for audit purposes. The same pipeline can be used to gather business-critical data for further analysis. The cost of system operation can usually be estimated with decent accuracy. To make use of the system introducing a client interface is necessary, which operational cost is typically not taken into account. Adding such an interface to the system may significantly increase CPU, memory consumption or latency, hence its optimization is necessary instead of adding extra cloud resources. In the case of latency sensitive systems, one has to be confident that adding the client interface will not cause severe degradation of latency. The novelty of the approach in the field of stream processing lies in a synergistic effort toward optimization of the system and the system client working as a whole, with focus on client interface optimization, since the client interface influences the system and vice versa, predated by analysis and identification of problems. Such a scope follows from the observation that most of the work has been done toward optimization of the system, leaving the client interface apart; in the case of resources shortage, its volume typically being increased by the price of cost, which could be hardly accepted by some companies. So, building a message passing system for gathering information from other often mission-critical systems can be beneficial, but also it is required to pay close attention to the impact it has on existing systems.

As a vehicle of our study, the Apache Kafka for stream processing with a Real-Time Bidding system has been adopted as a business model for online computational advertising. This work shows the cost in terms of CPU, memory and latency and possible cost reductions applicable to KafkaProducer, which is a client interface of Kafka. The methods described here can be adapted to other distributed queue API however, some methods are limited to the Java Virtual Machine (JVM) only.

2. State of the Art

The cloud providers offer products for building data transport and processing pipelines. Of course, the question arises about the cost, performance, and quality. Looking only on event transporting products, AWS offers at least two of them (distributed in SaaS model) targeting slightly different problems: SQS [5] and Kinesis [6].

SQS targets transporting events between services. It provides two type of messaging queues to achieve that goal. The first one is specialized at maximum throughput, best-effort ordering and with at-least-once delivery. The second one focuses on exactly-once and ordered delivery. In both cases when a message is consumed it is removed from the system and is no longer accessible for other services.

Kinesis is a distributed, persisted log. This means that contrary to SQS it is possible to access the same message twice. This is the desired property in the case one wants to make streaming data available for a wide variety of other systems.

It is worth noticing a new protocol called Aeron [7] [8] that may be used to create low latency and high throughput distributed queues. Aeron implements efficient and reliable UDP unicast, UDP multicast, and inter-process communication (IPC). Aeron is designed to provide high throughput with low and predictable latency. Aeron has also the ability

to archive communication to a persisted storage for later or real-time replay thus providing similar functionalities as a persisted log system.

There are other transport mechanisms available. Only a few compared in [9]. Another solution, Kafka [10], getting great interest at present, can be used as a replicated, persisted log like Kinesis. Kafka was developed in LinkedIn and now is an Apache project. The Rich Kafka ecosystem [11] suggests a wide range of use that can compete with Kinesis. Netflix reported using over 4000 Kafka brokers [12].

In [10] Jay Kreps et al. compare Kafka to ActiveMQ [13] and RabbitMQ [14] from a producer and a consumer standpoint. In both cases, Kafka has superior throughput. A more recent study [9] confirms Kafka superiority in terms of throughput compared to RabbitMQ but it also shows that Kafka has higher latency. None of these studies touch the problem of resource consumption of the client interface.

As noted in [2] an online data processing architecture can be built from multiple tiers of collection and process connected by one of the "Messaging Systems" (MS) for example Kafka, Kinesis or SQS among others. Messages are sent to the MS using a MS client interface. Messages transported by MS can be used by "Stream Processing" (SP) layer or stored for offline processing. In the SP layer, Flink [15] can use Kafka as source or sink of the data streams, meaning it can read a data stream from a Kafka cluster and write back the processed data. Kafka is the default MS that connects the processing infrastructure in Kafka Streams [16] and Samza [17]. In "Data Storage" (DS) layer Hadoop [18] can persist data transported by Kafka using Confluent HDFS Connector [19]. Given Kafka popularity and multitude of existing tools, the Kafka system is worth to be studied.

3. Contributions

A question arises how we should measure features of the system integrated by MS. This calls for a method to measure not only the MS cost but also the cost and impact of the integration on an application sending data to MS. For the purpose of assembling a client interface evaluation method, we focused on measuring KafkaProducer. Due to Kafka popularity, there is a multitude of benchmarks measuring Kafka throughput and latency in different configurations. Unfortunately, there are no benchmarks measuring the KafkaProducer impact on the system. KafkaProducer is an API for uploading data to Kafka cluster. In our work, we focused only on the Java implementation. In this paper we present:

- Pitfalls in selecting a method of latency measurements for latency sensitive systems as well as KafkaProducer impact on such a system.
- Garbage Collector (GC) impact on the system caused by KafkaProducer API call cost and excessive memory allocation resulting in CPU consumption increase.
- Conclusions and means to reduce the impact of KafkaProducer on the system by 75%.

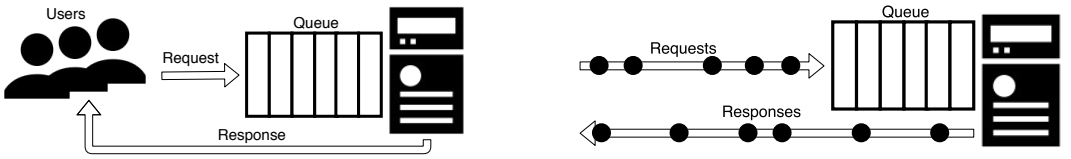
The methods described here can be adapted to other persistent log or distributed queue API however, some methods are limited to the Java Virtual Machine (JVM) only.

4. Performance Considerations

Measuring component impact on an application is always a hard task and can be seldom fully automatized. One can observe an increase in CPU consumption or increased latency but not always it can be directly linked to a specific component or a component part.

4.1. Latency

When measuring the latency one has to consider the type of load which the system will be subjected to in the production environment. We can model this load using the queueing theory. According to [20] the queueing networks can be classified into two categories: *open networks* and *closed networks*. The *closed networks* have no external sources nor sinks and can be further classified into two groups: the *interactive systems* and the *batch systems*.



(a) Interactive system model. (b) Open network model.

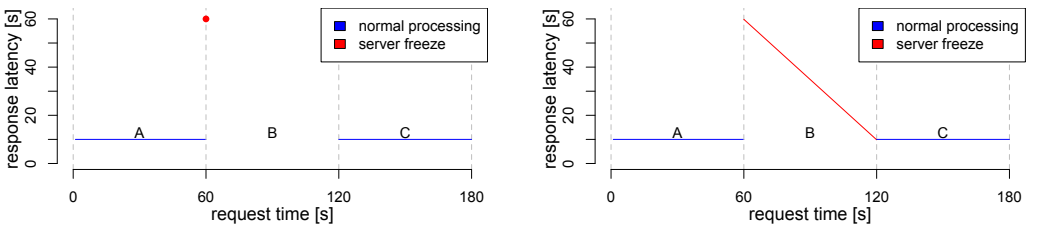
Fig. 1: Two classes of queueing networks.

Interactive systems (Fig. 1 (a)), referred to as the *closed systems* by [21], consist of a finite number of users sending request in a synchronous manner. Every user measures time interval between sending and receiving a request. Executing Alg. 1 in multiple threads it is possible to simulate multiple users querying the system and record query execution time. The *interactive systems* however are not well suited to measure a steady stream of requests. In the *interactive system* model when a system under measurement freezes, a phenomenon referred to as *coordinated omission* [22] occurs.

Algorithm 1: Latency measurement in the *interactive system* model.

```
1 Function SendRequest( request ):
2     startTime ← currentTime()
3     socket.send(request) // socket.send() blocks until server responds
4     return currentTime () - startTime
5 End
```

For illustration purposes consider the *interactive system* model with a single user sending requests in a synchronous manner and the system handling them with 10 second latency when processing normally. We expect the recorded latency characteristics to be similar as in Fig. 2 (a). As an example, assume 3 periods of 60 seconds each. In stage **A**



(a) Interactive system model. (b) Open network model.

Fig. 2: Illustration of latency measurements.

the system reports 10 second latency. Given a single user it means that the system handles 6 requests in this period. In stage **B** the system freezes and responds to one request with a delay of 60 seconds. In stage **C** the system handles another 6 requests with 10 second latency each. This behavior is correct in case of the *interactive system*. Problems begin when an expected request rate is specified in the test. The *interactive systems* ignore the request rate when the server slows down and report optimistic results. As reported in [21] and [22] many tools measure latency in the *interactive system* model. System freeze can be common, for example, every Java application may be subject to Garbage Collection (GC) pauses. As mentioned in [22] measuring and characterizing latency is not an easy task and some false assumptions and measurement techniques could lead to incorrect results.

The *open network* model (Fig. 1 (b)), referred to as *open system* in [21], assumes the requests entering the system with a given rate. This property can be exploited to measure the system latency while taking system freeze under

Algorithm 2: Sending a message in an asynchronous manner simulating the *open network* model.

```
1 Function SendRequest( request ):
2     request.startTime ← currentTime()
3     socket.sendAsync(request) // non blocking request
4 End
```

Algorithm 3: Receiving a message in an asynchronous manner simulating the *open network* model.

```
1 Function OnRequestCompleted( response ):
2     request ← response.request
3     return request.startTime - currentTime();
4 End
```

consideration. Consider using the nonblocking algorithms shown in Alg. 2 and Alg. 3 while testing the system from the previous example. In Alg. 2 we record only the request start time and invoke an asynchronous call to the system. In Alg. 3 we use the previously recorded time to calculate the delay. In the system freeze stage **B** the nonblocking code would continue to send requests at a steady rate of 6 requests per minute. In Fig. 2 (b) we can see that the expected latency measurements are limited by a function of stage **B**. The mean latency in this region is 35 seconds per request with six measurements compared to 60 seconds per request with one measurement in the *interactive system* model. Using the *open network* model we have 6 times more samples of the system freeze, the latency threshold plots show

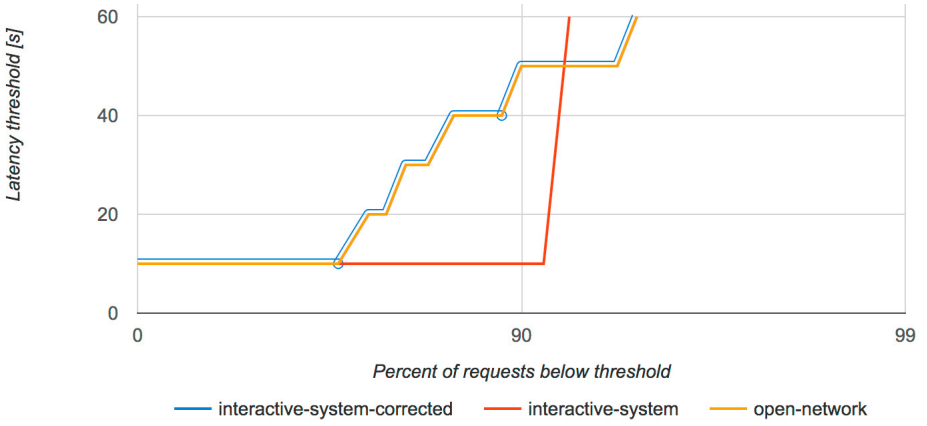


Fig. 3: Latency measurements in open network, interactive system and interactive system with correction.

degradation for lower latency values compared to the (interactive system) as shown in [22]. Fig. 3 shows how these theoretical measurements would look like on a latency threshold plot. Additionally we demonstrated how measurements taken in the *interactive system* model can be corrected using HdrHistogram [23]. According to the state of the art in latency measurements [22][23] the latency should be measured and interpreted using HdrHistogram, which can observe the complete latency distribution. HdrHistogram is designed for recording histograms of value measurements in latency and performance sensitive applications. As one can notice the *interactive-system* measurements with coordinated omission correction applied by HdrHistogram (*interactive-system-corrected*) are identical to the *open-network* measurements. Thanks to this correction, while measuring latency in the *interactive system* model, we are able to observe latency degradation on lower percentiles as if we were measuring latency in the *open network* model. Observing latency increase that could be omitted by the *interactive system* model is crucial for latency sensitive systems. To achieve this we used HdrHistogram post-recording correction method with the expected intervals between samples set to 10 seconds. This means that we expect the delay between requests not to exceed 10 seconds, and if it does HdrHistogram corrects the measurements by adding measurement values that could be observed in the *open*

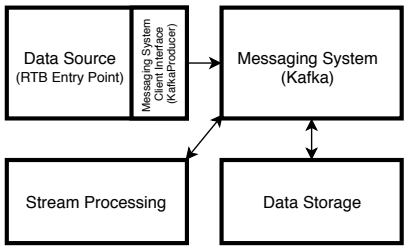


Fig. 4: RTB entry point extended by the Messaging System

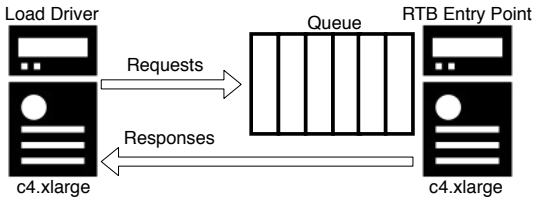


Fig. 5: RTB entry point latency test setup.

network model but are omitted because of the fact that the *interactive system* model has a limited amount of actors and a blocking nature.

4.2. Resource Consumption

Estimation of resource consumption is a hard task that requires setting up an environment and running the application. Usually, it is worth to measure the impact of a new component on the existing application. Insights gained from this task may lead to decisions regarding the environment change, cost estimates or component optimization. Resource measurements can be conducted in many ways:

- Using Linux *perf counters* [24].
- Storing system information in a time series database like *graphite* [25].
- Using dedicated performance monitors like *Java Flight Recorder* [26].

Usually, the second option is reasonably cheap in terms of resources and time. On the other hand, the third option gives access to data inaccessible otherwise (for example object allocation statistics). The first option can be easily converted into a time series stored in an external database. One has to be careful when choosing the appropriate metrics to measure. A good starting point is looking at limitations of our system. In the second step, one should look at limitations of the new component.

5. Experimental Evaluation

For the measurements, we have chosen a component serving as an entry point of a Real-Time Bidding (RTB) system (RTB entry point). In [27] RTB is defined as a business model for online computational advertising in which transactions take between 10 ms and 100 ms. A KafkaProducer has been added to the RTB entry point serving as an MS client interface (Fig. 4). KafkaProducer takes the messages handled by the RTB entry point and sends them as a data stream to the MS and thus exposing the messages to the SP and the DS layers (see section 2). In the RTB systems low latency is crucial and hence the impact of adding a new component should always be measured. Moreover, due to a high request rate, an appropriate model for the test should be considered to show early latency degradation or test results should be corrected (see subsection 4.1). Furthermore gathering additional information should not block the main processing path. To achieve a nonblocking architecture asynchronous buffers were introduced between the RTB entry point application four threads and KafkaProducer API. The buffers gather messages in a binary form and then send them as a single messages to the KafkaProducer. This solution leaves us with some decisions to make on how we should handle messages when buffers are full. We have three options: block the main processing (not acceptable in our case), grow the buffers or drop incoming messages until there is space in the buffers.

The RTB entry point under test was separated from other parts of the RTB system. The Load Driver (LD) produces requests to the RTB entry point (Fig. 5). The LD has a configurable amount of threads generating requests to the RTB system at a steady rate. Each thread waits for a response after a request is made. This means the LD tests the system in a *interactive system* model, thus it suffers from *coordinated omission* problem. We applied HdrHistogram post-recording correction method with expected interval of 10ms (see subsection 4.1). We configured the LD to send

10000 requests per second, each request was in *Json* format and 2.5KB in size. Both the LD and the RTB were run on a c4.xlarge instance each. According to the AWS specification, c4.xlarge has four virtual CPUs based on Intel Xeon E5-2666 v3 (Haswell) and 7.5GB memory. Due to the separation of the component and differences in the test and production environment, the results might not be strictly accurate. However, we were able to draw conclusions that could be applied to the production environment. The goal was not to have lower performance or higher resource consumption than the software baseline without KafkaProducer.

5.1. Latency

The tests were conducted using three configurations: the baseline without KafkaProducer (*no-kafka*), KafkaProducer with four 16MB buffers (*kafka+4x16MB-buffers*) and KafkaProducer with four 125MB buffers (*kafka+4x125MB-buffers*). The test for each configuration had 10 min intervals. Fig. 5 describes the test setup. Each test was repeated four times. Each test sequence was preceded by a 5 min warm-up period. Fig. 6 shows that the

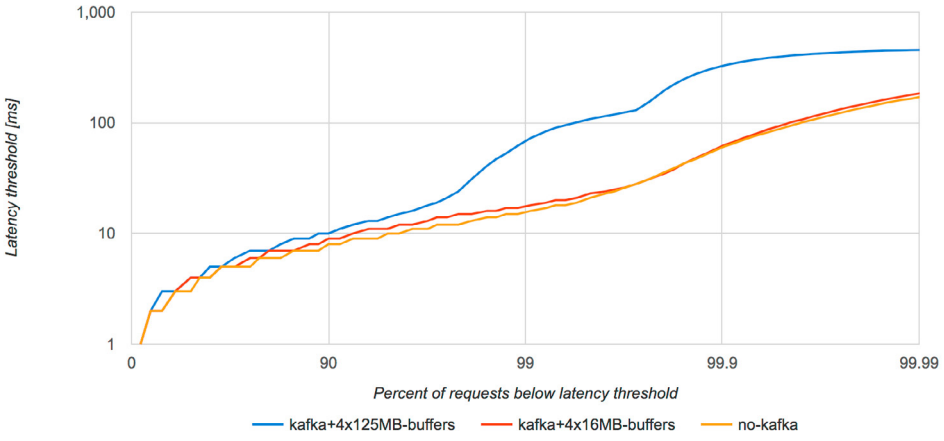


Fig. 6: RTB entry point latency test results.

latency increase is low yet visible. One can notice a slight increase in latency for the majority of the request but the overall performance of *kafka+4x16MB-buffers* is acceptable because it is similar to the *no-kafka* baseline performance. In both cases, 99.95% of the requests are below 100 ms. As mentioned before, the memory is limited on the c4.xlarge instances. We repeated the test with bigger buffers allocated (*kafka+4x125MB-buffers*). As one can notice using the bigger buffers (thus reducing available memory and potentially increasing GC pressure) can cause the system to slow down. In this case, the latency limit of 100 ms was broken at 99.4% of requests. This is an example of how resource sensitive applications can be. Since 16MB buffers did not increase latency beyond acceptable levels we decided to conduct more detailed tests with buffers between 4KB and 16MB. In subsection 5.2 we show that using 16MB buffers increases CPU consumption compared to using buffers of smaller size.

5.2. Resource Consumption

Resource consumption was measured in detail for CPU and memory. We noticed that increased memory consumption greatly increases the CPU usage due to GC pauses caused by inefficient memory management in KafkaProducer. Using Oracle Java Mission Control and Oracle Java Flight Recorder we were able to diagnose a set of problems with the memory allocation in KafkaProducer API version 1.0.0.

The first problem is related to the number of API calls. Every call allocates structures within KafkaProducer. We managed to solve this problem by reducing the number of calls made by the system. Each test took 10 min and was repeated two times. During the tests, we discovered that buffering messages before KafkaProducer and batching them as a single message into a single API call decreases the amount of memory allocated. We repeated the tests for different sizes of buffers and compare the results to a baseline without KafkaProducer and with KafkaProducer

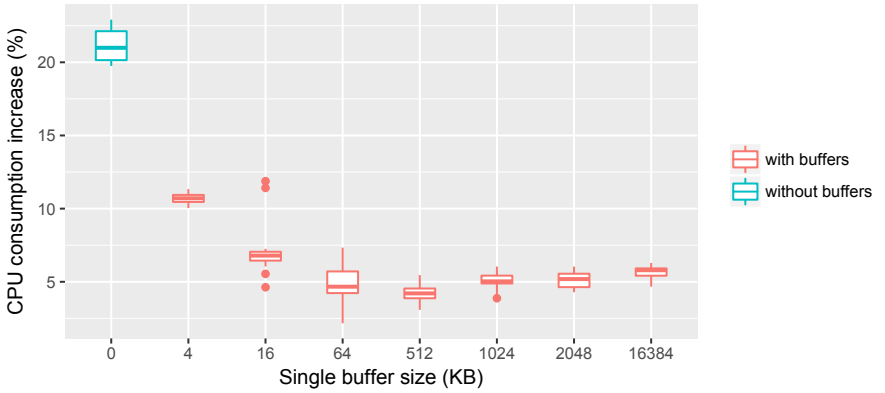


Fig. 7: CPU consumption test results.

```
public interface Serializer <T> extends Closeable {
    void configure(Map<String, ?> configs, boolean isKey);

    byte[] serialize(String topic, T data);

    @Override void close();
}
```

Fig. 8: Kafka Serializer interface.

```
default ByteBuffer serializeAsByteBuffer(String topic, T data) {
    return Utils.wrapNullable(
        serialize(topic, data)
    );
}
```

Fig. 9: Method added to Kafka Serializer interface.

without our buffering technique. Fig. 7 shows that the decline in the CPU resources required to send a message is proportional to the size of the buffer. Comparing the CPU measurements of the RTB entry point running without KafkaProducer, with KafkaProducer without buffers, and with KafkaProducer with additional buffers reducing the amount of API calls, one can see that we were able to reduce the cost of adding KafkaProducer from 21% to only 5% of additional CPU consumption using 1MB buffers and 4.5% using 512KB buffers. Further increase of the buffer size did not show improvement and at some point, it caused increase in CPU consumption and in latency as shown in subsection 5.1. To picture the amount of savings one could achieve using our solution, consider a cluster running 10 machines with auto-scaling when CPU goes out of range [40%-60%]. In the case of 21% CPU increase, the cluster would have to auto-scale up by four machines compared to only one machine for 5% CPU increase. Note that the tests were executed with the internal KafkaProducer parameter called *batch.size* set to 256KB. If it would produce CPU consumption savings similar to our method they should be visible between 4KB and 64KB buffer tests, and they are not. We experimented with other values of this parameter with no positive effect on CPU consumption.

The second problem is caused by unnecessary memory allocation and copying forced by the Serializer interface. Fig. 8 shows the interface of Serializer from the Kafka repository. As one can notice the *serialize* method returns *byte[]* (byte array) which in fact forces the programmer to create an implementation that will always allocate a new byte array and copy memory content. Because of this API design, KafkaProducer is allocating excessive amounts of byte arrays on the heap. Returning a byte array out of the Serializer makes it impractical to reuse the allocated memory, especially if objects serialize to arrays with variable length. Fortunately, Java provides an abstraction over memory (including byte arrays). *java.nio.ByteBuffer* acts as a mutable pointer to continuous memory that remembers its position as well as the memory start and end. With its help, we could allocate memory up front and reuse the memory avoiding allocation for every serialized object. This way we were able to reduce byte array allocation by almost 50%.

To minimize the impact on existing code we have added a method to the Serializer interface with a *default* implementation (Fig. 9). According to documentation a programmer extending an interface containing a *default* method in Java can: not mention this method and use the implementation provided by the interface; redeclare the method making it abstract; redefine the method providing a new implementation. In our case we use the *default* implementation to achieve backward compatibility with existing Serializer implementations. After this change, it was straightforward to

replace byte arrays with ByteBuffers where required. The next step was to redeclare serializeAsByteBuffer instead of serialize method in our Serializer implementation, returning our pre-allocated memory that was pointing to the start and end of the serialized object.

Another test was made using modified and original version 1.0.0 of KafkaProducer. The RTB system was not used in this test to get a clear picture of KafkaProcuder memory allocation depending on message size. Instead, we isolated the component and preallocated all memory required for messages up front. The environment configuration was as follows:

- KafkaProducer options: acks=1, retries=1, send.buffer.bytes=128KB, max.request.size=batch.size=4MB.
- Kafka topic with 32 partitions.
- 64MB file containing random data. Loaded into 'off heap' memory and splitted into N KB chunks.
- Tests were executed for N = 1, 16, 64, 512, 1024, 2048. After every test the JVM was terminated.

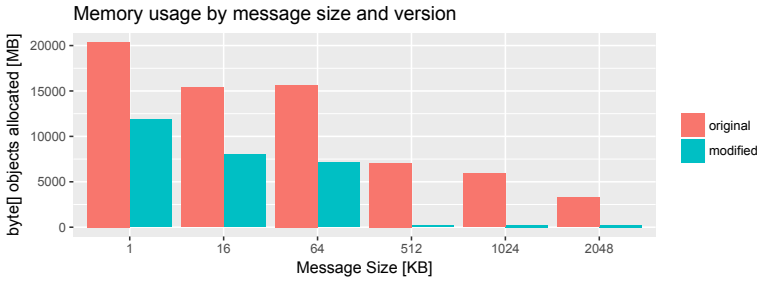


Fig. 10: Memory consumption test results.

In Fig. 10 the difference in the number of byte arrays allocated by the original and modified KafkaProducer versions is presented. For the message size of 1KB, the Java Flight Recorder measurements indicated that 19.9GB of byte arrays were allocated for the original client and 11.6GB for the modified version. Interestingly enough increasing message size also reduces byte array allocation which may indicate why batching messages reduces CPU consumption.

Table 1: CPU consumption increase caused by KafkaProducer with different buffer size applied.

	no buffers	4 (*KB*)	16 (*KB*)	64 (*KB*)	512 (*KB*)	1024 (*KB*)	2048 (*KB*)	16 (*MB*)
original	21%	10.7%	6.8%	4.7%	4.2%	5%	5.2%	5.8%
modified	-	11.3%	6.6%	4.6%	4.2%	3.6%	4.3%	5.5%

The last experiment was conducted using the environment from Fig. 5. Tab. 1 shows the increase of the CPU consumption median for different sizes of the buffer depending on the version of KafkaProducer. The optimal value of the buffer size for the original KafkaProducer is 512KB. Increasing the buffer size further, while using the original KafkaProducer, additional CPU consumption increases. For the modified KafkaProducer the lowest additional CPU consumption is at 3.6% for buffer size equal to 1024KB and this is the lowest point we were able to get to. The modified code has a negative impact for very small buffer sizes.

6. Conclusions and Further Work

In this paper, we provided measurements of the KafkaProducer API performance. We showed it has problems with the API invocation cost and memory allocation. We provided two solutions for reducing memory demands and CPU usage. We have proven that unnecessary memory allocation has a negative impact on CPU usage. Although the impact was not as severe as we thought it would be, reducing memory allocation can decrease CPU usage. In our case when using original KafkaProducer 512KB buffers gave best results with additional 4.2% CPU consumption. The

same level of additional CPU consumption could be observed for the modified version of KafkaProducer with 512KB buffers. We were able to lower the negative impact to 3.6% additional CPU consumption when a KafkaProducer with lower memory allocation and 1MB buffers. This means that we could benefit from using the modified version of KafkaProducer only when using buffers bigger than 512KB. The method of measurements, as well as the buffer design, may be extended to other MSs and their client interfaces. The reduction of CPU consumption on the client side can lead directly to cost reduction by scaling down the number of machines required to send data to the Kafka cluster.

Future work should include comparison tests with other MS client interfaces as well as more detailed measurements of memory allocation impact on CPU usage.

7. Acknowledgments

R.W. is grateful for his doctoral grant at AGH-UST that partially supports this work. He also thanks Codewise for the possibility to work towards his Ph.D. The research was done by R.W. as a part of a project at Codewise.

References

[1] AWS Pricing. https://aws.amazon.com/ec2/pricing/, . [Online; accessed 20-Jan-2018].
[2] Marcos Dias de Assuno, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing. *J. Netw. Comput. Appl.*, 103(C):1–17, February 2018. ISSN 1084-8045. doi: 10.1016/j.jnca.2017.12.001. URL https://doi.org/10.1016/j.jnca.2017.12.001.
[3] Peter Bailis, Deepak Narayanan, and Samuel Madden. Macrobase: Analytic monitoring for the internet of things. *CoRR*, abs/1603.00567, 2016. URL http://arxiv.org/abs/1603.00567.
[4] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1167–1178, 2015. doi: 10.1109/ICDE.2015.7113365. URL https://doi.org/10.1109/ICDE.2015.7113365.
[5] AWS SQS. https://aws.amazon.com/sqs/, . [Online; accessed 20-Jan-2018].
[6] AWS Kinesis. https://aws.amazon.com/kinesis/, . [Online; accessed 20-Jan-2018].
[7] Aeron. https://github.com/real-logic/aeron/. [Online; accessed 20-Jan-2018].
[8] Todd Hoff. Aeron: Do We Really Need Another Messaging System? http://highscalability.com/blog/2014/11/17/aeron-do-we-really-need-another-messaging-system.html, 2017. [Online; accessed 20-Jan-2018].
[9] Vineet John and Xia Liu. A survey of distributed message broker queues. *CoRR*, abs/1704.00411, 2017.
[10] Jun Rao Jay Kreps, Neha Narkhede. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, pages 1–7, 2011.
[11] Kafka Ecosystem. https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem/, . [Online; accessed 12-May-2018].
[12] Kafka Inside Keystone Pipeline. https://medium.com/netflix-techblog/kafka-inside-keystone-pipeline-dd5aeabaf6bb/. [Online; accessed 25-Jun-2018].
[13] ActiveMQ. http://activemq.apache.org/. [Online; accessed 25-Jun-2018].
[14] RabbitMQ. https://www.rabbitmq.com/. [Online; accessed 25-Jun-2018].
[15] Flink. https://flink.apache.org/. [Online; accessed 13-May-2018].
[16] Kafka Streams. https://kafka.apache.org/documentation/streams/, . [Online; accessed 13-May-2018].
[17] Samza. http://samza.apache.org/. [Online; accessed 13-May-2018].
[18] Hadoop. http://hadoop.apache.org/. [Online; accessed 13-May-2018].
[19] Confluent Kafka Connect. https://docs.confluent.io/3.0.0/connect/index.html. [Online; accessed 10-May-2018].
[20] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, New York, NY, USA, 1st edition, 2013. ISBN 1107027500, 9781107027503.
[21] Steffen Friedrich, Wolfram Wingerath, and Norbert Ritter. Coordinated omission in nosql database benchmarking. In *Datenbanksysteme für Business, Technologie und Web*, pages 215–225, 2017.
[22] Gil Tene. How NOT to Measure Latency. https://www.infoq.com/presentations/latency-response-time, 2016.
[23] HdrHistogram. http://hdrhistogram.org/. [Online; accessed 14-May-2018].
[24] Arnaldo Carvalho de Melo. The new linux 'perf' tools. In *Slides from Linux Kongress*, 2010.
[25] Graphite. https://graphiteapp.org/. [Online; accessed 29-May-2018].
[26] Advanced Java Diagnostics and Monitoring Without Performance Overhead. http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-wp-2008279.pdf, 09 2013.
[27] Yong Yuan, Fei-Yue Wang, Juanjuan Li, and Rui Qin. A survey on real time bidding advertising. *Proceedings of 2014 IEEE International Conference on Service Operations and Logistics, and Informatics, SOLI 2014*, pages 418–423, 11 2014.