Refactoring Kieker's I/O Infrastructure to Improve Scalability and Extensibility

Holger Knoche
Kiel University, Software Engineering Group
24118 Kiel, Germany
hkn@informatik.uni-kiel.de

Abstract

Kieker supports several technologies for transferring monitoring records, including highly scalable messaging solutions. However, Kieker's current I/O infrastructure is primarily built for point-to-point connections, making it difficult to leverage the scalability of these solutions.

In this paper, we report on how we refactored Kieker's I/O infrastructure to make better use of scalable messaging, improving extensibility along the way.

1 Introduction

Large software installations, especially those built on microservices, steadily produce large amounts of monitoring data. In order to process this data, the monitoring infrastructure needs to scale as well. A fundamental building block of such infrastructures are scalable messaging solutions, such as RabbitMQ[1] or Apache Kafka.[2] Using these solutions, it is possible to transparently distribute the load over a potentially large group of processing nodes, thus providing the required scalability.

However, to fully leverage the potential of these solutions, certain conditions must hold. In particular, a message producer must not assume that two messages will be processed by the same consumer. As a consequence, all messages must be self-contained, i.e. they must be processable without knowledge of any previous messages. Furthermore, sending and receiving a message may incur considerable overhead. Therefore, sending large numbers of small messages should be avoided.

Due to its extensible architecture, Kieker [1] supports numerous technologies for transferring monitoring data, including messaging solutions. However, several parts of Kieker's I/O infrastructure have been designed with point-to-point connections in mind. This is particularly true for the data formats and transfer protocols. For instance, the default binary data format uses string tables to avoid transferring string values redundantly. However, the string table entries are sent separately from the data. Therefore, decoding the data is impossible without additional knowledge, thus violating the self-containment requirement. In addition, the interfaces are designed for processing one monitoring record at a time, which may lead to an unneccessary high number of messages being sent.

In this paper, we describe how we refactored Kieker's I/O infrastructure to make better use of messaging solutions, thus paving the way for scalable monitoring based on Kieker. We furthermore highlight how these refactorings also led to an improvement of Kieker's extensibility. The remainder of this paper is structured as follows. In Section 2, we describe the situation before the refactoring in further detail. Section 3 presents the refactorings that were applied. A short performance evaluation is presented in Section 4, and Section 5 concludes the paper.

2 Situation Before Refactoring

Currently, Kieker's I/O infrastructure consists of two separate parts, namely readers and writers, which are contained in different components. While all readers are contained in the `kieker-analysis` component, all writers are part of the `kieker-monitoring` component. Dependencies between these two components are discouraged. Especially the `kieker-monitoring` component is supposed to have as few dependencies as possible since it is installed on the system to be monitored.

For each medium to write monitoring records to, a reader-writer pair is required. Writers are derived from the `AbstractMonitoringWriter` class, which requires the concrete writer to implement a method to write a single monitoring record. Unlike writers, readers are part of Kieker's configurable pipes-and-filters architecture and thus a bit more complex. They are derived from `AbstractReaderPlugin`, which requires them to implement a `read` method. In addition, readers define an output port to which the read monitoring records are delivered for further processing by other filters.

The choice of the data format as well as the transport protocol are left to the developer of the reader-writer pair. Most writers (and thus also the accom-

[1] `http://www.rabbitmq.com/`
[2] `http://kafka.apache.org/`

panying readers) use the `String`-based representation of the monitoring records (e.g., the filesystem writers) or the default fixed-length binary encoding provided by the monitoring records themselves (e.g., the TCP writer). Few writers such as the JMS writer use their own means of serializing and deserializing the monitoring records.

The current infrastructure has some flaws, which became apparent during the development of reader-writer pairs for the Advanced Message Queueing Protocol (AMQP) used by RabbitMQ, and for Apache Kafka. These pairs were supposed to use the default binary format, which, as previously noted, avoids sending character strings redundantly by means of string tables. However, the existing mechanism for synchronizing the string tables between record producer and record consumer turned out to be unsuitable for the intended use case. This mechanism works as follows. Once a new string is encountered by a writer, a special `StringRegistryRecord` is sent to the reader before the actual monitoring record, containing the string and the assigned numeric ID. The reader then updates its string table accordingly and is able to decode the record. While this mechanism works reasonably well with point-to-point connections, it does not work in message-based settings as it assumes that all records are processed by the same consumer.

A second flaw that became apparent was that there is currently no support for sending multiple monitoring records as a single chunk of data. Messages usually entail considerably more metadata than, for instance, a network packet. Therefore, sending one record per message can lead to a significant amount of avoidable overhead.

The third flaw encountered during the development results from the distribution of reader-writer pairs over two components. Such pairs usually require client libraries for the underlying technology, which are exclusively used by the reader and the writer. But since these are placed in separate components, the client libraries must be declared as project dependencies, leading to a high number of such dependencies. A similar issue exists for test cases. Readers and writers are usually tested together; however, there is no appropriate location for such tests.

3 Applied Refactorings

In order to address the previously described flaws, we refactored Kieker's I/O infrastructure as follows. The primary goals of the refactoring were (i) to introduce a reusable and flexible chunking mechanism, which allows to package a set of monitoring records together with the required metadata, and (ii) to restructure the I/O infrastructure in a way that reader-writer pairs can be co-located in a separate component together with their specific dependencies and tests.

For the first goal, we introduced the notion of a *collector* on the monitoring side. A collector takes

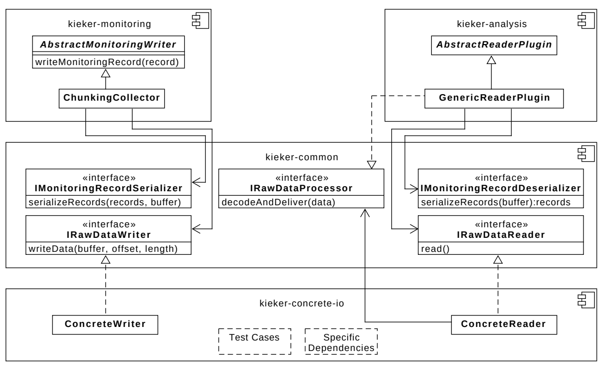

Figure 1: I/O infrastructure after refactoring

the place of the writer in the current state and is therefore derived from `AbstractMonitoringWriter`. The collector stores each incoming record in a queue, which is inspected by a separate thread in regular intervals. Once this thread detects that the queue contains enough records to fill a chunk or that the records have exceeded a given wait time, it removes the record from the queue and passes them to a *serializer*.

Serializers are the second new concept introduced by the refactoring. Since many client libraries operate on binary data such as byte arrays and byte buffers, we decided to restrict the responsibility of the writer to actually writing the raw data, and to provide reusable serializers for different data formats. For transferring monitoring records and metadata, we created an extensible container format capable of storing data in different formats. The default data format is based on the existing binary format, however, the string tables are generated individually for each chunk and packaged together with the record data. After the records have been serialized, the collector passes the raw binary data to the actual writer.

On the analysis side, a similar structure was created. Like the writer, the actual reader is only responsible for receiving the raw binary data and passing them to the further processing stages. The decoding process is orchestrated by a generic reader plugin, which deserializes the raw data using a deserializer and feeds the decoded records into Kieker's pipes-and-filters architecture.

In order to avoid unnecessary technical dependencies between the components, all participants of the process operate only on interfaces located in the `kieker-common` component. As evident from Figure 1, the concrete readers and writers can now be easily put into separate components together with their specific dependencies and tests, thus also achieving the second goal of the refactoring.

4 Performance Evaluation

Since major changes were made to Kieker's I/O infrastructure in the course of the refactoring, we inves-

Configuration	**95% CI** (in μs)	σ
Baseline	[106.0;106.2]	21.0
Current infrastructure	[152.8;152.9]	28.3
Collector (no bypass)	[163.3;163.6]	64.9
Collector (bypass)	[141.5;141.6]	40.3

Table 1: Response times measured by MooBench

tigated whether these changes led to significant performance changes. In particular, we were interested whether the reduced overhead due to fewer messages would make up for the additional complexity introduced by the collector and the (de)serializers. We investigated the following evaluation questions:

EQ1 *Does the refactoring have a (negative) performance impact on the monitored application?*

EQ2 *To what extent does the chunking affect the overall resource consumption when using a messaging technology?*

For investigating the first evaluation question, we created "null" writers for both the old and the new infrastructure which serialized incoming records, but discarding the serialized data afterwards. We then used the MooBench [2] benchmark to measure the monitoring overhead observable to the monitored application. The benchmark was run on a Raspberry Pi 3 running at 1.2 GHz using Raspbian Jessie Lite and Oracle JDK 1.8.0u144 for the `armhf` platform.

Since the monitored application and the actual writers were decoupled from each other by a queue in a previous refactoring [3], we did not expect noticeable changes. Surprisingly, as shown in Table 1, the benchmark revealed a slightly higher overhead for the new, collector-based infrastructure. As this additional overhead could be removed by bypassing the first queue and delivering the monitoring records directly to the collector, we assume that it was caused by synchronization issues due to the two involved queues, as it also occurred on a desktop machine. Thus, we conclude that the refactoring does not have a negative performance impact on the monitored application.

For investigating the second evaluation question, we created a reader-writer pair for AMQP for both the current and new, collector-based infrastructure. Each writer was then put into a test rig which issued monitoring records at a constant rate r. During a run, the test rig determined the amount of CPU time consumed by it once per second using the Java Management Extensions (JMX). The network throughput was measured using the `dstat` tool. As this tool measures the network load of the entire system, we only activated the SSH daemon on the Raspberry Pi, which accounted only for a few hundred bytes per second of network load during our tests. The respective readers were connected to a `CountingFilter` to make sure that the expected number of records were received.

Chunk size	**95% CI CPU** in CPU sec. / sec.	**95% CI net** in KiB
Old writer	[0.612;0.620]	[1,913.7;1,914.2]
1	[0.768;0.780]	[2,706.4;2,710.3]
16	[0.460;0.477]	[684.2;684.3]
32	[0.273;0.275]	[639.9;640.1]
128	[0.340;0.356]	[593.8;594.2]
1024	[0.338;0.352]	[577.3;581.1]

Table 2: CPU and network utilization for $r = 10000$ records per second

The test rig was again deployed on the Raspberry Pi, which has a 100 MBit ethernet connection. In order to make sure that the writer would be able to run at its full capacity, the message broker and the reader were deployed on significantly more powerful machines (an Intel Core i7-3770K with 16 GB of RAM and an Intel Core i7-4500U with 8 GB RAM, both with Gigabit network interfaces) connected to the same Gigabit ethernet switch.

As obvious from the results shown in Table 2, the network as well as the CPU utilization were reduced significantly compared to the current state if an appropriate chunk size was chosen. Notably, the CPU consumption increased after reaching a low at a chunk size of 32, an observation which we reserve for our future work.

5 Conclusions and Future Work

In this paper, we have presented our refactoring of Kieker's I/O infrastructure to make better use of scalable messaging, improving writer performance for such situations along the way. Furthermore, we achieved our goal of restructuring the infrastructure for better extensibility.

In our future work, we intend to further investigate the dependency between chunk size, resource consumption and record frequency, and to develop a dynamically scalable monitoring infrastructure based on the foundations created by our refactoring.

References

[1] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd International Conference on Performance Engineering.* 2012.

[2] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps". In: *Software Engineering Notes* 40.2 (2015).

[3] H. Strubel and C. Wulf. "Refactoring Kieker's Monitoring Component to Further Reduce the Runtime Overhead". In: *Proceedings of the 7th Symposium on Software Performance.* 2016.