# Refactoring Kieker's Monitoring Component to Further Reduce the Runtime Overhead

Hannes Strubel
Software Engineering Group
Kiel University, Germany
hstr@informatik.uni-kiel.de

Christian Wulf
Software Engineering Group
Kiel University, Germany
chw@informatik.uni-kiel.de

## Abstract

Kieker's monitoring component is tuned for a low runtime overhead. Nevertheless, we recently identified potential for improvement. Unfortunately, we could utilize this potential only by refactoring major parts of its architecture. In this paper, we describe these changes and discuss their advantages. Moreover, we present an evaluation which shows that our changes reduce the runtime overhead to 17% in our setup while simultaneously having a complexity of only 73%.

## 1 Introduction

Ten years ago, Kieker [5] was released for the first time. At that time, its monitoring component provides basic capabilities to record the runtime behavior of a given application. Over the years, Kieker's monitoring component has been continuously expanded to include new monitoring approaches and technologies (e.g., [2, 8]). In 2012, Waller et al. [6, 7] started to systematically analyze also non-functional attributes of Kieker. For monitoring frameworks in general, it is crucial to influence the application under monitoring as less as possible. Otherwise, the recorded data does not reflect the actual runtime behavior. As a result of their analyses, Waller et al. were able to significantly reduce the runtime overhead introduced by Kieker.

Recently, we identified further potential for improvement which however requires to change the internal architecture of the current monitoring component. In this paper, we describe these changes and discuss their advantages. The resulting monitoring component is not tailored to Kieker and thus can easily be adapted by other monitoring frameworks. Compared to the current version 1.12, its architecture is less complex and has a considerably lower runtime overhead. In our evaluation, we show that we reduced the complexity of the monitoring component to 73% and the mean overhead measured by MooBench [4, 6] to 17%.

## 2 Current Monitoring Component

Figure 1 shows the data flow of monitored runtime information encapsulated as records through the architecture of the current monitoring component. For the sake of brevity, we describe the data flow toward the asynchronous TCP writer which writes the collected runtime information to a TCP stream. All other writers are executed in a similar way.

A record, emitted by a probe, first reaches the monitoring controller which serves as a facade for Kieker. Then, it is delegated to the writer controller which passes the record to the blocking queue of the TCP writer. At that point, the application thread returns and the writer's worker thread takes control over the record. It reads the record from the queue and sends it out via the associated TCP stream. However, before sending, the record is compressed to save bandwidth. Each string attribute of the record is replaced by a unique 4-byte identifier (id). The id and the string are registered as a *registry record* in an internal registry maintained by the registry controller. Upon a new registration, the registry record is passed from the registry through the monitoring controller to the writer controller which puts the record into the writer's non-blocking queue. Afterwards, another worker thread of the writer takes the registry record from this queue and sends it out via TCP. In this way, the receiver is able to reconstruct the record's strings from the ids.

This monitoring architecture has some potential for improvement. Besides collecting runtime information, the application threads atomically increase the number of transfered records to identify the beginning of monitoring by testing for zero. They also check in the writer controller for the type of the incoming record to put it into the correct queue. In addition, the used implementation of queues is very slow which unnecessarily delays all application threads. Since the record compression uses an additional worker thread, the serialization is performed asynchronously although the registry records must be transmitted prior to their associated records. For this reason, the corresponding string registry is synchronized. Additionally, all writers are expected to use the string registry. Hence, there is a central registry maintained by the registry controller although some writers write uncompressed records as plain text to the console or to a text file. Finally, each writer has to declare and to manage its own set of worker threads which unnecessarily burdens the programmer with concurrency issues.

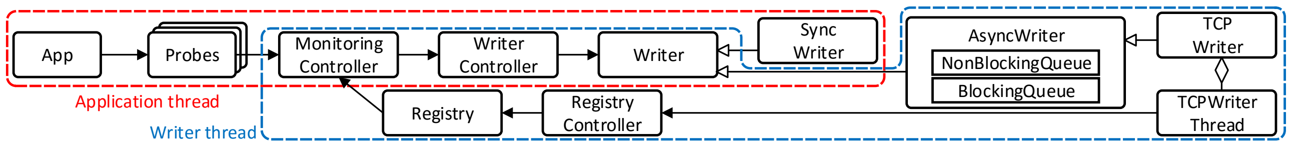

Figure 1: The architecture and the data flow of Kieker's current monitoring component

## 3 Refactored Monitoring Component

We refactored the current monitoring component in the following way. First, we reduced the effort of the applications threads by removing the atomic counter. Instead, we introduced an event-based system which triggers an event at the beginning of the monitoring. Moreover, we removed the registry records so that the application threads do not check for record types anymore. Instead, new string-id pairs are now registered and immediately serialized by the writer without the indirection via the controllers. In this way, we also avoid one of the two queues. We then replaced the blocking implementation of the remaining queue by a high-performance, lock-free alternative of JCTools[1]. Finally, we removed all synchronous writers to prevent programmers to use or to extend them. This restriction guarantees that the applications threads only add their records to queue and return immediately.

Second, we reduced the effort by the writer threads. Since we removed the registry records and their corresponding queue, we also removed their associated writer thread. Instead, the remaining writer thread first checks whether the string attributes of the record has already been registered and serialized. If so, it writes out the compressed record. Otherwise, it serializes the corresponding string-id pairs first. In this way, we avoid instantiating registry records and thus reduced the pressure on the garbage collector. If a writer does not make use of the compression, it does not declare a registry. Since the registry is now local to the individual writers, we removed the original registry implementation which is a synchronized map written completely by hand. Instead, we provide a faster, unsynchronized *writer registry* which reuses Java's default map implementation. Furthermore, we could now remove the central registry and the registry controller. Finally, we moved the thread management from the individual writers to the writer controller.

## 4 Complexity Evaluation

We expect a lower complexity from the refactored monitoring component due to the following main reasons: (1) It does not use a registry controller anymore. (2) It avoids a special handling of registry records by the writer controller. (3) It provides a single record queue and a single monitoring thread for all writer implementations. We use Eclipse Neon[2] with the Hypergraph-based Software Evaluation-Plugin[3] to apply the following three complexity metrics on all changed files for both Kieker versions: lines of code, cyclomatic complexity [3], and information complexity [1].

| Kieker Version | Lines of Code | Cyclomatic Complexity | Information Complexity |
|---|---|---|---|
| Current | 1092 | 2.48 | 242 |
| Refactored | (60%) 654 | (70%) 1.73 | (73%) 176 |

Table 1: The complexity of the current and the refactored monitoring component (1) in their lines of code, (2) in their cyclomatic complexity, and (3) in their information complexity. The percentages represent the fraction of the refactored to the current version.

Table 1 shows our results. The refactored version comprises only 60% of the original number of lines of code. Moreover, its cyclomatic and information complexity are only 70% and 73%, respectively, of the current version. Thus, the refactored version can be considered as less complex than the current version.

The reduced lines of code mainly result from the new, lean writer registry implementation and the removal of several, now obsolete classes, e.g., the registry controller. The cyclomatic and information complexity are reduced because we avoid to pass registry records through the monitoring component. Instead, each writer is now responsible for its serialization. Moreover, it does not need to create and to manage its worker threads anymore. Now, the writer controller takes over this task in a uniform way for all writers.

## 5 Runtime Overhead Evaluation

We expect a lower overhead from the refactored version due to the following reasons: (1) It is less complex (see Section 4). (2) It uses a high-performance, lock-free queue for all writer implementations. (3) It reduces the pressure on the garbage collector since it avoids creating registry records. (4) It does not count the number of processed records anymore by default.

To measure and compare both versions, we use MooBench [4], a benchmark for assessing the overhead of monitoring frameworks. We executed it on Debian 3.16.7 with the Java HotSpot 1.8.0_51 running on an Intel Xeon E5-2650 with 128 GB RAM. We configured MooBench to measure the "Discard Writer", which discards incoming records, and the TCP writer.

[1]`https://github.com/JCTools/JCTools`
[2]`https://eclipse.org/neon`
[3]`https://build.se.informatik.uni-kiel.de/eus/se/snapshot` in version 1.0.0.201608110354

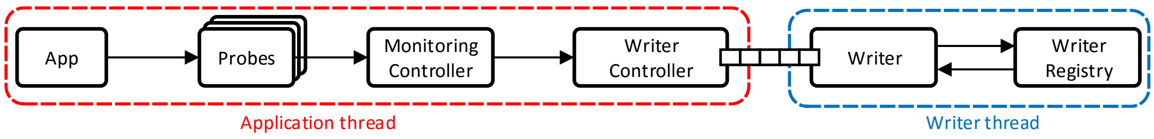

Figure 2: The architecture and the data flow of Kieker's refactored monitoring component

| Kieker Version | No Instr. | Deact. Probe | Data Coll. | Discard Writer | TCP Writer |
|---|---|---|---|---|---|
| Current | 0.087 | 0.478 | 2.607 | 18.932 | 19.715 |
| 95%-ci (±) | 0.000 | 0.005 | 0.007 | 0.014 | 0.015 |
| Refactored* | 0.088 | 0.468 | 2.335 | 12.424 | 16.855 |
| 95%-ci (±) | 0.000 | 0.004 | 0.007 | 0.007 | 0.035 |
| Refactored | 0.083 | 0.472 | 2.384 | 2.785 | 3.265 |
| 95%-ci (±) | 0.000 | 0.006 | 0.012 | 0.009 | 0.011 |

Table 2: Mean response times (in $\mu s$) of MooBench's internal operation call [4, 6] monitored by Kieker's current version and by the refactored version (* represents the version with Java's `LinkedBlockingQueue`).

We set the number of VM runs to 10 and the number of method calls to 20 mio. (method time of 0 ms; recursive depth of 10). We used MooBench's default warmup policy which discards the first half of calls (10 mio.) per run. We chose this configuration because it has yielded stable results for both versions.

In order to understand how much the change in the architecture and the replacement of the queue implementation contribute to the overhead, we applied MooBench on our refactored version first with Kieker's blocking queue and then with JCTools' MpSc queue. Table 2 shows the mean response time in $\mu s$ of MooBench's internal operation call monitored by Kieker's current version and by the two refactored versions. The first two phases "No Instrumentation" and "Deactivated Probe" do not show any differences if we include the confidence intervals and the OS's timer precision[4] of ±0.015 $\mu s$. However, the current version has a higher overhead in the "Data Collection" phase due to the atomic counter. The last two columns "Discard Writer" and "TCP Writer" show the results for the corresponding writers. We reduced the runtime overhead from 18.9 $\mu s$ to 12.4 $\mu s$ and, respectively, from 19.7 $\mu s$ to 16.8 $\mu s$. These improvements result from the lower complexity and the reduced pressure on the garbage collector as mentioned at the beginning of this section.

However, we reached the most significant speedup by replacing Java's `LinkedBlockingQueue` by JCTools' `MpScArrayQueue`. The results show that we reduced the runtime overhead from 12.4 $\mu s$ to 2.7 $\mu s$ and, respectively, from 16.8 $\mu s$ to 3.1 $\mu s$. Hence, the TCP writer of the refactored version has an overhead of only 17% compared to the current version.

[4]Measured with `https://git.io/v6gtt`

## 6 Conclusion

We described how the architecture of the current monitoring component looks like and why it hinders to further reduce the monitoring overhead. Then, we presented a refactored version and compared it with the current one. In short, we minimized the load on the application threads and replaced the blocking queue by a lock-free alternative. Our evaluation shows that we could reduce the runtime overhead to 17% in this way. This result is especially impressive because the complexity is also reduced, namely to 73%.

As future work, we plan to further reduce the overhead by pooling records in the data collection phase. This approach would avoid the runtime costs of frequently creating and removing record instances. Moreover, it could pre-fill records with static information such as the class and method name so that such information may not be collected at runtime anymore.

## References

[1] E. B. Allen, S. Gottipati, and R. Govindarajan. "Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach". In: *Software Quality Journal* 15.2 (2007), pp. 179–212.

[2] R. Jung. *An Instrumentation Record Language for Kieker.* Tech. rep. Kiel University, Aug. 2013.

[3] T. J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320.

[4] *MooBench.* URL: `https://build.se.informatik.uni-kiel.de/kieker/moobench`.

[5] A. Van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proc. of the ICPE.* 2012.

[6] J. Waller, F. Fittkau, and W. Hasselbring. "Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability". In: *Proc. of the Symp. on Software Perf.* 2014.

[7] J. Waller and W. Hasselbring. "A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring". In: *Proc. of MSEPT.* 2012.

[8] C. Zirkelbach, W. Hasselbring, and L. Carr. "Combining Kieker with Gephi for Performance Analysis and Interactive Trace Visualization". In: *Proc. of the Symp. on Software Perf.* 2015.