A Concurrent and Distributed Analysis Framework for Kieker

Nils Christian Ehmke, Jan Waller, and Wilhelm Hasselbring
Department of Computer Science, Kiel University, Kiel, Germany
{nie, jwa, wha}@informatik.uni-kiel.de

Abstract: Kieker is a framework for monitoring and analyzing software systems. The analyses performed upon the monitoring data are based on pipe-and-filter networks, that are usually executed on a single computer core.

In the case of computationally expensive and memory consuming online trace analyses, this basic approach is no longer sufficient. Our approach extends the Kieker framework by adding explicit support for concurrent and distributed analysis networks. The support for concurrency is realized by adding unbounded buffers to the pipe-and-filter networks and by consequently executing the resulting parts asynchronously. Support for distribution is added by aggregating parts of the analysis networks into analysis nodes. These nodes can then be distributed.

Our approach is evaluated and benchmarked in a series of thorough lab experiments. These experiments indicate a high communication overhead within our framework modifications. As a result, only little speedup can be achieved in the case of most analyses.

1 Introduction

Trace analysis can be a supporting tool for software engineers to handle the complexity of both legacy and modern, stable-running software systems [FWWH13]. However, providing a detailed online and interactive trace visualization is difficult. If the incoming data reaches the size of several gigabytes, it cannot be processed on a single computer anymore. Furthermore, the online trace analysis usually needs to be performed very fast or even in real-time, especially when combined with user queries.

Kieker[1] [vHWH12] is a Java-based monitoring and analysis framework. It provides the possibility to monitor, amongst others, Java software systems and to perform analyses on resulting monitoring logs. Kieker analyses consist of pipe-and-filter based processing networks. However, the framework has not explicitly been designed to support concurrent or distributed networks. An online trace analysis, for example, is therefore difficult to implement. To solve these shortcomings, our approach for concurrent analysis networks uses unbounded FIFO buffers within the network to execute parts of it asynchronously. The support for distributed analysis networks is realized by aggregating filters into analysis nodes, which can then be distributed on multiple systems.

[1] http://kieker-monitoring.net/

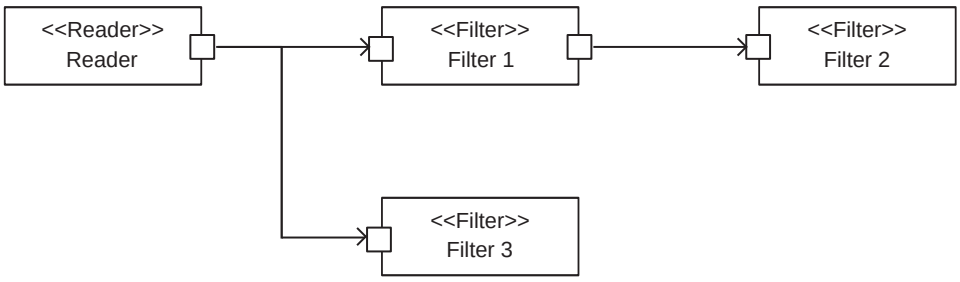

Figure 1: An exemplary analysis network

We perform various lab experiments to measure our approach with respect to performance and memory consumption, using different metrics. Based on our results, we make suggestions for improvements of the Kieker analysis framework.

In summary, our main contributions are:

- Presenting an approach for concurrent and distributed analysis networks with Kieker
- Evaluating the modifications with respect to performance and memory consumption
- Making suggestions for improvements of the analysis framework

The rest of this document is structured as follows. In Section 2, we describe the current implementation, the implementation of the concurrent, and the implementation of the distributed part. Section 3 contains a summary of the performed evaluations and our results. Existing work, related to our research, is presented in Section 4. We finally sum up the results in Section 5 and give recommendations for further improvements.

Note: Due to the limited scope of this paper, we vastly shorten the descriptions of the implementation and the evaluation. Details can be found in the corresponding master's thesis [Ehm13].

2 Implementation Details for our Framework Modifications

Kieker provides readers, filters, and repositories to assemble pipe-and-filter [HW04] analysis networks. Readers are responsible for reading, receiving, or generating monitoring records. They can be connected to filters. Filters can work on the given data (e.g., they can filter or enrich incoming data) and sent the results to their successors. Both readers and filters can be connected with repositories. Repositories act as shared data storages for these components.

An exemplary analysis network, consisting of three filters, is shown in Figure 1. A reader is connected with two filters, named Filter 1 and Filter 3. Filter 1 is also connected with another filter, named Filter 2. A monitoring record, read by the reader, is synchronously

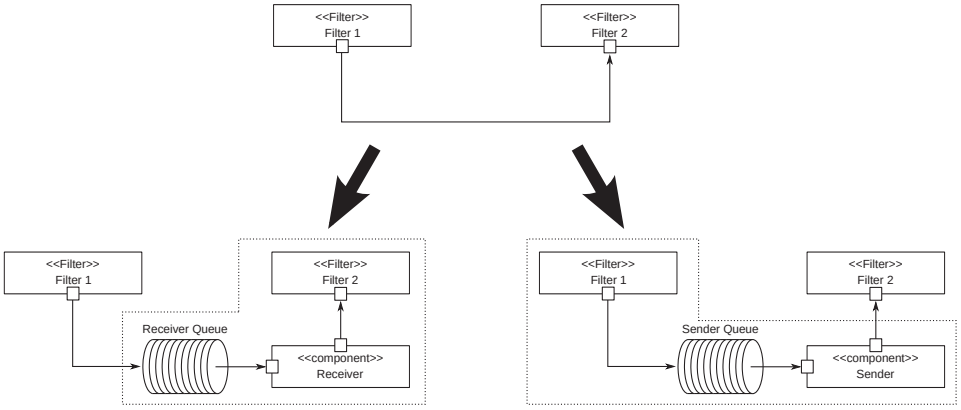

Figure 2: The basic concurrent design

sent to Filter 1 (i.e., the method call does not return immediately). Filter 1 processes the record and sends its result with a synchronous call to Filter 2. Once Filter 2 finished its calculations, the method call returns to Filter 1. As Filter 1 has no more successors, the method call returns to the Reader. Now the reader can send the original record to Filter 3. Once Filter 3 finished processing, the method call returns to the reader again. The record has been send to all three Filters and the reader can continue with the next record.

Note that the forwarding sequence is currently strictly sequential. The framework does not take advantage of the fact, that Filter 1 and Filter 3 could work in parallel. Furthermore, the current termination sequence is not developed for concurrent analysis networks. For a termination, each reader and filter would attempt to terminate its predecessors, itself, and subsequently its successors. In case of concurrent or even distributed components, this approach is no longer sufficient. We remedy these shortcomings in the following sections.

2.1 Design for our Concurrent Analysis Network

Our approach for concurrent analysis networks in Kieker is to extend the filters with unbounded FIFO buffers. Instead of sending data directly to successors, filters write the data into the buffers. The buffer is handled by an asynchronous thread. This results in a temporal decoupling of the filters. The basic design can be seen in Figure 2: A simple analysis network, consisting of two connected filters. In order to introduce concurrency, it is possible to set the ports of the filters into an asynchronous mode. The connection between the two filters is then enriched by two sub-components. The first component is an unbounded queue, storing the data to be received or sent. The second component, a concurrent thread, is responsible for the actual (synchronous) sending or receiving of the data from the buffer.

As discussed earlier, the current termination sequence used within Kieker is not sufficient for concurrent analysis networks. We introduce a new termination sequence more suitable for our concurrent design. Our approach is based on meta signals, which are sent and

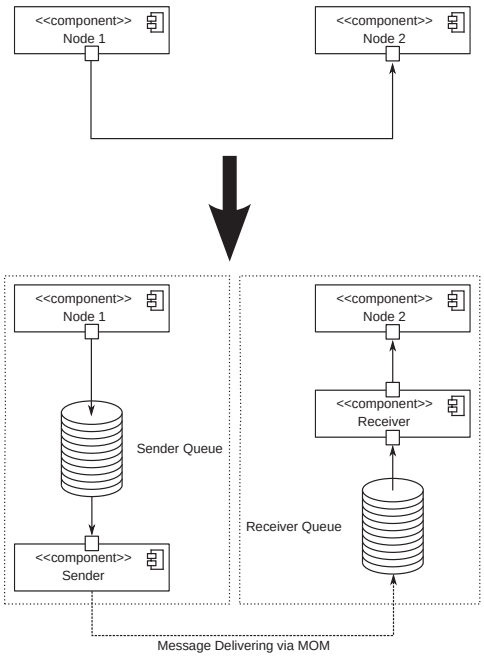

Figure 3: The basic distributed design

relayed through the network. As readers are the driving components within our networks, they send an initialization signal at start time and a termination signal at their own termination time. Filters use these meta signals to detect their own point of termination and relay them.

Our presented implementation allows to easily use concurrent analysis networks in Kieker. Only minor modifications at the configuration of the components are necessary in order to introduce concurrency to an analysis. It is even possible to use already existing analysis networks without further configuration, because our changes at the Kieker analysis API are very marginal.

However, it should be pointed out that our design also has various drawbacks. The main disadvantages are results of the modified termination sequence. Although the termination is autonomous and works for most cases, it is currently not possible to explicitly terminate a running analysis network via the API. Error cases (e.g. occuring exceptions within plugins), cycles within the networks, and filters without input ports are not handled correctly, either. Furthermore, our implementation does currently not support asynchronous repositories.

Table 1: Enterprise servers used for the experiments

Designation	Type	CPU	RAM	OS
Blade0	T6340	2 x UltraSparcT2+	64 GB	Solaris 10
Blade1	X6270	2 x Intel Xeon E5540	24 GB	Solaris 10
Blade5	T6340	2 x UltraSparcT2+	64 GB	Solaris 10
Blade6	T6320	1 x UltraSparcT2	64 GB	Solaris 10

2.2 Design for our Distributed Analysis Network

Our general approach for distributed analysis networks is the aggregating of filters into computation nodes. This also provides an abstraction for Kieker analysis. These nodes can be configured to run either in a distributed or in a local mode. If configured for the distributed mode, the basic design (see Figure 3) is similar to the concurrent one. We add unbounded FIFO buffers and use a sender and a receiver thread for each node. The sender blockingly reads from this buffer and sends the data to the following nodes. The incoming data is buffered in another FIFO queue on the receiving side. The data is read by a receiver and delivered to the plugins within the node. For the distributed delivery of data, we use ActiveMQ,[2] a JMS implementation, as a message broker.

Our implementation allows to distribute parts of an analysis network. Although involving minor effort, the filters of an analysis can be aggregated into nodes which can be executed on multiple systems. Again, existing analysis networks can be executed by the new framework without modifications.

The drawbacks of our distributed design are similar to those of the concurrent design. Furthermore, it is necessary to start the analysis nodes of a network in the correct order. This makes the usage of the distributed framework cumbersome.

3 Evaluation of our Framework Modifications

We perform the evaluation on various enterprise servers, using up to four servers for distributed analysis networks. The used enterprise servers and their configurations are listed in Table 1. The Java version for executing the experiments is 1.7.0_25.

Each of our experiment uses five to six data sets to compare up to five different synchronous, asynchronous, and distributed configurations. To provide a significant amount of data, we execute each of the configurations ten times with each of the five to six data sets. Each of these executions is repeated five times in the same JVM to provide an appropriate warm up. This is necessary to provide reproducable results despite of the JIT compiler of the JVM. For the statistical analysis of the data sets we use only the fifth repetition of each execution.

[2] http://activemq.apache.org/

The plotted values in the execution time diagrams are mean values. Additionally we mark double-sided 95 percent confidence intervals. Said intervals are calculated with a Student t-distribution. The unknown variance is estimated using the sampling variance.

For the efficiency, we are not able to exactly designate the number of used processors. Thus, we equalize the number of used processors with the number of used threads. We set the efficiency as the ratio between the speedup and the number of threads. To take the confidence intervals into account, we use the lower and upper limit of the speedup to determine the efficiency. We calculate the lower limit of the speedup between two execution times as the ratio between the upper limit of the first execution time and the lower limit of the second execution time. Analogous we calculate the upper limit of the speedup as the ratio between the lower limit of the first execution time and the upper limit of the second execution time.

The results for the concurrent analyses show an increased memory consumption. In some cases the needed memory is up to ten times higher than the one for the sequential analyses. The speedup, resulting from the use of additional concurrency, is usually only very low or even negative.

For the distributed analyses, the results indicate an increased memory consumption as well. The additional components and buffers require a high amount of memory. Furthermore, the performance is even worse. However, using records from the file system instead of creating them in memory, results in a throughput, which is at least in a similar order of magnitude as the one of the concurrent analyses.

In the following, we present excerpts from three of our experiments. Note that we modify the sequence diagram producing filter in order to write diagrams only in memory.

3.1 Experiment 1 - CPU and Memory/Swap Records Processing

For the first experiment we process records containing CPU and memory data. We use a load driver to generate said records in memory. The generated records are sent to a type filter. Depending on whether the incoming record is a CPU or a MemSwap record, the filter sends the record to one of two buffer filters. These buffer filters were originally intended to save memory, by buffering strings within incoming records in shared memory. For this experimental setup, we use the filter to construct additional computing load. The buffer filters relay the records to display filters. They are simply reading the record's contents to fill data models. The data models could later be used to visualize the record data.

The load driver is configured to generate $2^{16}, \ldots, 2^{21}$ data sets. Each of these data sets consists of 16 CPU records and one MemSwap record. The driver generates therefore $17 \cdot 2^{16}, \ldots, 17 \cdot 2^{21}$ records in our setup. The memory display filter is configured to store at most 20 entries. The CPU display filter is configured to store at most 20 entries per CPU. Older entries are removed in FIFO order.

We compare three types of configurations. The first configuration uses no asynchronous ports at all (*Synchronous*). For the second configuration we set all available input ports

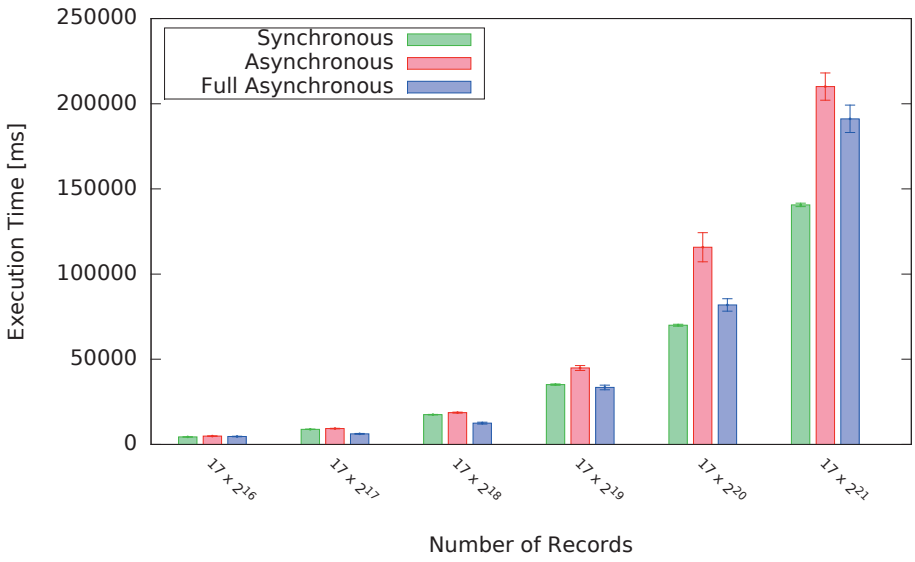

Figure 4: Results from a CPU and Memory/Swap record processing experiment

into an asynchronous mode (*Full Asynchronous*). A third configuration sets only the input ports of the buffer filters into an asynchronous mode (*Asynchronous*). The test system for this experiment is Blade 1.

One of the asynchronous configurations can improve the execution time for a specific amount of records. However, for a higher number of records the speedup gets worse and even negative, due to the communication overhead. The results are shown in Figure 4.

3.2 Experiment 2 - Trace Reconstruction

For the second experiment a trace reconstruction is performed. Incoming records are transformed and used to reconstruct message traces. The traces are sent to a sequence diagram producing filter. We use various load balancers with different number of outputs. We vary the number of outputs from one to four. We use also a synchronous configuration with one output and vary the number of threads therefore from one to five. The test system for this experiment is Blade 1.

Results of the experiment can be seen in Figure 5. Dotted lines show the theoretical minimal efficiency. The efficiency can be considered as relatively constant for each number of threads. It can also be seen that it is only slightly above the theoretical minimum. The framework can therefore take only very slight advantage of the additional parallel architecture during this experiment. This experiment reveals the immense communication overhead compared to the actual processing time within the framework.

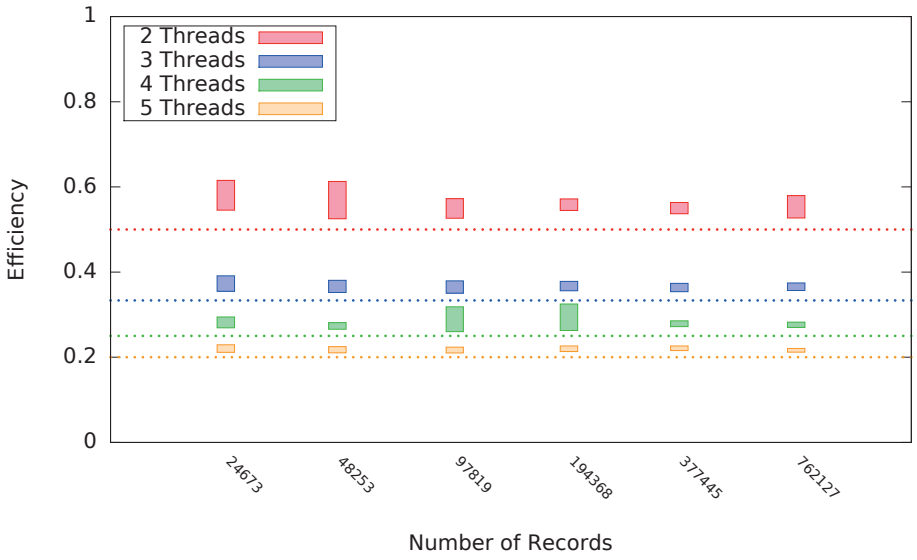

Figure 5: Results from a trace reconstruction experiment

3.3 Experiment 3 - Trace Analysis

For the third experiment, we use a modified configuration of Kieker's trace analysis tool. We test three types of configurations. A first configuration (*Synchronous*) uses the connections of filters as delivered by the original trace analysis tool. The used repositories within the analysis are not thread-safe . Thus we use a second configuration (*Asynchronous*). There we clone the necessary filters and repositories for each of the above mentioned outputs. For each of the cloned nodes we set the input ports into the asynchronous mode. This configuration performs more work than the first configuration. Thus we also test a third configuration (*Synchronous with Cloned Nodes*) in which we use the second configuration without asynchronous ports. For all three configurations we have to remove a filter printing the reconstructed system model. This is necessary as the filter contains no input port and is therefore difficult to use with our termination sequence. The test system for the first threee configurations is Blade 1. The fourth experiment is executed using all of the enterprise servers.

The data sets for the experiment are taken from monitoring the Kieker.WebGUI project. However, the noted number of records is not precisely the number of actual used records. In order to avoid invalid traces (and therefore further exceptions), we remove up to some hundred records per data set.

The execution times are shown in Figure 6. The asynchronous configuration is significantly slower than the first synchronous configuration for a lower number of records. However, the second synchronous configuration performs the same (additional) work and is much slower that the asynchronous configuration. Within a specific range, the asyn-

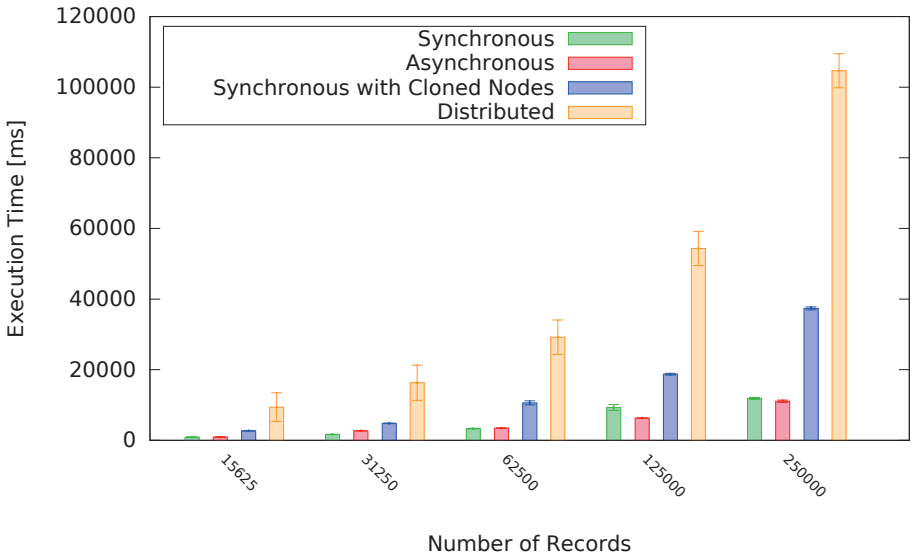

Figure 6: Results from a trace analysis experiment

chronous configuration can be used to speedup the trace analysis. With regard to the synchronous configuration with cloned nodes, the speedup is much higher. Furthermore, it can be seen, that the distributed configuration is significantly slower than all other configurations.

4 Related Work

Kahn Process Networks (KPN) are a Model of Computation (MoC). They consist of components, representing concurrent processes, which can be connected with each other to communicate. The connections, called channels, use unidirectional and unbounded FIFO buffers. Components can therefore write into the channels without being blocked, while the reading can block them until data is available [Kah74, LP95]. Although being only a MoC, the KPN can be considered as related to the work in this master's thesis. The networks can be described using a number of concurrent processes (which are similar to our plugins) and FIFO channels (which are similar to our FIFO buffers between connections). Although our approach is still slightly different than the one of the KPN, research about KPN and related MoC could still be useful for the Kieker framework. It should be noted though, that our approach is still slightly different than the one of the KPN. We do not use the components as concurrent processes, but rather the ports themselves. Also, our components can receive input from different ports. Processes within KPN can only wait for data on one of the available input channels [Kah74]. Furthermore, it is difficult to formally describe our networks using mathematical equations.

5 Conclusion

In this paper we present an approach to support concurrent and distributed analyses in Kieker. Our support for concurrent analyses is implemented by using FIFO buffers and sender/receiver threads between filters. For the distributed support we decide to use the MOM ActiveMQ for the message delivering. We aggregate filters into nodes and enable using them in a distributed way. In order to support a termination of the analysis, we implement an autonomous shutdown based on meta signals. We present results of three experiments to measure our modifications with different metrics.

Our results indicate that the framework modifications do not increase the performance in common analyses. The concurrent part of the framework usually leads only to little or no speedup. The resulting communication overhead is too high. We also realize, that the unbounded buffers between the filters lead to disproportionate high memory consumption. The distributed part of the framework does not lead to any speedup at all within our performed experiments.

In order to improve the performance of the concurrent part, we suggest a more suitable data structure (e.g., multi-producer one-consumer queues) for the buffers. It is also possible that bounded buffers would improve the memory consumption. Another possible approach for a concurrent framework would be to run each filter in an own thread and add bounded buffers between all connected components. Though it is still necessary to rework the termination sequence. For the distributed part we suggest to use either a decentralized solution or a direct connection instead of a MOM. A possible improvement would also be the batching of messages. This could reduce the number messages the MOM would have to handle, but would mean additional logic within the sender and receiver threads. More and other approaches for distributed analysis are researched in ExplorViz [FWWH13].

References

[Ehm13] Nils Christian Ehmke. Development of a Concurrent and Distributed Analysis Framework for Kieker. Master's thesis, Kiel University, 2013.

[FWWH13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In *1st IEEE International Working Conf. on Software Visualization*, 2013.

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 4th edition, 2004.

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the 6th IFIP Congress*, pages 471–475, Stockholm, Sweden, 1974.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd joint ACM/SPEC Int. Conf. on Perf. Eng.*, pages 247–248. ACM, 2012.