Development of a Concurrent and Distributed Analysis Framework for Kieker

Master's Thesis

Nils Christian Ehmke

October 5, 2013

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Dipl.-Inf. Jan Waller

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Kieker designates both a framework and a collection of tools to dynamically monitor and analyse software systems. One part of this framework enables to use a selection of interception technologies to collect various types of monitoring data. Another part of Kieker provides the possibility to perform analyses on said monitoring logs. This analysis part, which is the main focus of this thesis, consists of a flexible pipes and filters architecture. Analyses can be assembled using a range of existing filters or self-developed components. However, Kieker analyses usually run only on a single computer core. An explicit support for concurrency or distribution is not available.

In this thesis, we study an approach to extend the framework in order to support concurrent and distributed analyses. These modifications enable us to take advantage of additional memory or computational power in other systems or cores. Our main motivation for these modifications is enabling online trace analyses. With online trace analyses we designate the possibility to observe executions of running applications. Such analyses can help to understand the dynamic behaviour of complex software systems. Especially when combined with user queries, online trace analyses can be an important tool for maintenance tasks. It should be noted, though, that they can be very computationally expensive and memory consuming. Particularly enabling user queries is very likely to result in real-time requirements.

As a foundation, we present various performance metrics for an evaluation and give a short introduction to the architecture of Kieker. For the concurrent part of the framework we extend the existing filters with additional unbounded buffers. Instead of passing data to successors, filters write into these buffers without being blocked. This achieves a loose temporal linkage of the components. This means that it is not necessary for filters to wait for each other. For the distributed part we aggregate filters into analysis nodes. Nodes can be configured in order to run in a distributed way. The message broker Apache ActiveMQ takes over as a Message Oriented Middleware (MOM) to deliver messages between those nodes.

In order to compare the framework modifications with the old framework, we conduct a comprehensive evaluation. We perform various experiments on blade servers. Our main focus lies on performance and memory consumption during the experiments. The results indicate a very high communication overhead. Only little or no speedup for common analyses can be achieved using the concurrent framework. Due to the usage of a centralized MOM, the distributed framework is too slow for practical use in the Kieker environment. However, both frameworks could be used to speedup some computationally very intensive analyses which rely on only very few records.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API	Application Programming Interface
AOP	Aspect Oriented Programming
CPU	Central Processing Unit
CEP	Complex Event Processing
EMF	Eclipse Modeling Framework
FIFO	First In, First Out
GC	Garbage Collector
GPL	GNU General Public License
GQM	Goal Question Metric
JMS	Java Message Service
JMX	Java Management Extensions
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KPN	Kahn Process Network
MoC	Model of Computation
MOM	Message Oriented Middleware

Chapter 1

Introduction

In this master's thesis we extend the Kieker framework in order to support concurrent and distributed analyses. This chapter provides a first introduction to the topic, including our main motivation for the development. Furthermore we briefly define our goals and outline the structure of this document.

1.1 Online Trace Analysis

Trace analysis can be considered as an important tool to handle the complexity of modern software systems. According to Reiss and Renieris [2005], software can be large and performance issues in server systems are occurring only on occasion. They are of the opinion, that many issues are often influenced by the joined behaviors of the system. Furthermore the software engineering discipline often runs into legacy (information) systems with lacking, incomplete, or simply scattered documentation [Bisbal et al. 1997; Wong et al. 1995]. For all of these cases a view on the dynamic architecture can be desired. A special approach for distributed architectures is presented by Moe and Carr [2001]. They use collecting, statistical summary, and visualization of trace data to support software engineers to gain understanding of systems. More generally, Holten et al. [2007] explain why dynamic information can help software engineers to understand a system's behaviour. According to them, this understanding is important, for example, during maintenance tasks.

Some approaches use post-mortem performance analyses. Events are recorded during runtime and later analyzed using various tools [Wolf et al. 2004]. Other approaches provide dynamic visualizations of systems in action, but at the expense of details or program performance [Reiss and Renieris 2005]. Providing a detailed online and interactive trace visualization is difficult. The main challenge consists of the huge amount of data. Execution traces can grow extremely fast. Holten et al. [2007] consider that an observation of a few minutes could already result in single traces containing hundreds of thousands or even millions of calls.

To provide such an online trace analysis, it is therefore necessary to overcome two main difficulties. The first difficulty is the temporary buffering of trace data. If the incoming data reaches the size of several gigabytes, it is possible that it cannot be processed on a single computer anymore. The second difficulty is the processing of the data itself. Online trace analysis can be time consuming (especially when combined with interactive queries of users) and needs to be performed very fast. Online trace analysis has usually real time requirements.

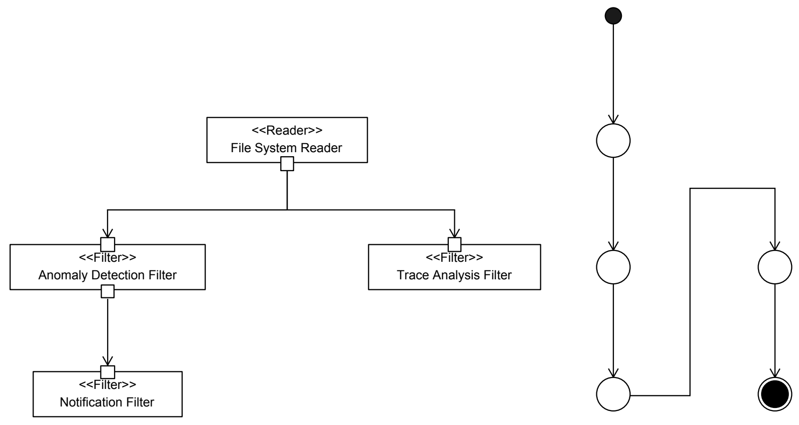
1.2 Monitoring and Analysis with Kieker

Data for trace analyses is often gained by using appropriate monitoring tools or frameworks. According to IEEE and ISO/IEC [2010], monitoring means, in terms of software engineering, to "collect project performance data with respect to a plan, produce performance measures, and report and disseminate performance information". Monitoring can help with topics like runtime failure detection, diagnosis, and reverse engineering [van Hoorn et al. 2009]. Monitoring is not only used during development, but also for software in productive use. It can be an important tool for both legacy and modern, stable running software systems.

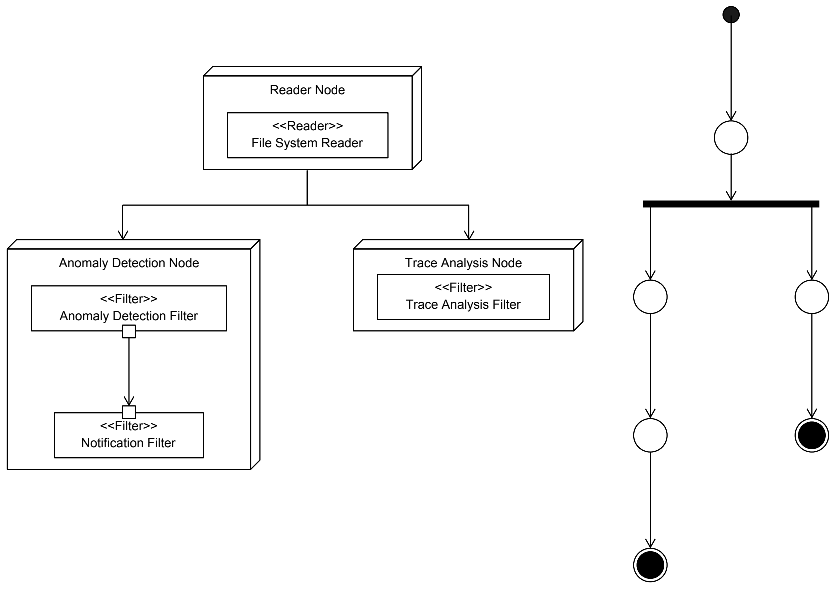
Kieker [Kieker Project 2013c] is one possible candidate to provide both monitoring and analysis. It is a Java based framework and tool collection for application performance monitoring and architecture discovery analysis. Developments at Kieker are performed by academic and industrial partners [van Hoorn et al. 2009]. It allows to monitor, for example, a running Java application and to collect runtime information. This information can be called methods and their respective response times or also the system's memory utilization. In order to obtain the reconstructed architecture or a visual representation of the response times, an analysis of the collected data is necessary. In Kieker such an analysis consists of a pipes and filters based processing network, which is not limited to the mentioned tasks and can be extended with further analysis components [Kieker Project 2013b].

However, the framework has not explicitly been designed to support concurrent or distributed components. An online trace analysis, for example, is therefore difficult to implement. Additional memory or computational power of other systems can only be used with disproportionate effort. Consider, for example, the situation in Figure 1.1a, consisting of two main tasks. The first task is the detection of anomalies based on the method response times. The incoming data is sent to the filter which then, if necessary, informs the Notification Filter to send an anomaly alert. The second task is the trace analysis, which could be used to visualize the current execution traces for a control center cockpit. Both tasks are theoretically completely different and independent.

A possible partitioning of the same analysis (Figure 1.1b) could be performed using three computing nodes. A computing node here would be another system or simply a thread running on another core. A first node could perform the reading, a second node the anomaly detection, and a third node the trace analysis. As mentioned before, such a partitioning is currently not explicitly supported by Kieker. Similarly, a physical distribution would require the implementation of additional network components and connectors.



(a) Configuration on a single computing node with sequential data flow



(b) Configuration with multiple computing nodes and parallel data flow

Figure 1.1. Exemplary analysis for anomaly detection and trace analysis

The speedup of long running analyses is not the only motivation for a distributed analysis framework though. The additional memory of other systems, for example, can be the only reason why a complex online trace analysis is possible in the first place. Some other important reasons are redundancy, load balancing, and cooperative computing. For the latter one could also imagine a system which merges the data of different subsystems. This could, for example, be an environment as shown in Figure 1.2.

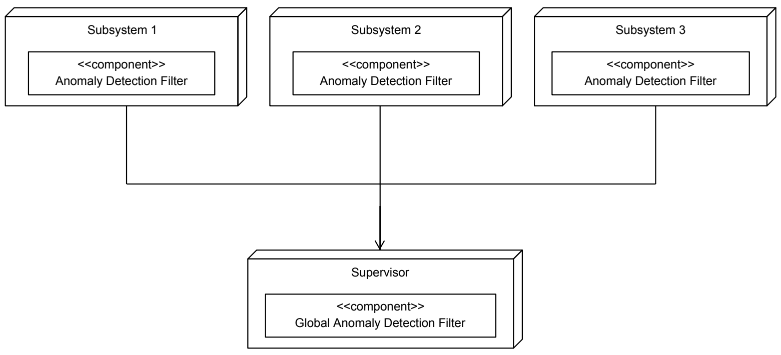


Figure 1.2. Supervised anomaly detection within a distributed system

In this hypothetical environment each subsystem uses an anomaly detection filter for early failure detection. The anomaly filter of each subsystem is connected to a supervisor node. The supervisor node could recognize local or temporal correlations of failures and initiate appropriate measures.

1.3 Challenges and Research Questions

This master's thesis aims to solve the following challenges and research questions. We present a more comprehensive and detailed view on those issues in Chapter 3 using an appropriate approach.

A first goal is to add a concurrent support to the already existing framework. This makes it possible to run filters in additional threads and forward data between filters in an asynchronous way. The result is non blocking communication and the possibility to use additional computing power of multi processor systems.

A second goal is to add a distributed part to the framework. This enables to run filters on different computers. This results in communication beyond computer limits and the possibility to use additional computing power and memory of distributed systems.

Finally we evaluate the framework with respect to performance and memory utilization. We question whether our approach is fast, suitable, and performant. We also make suggestions how the framework can be improved.

1.4 Outline of the Thesis

The rest of this document is structured as followed. Chapter 2 describes the necessary foundations for this thesis like the needed performance metrics, the Complex Event Processing (CEP), the Kieker framework, and its analysis architecture. In Chapter 3, we describe the above mentioned goals in more detail, the actual solutions, and the implementation for the listed research questions. Various experiments on blade servers are performed in Chapter 4. We evaluate the resulting framework in this chapter with regard to performance and memory consumption. We also verify which of the planned goals were achieved. In Chapter 5, we present related work to this thesis. We finally sum up the results in Chapter 6 and give recommendations for further research in the direction of this master's thesis.

Chapter 2

Foundations

In this chapter, we describe the necessary foundations for the rest of the thesis. It contains definitions of common performance metrics, which we will use for the evaluation. We also give a short introduction to Complex Event Processing (CEP). This includes its importance for modern architectures, as a subset of its methods will later be used for the distributed communication. We present furthermore the Kieker framework, with focus on the important components of the analysis part.

2.1 Performance Metrics

In this thesis we develop extensions for Kieker's framework with a main motivation of gaining additional performance. We need therefore the possibility to compare different implementations in the form of appropriate metrics. For this purpose we describe two common performance metrics for parallel systems. We also present Amdahl's Law as limiting theoretical factor.

2.1.1 Speedup

A commonly used metric is speedup, which describes the gain in performance of a task[1] when using an abstract enhancement. It can be defined as the ratio between the performance of a task using the enhancement and the performance of a task without using the enhancement. An alternative and more common used definition uses the ratio between the execution time without the enhancement and the execution time with the enhancement [Hennessy and Patterson 2007].

$$\text{Speedup} = \frac{\text{Execution time without enhancement}}{\text{Execution time with enhancement}} \tag{2.1}$$

When talking about speedup, the used enhancement is often parallelization. Some literature define the speedup in this case as the ratio between the execution runtime of the best sequential algorithm and the parallel execution time. The execution runtime of the best sequential algorithm, however, cannot always be used. It is possible that the best

[1]Speedup can be used do determine the performance gain of an algorithm, a program, or a system. We refer to all three in the following, when using the abstract term "task".

sequential algorithm is not known. It is also possible that another sequential algorithm is faster for the specific problem size, although the best sequential algorithm has a better asymptotic execution time [Rauber and Rünger 2010]. According to them, the speedup is therefore in practice often computed less with the best sequential algorithm, but rather with just a sequential version of the parallel implementation.

The execution time, however, depends on the number of used processors. We define therefore speedup for this thesis as a function in dependency of said processors.

$$\text{Speedup}(n) = \frac{\text{Execution time without parallelization}}{\text{Execution time with parallelization using } n \text{ processors}} \tag{2.2}$$

2.1.2 Efficiency

A second common metric for parallel systems is efficiency. It is closely connected to speedup, as it can be defined as the ratio between the speedup and the number of used processors (Formula 2.3) [Colombet and Desbat 1998]. Efficiency describes the average utilization of the processors [Eager et al. 1989].

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of used Processors}} \tag{2.3}$$

Analogous to speedup, we can define efficiency for this thesis as a function.

$$\text{Efficiency}(n) = \frac{\text{Speedup } (n)}{n} \tag{2.4}$$

It is, however, not possible to reduce the execution runtime of a parallel program arbitrarily [Rauber and Rünger 2010]. This fact can be gathered from Amdahl's Law. Both the speedup and the efficiency can therefore only move within certain limits.

2.1.3 Amdahl's Law

Amdahl's Law is an argument originating from Amdahl [1967]. It uses a simplified job model to make a statement about the upper bound of the speedup of a parallel program.

For said model a job consists of tasks. A fraction of this tasks ($0 < s < 1$) is sequential and cannot be enhanced. The remaining fraction ($p = 1 - s$) of the tasks, however, can run concurrently. It can be enhanced with an arbitrary number n of processors – but at most with N processors (meaning the system contains N processors) [Kleinrock and Huang 1992].

Using this model, our speedup and efficiency functions have the following forms.

$$\text{Speedup}_{s,p}(n) = \frac{s + p}{s + \frac{p}{n}} \tag{2.5}$$

$$\text{Efficiency}_{s,p}(n) = \frac{s+p}{s \cdot n + p} \tag{2.6}$$

To give value ranges for speedup and efficiency, we observe the behaviour of the functions at the value limits (e.g., when a program is strictly sequential and cannot be parallelized or when a program is perfectly parallelizable).

$$\lim_{s \to 0} \text{Speedup}_{s,p}(n) = N \tag{2.7}$$

$$\lim_{s \to 1} \text{Speedup}_{s,p}(n) = 1 \tag{2.8}$$

$$\lim_{s \to 0} \text{Efficiency}_{s,p}(n) = 1 \tag{2.9}$$

$$\lim_{s \to 1} \text{Efficiency}_{s,p}(n) = \frac{1}{N} \tag{2.10}$$

The value ranges are therefore

$$1 \leqslant \text{Speedup}(n) \leqslant N \tag{2.11}$$

and

$$\frac{1}{N} \leqslant \text{Efficiency}(n) \leqslant 1. \tag{2.12}$$

The question raises whether we can give a more precise upper bound, if we fix s and p for a given program. To answer this question, we simplify Formula 2.5 as $s + p = 1$ holds.

$$\text{Speedup}_{s,p}(n) = \frac{1}{s + \frac{p}{n}} \tag{2.13}$$

For gain optimal speedup (the function is monotonically increasing), we use N processors.

$$\text{Speedup}_{s,p}(N) = \frac{1}{s + \frac{p}{N}} \tag{2.14}$$

Formula 2.14 is directly given by Amdahl's Law [Gustafson 1988] (Note that some literatures, like Kleinrock and Huang [1992], define Amdahl's Law with an upper bound). To give the mentioned bound for the speedup, we observe again the behaviour at the value limits.

$$\lim_{N \to \infty} \text{Speedup}_{s,p}(N) = \frac{1}{s} \tag{2.15}$$

Using even an infinite number of processors, the optimal speedup is therefore limited to $1/s$ [Gustafson 1988].

Note that Amdahl's Law uses a simplified model for the speedup. For the model it is assumed that the problem size is fixed. However, Gustafson [1988] noted that "problem size scales with the number of processors". Sun and Chen [2010] showed that models with fixed time or with bounded memory scale much better than a model with fixed size, which is based on Amdahl's law. They concluded that multi core architecture is scalable.

On the one hand Amdahl's Law gives us therefore only a both theoretical and pessimistic limit for the speedup. On the other hand it also makes a statement on how we can speed up a program. To gain an optimal speedup within parallel programs, we should either try to reduce the proportion of the sequential part or to increase the proportion of the parallel part. This means especially that we should try to lower the communication overhead between processors as far as possible.

2.2 Complex Event Processing

In this section we define and describe the term Complex Event Processing (CEP). The topic is very extensive and we will actually use only a minor part of it. We will use the core of a CEP engine as a tool to implement the distributed part of Kieker. It should be noted though, that another possible approach for distributed analyses would be to convert the Kieker framework in order to fully rely on a CEP engine.

2.2.1 Introduction

According to Eckert and Bry [2009], the generic term CEP outlines various methods, techniques, and tools to process events simultaneously. An event can here be defined as "an object that is a record of an activity in a system" (see Robins [2010]). Such an event can be on the lower level, for example, sensor or market data [Eckert and Bry 2009; Buchmann and Koldehofe 2009]. The system takes a step further and uses aggregation and composition to create new complex events [Buchmann and Koldehofe 2009]. The events in a system can be classified in an event hierarchy. The basic events would be on the bottom and the more complex and composed events further up. The events in the lower layers would be closer to the hardware level. However, the events in the upper layers would be closer to the business level [Robins 2010; Luckham and Frasca 1998]. A simplified illustration of the event hierarchy can be seen in Figure 2.1. The hierarchy is not limited to two abstract layers. There are also additional components between the layers to process and generate the complex events.

The resulting (complex) events can be used to recognize correlations, which would be otherwise not visible. As an example (adapted from [Eckert and Bry 2009]) consider the two basic events *Temperature* and *Smoke*, sent by two independent sensors. An event

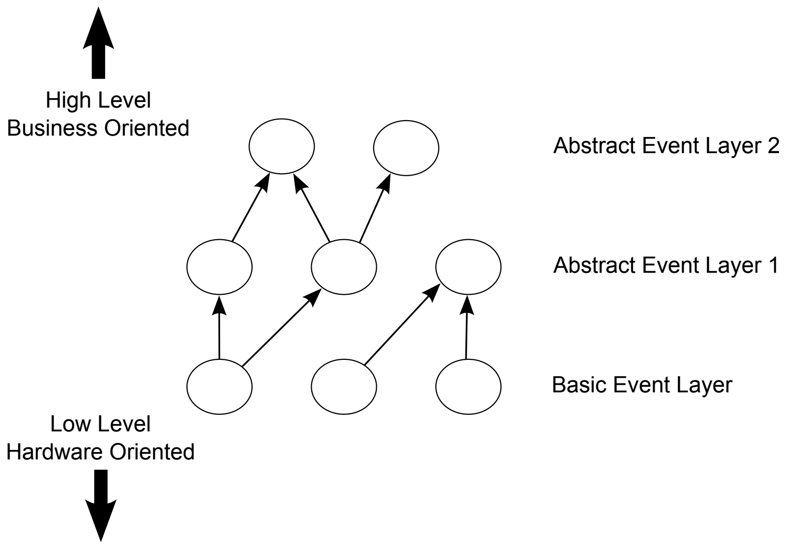


Figure 2.1. Simplified event hierarchy within CEP (Adapted from Luckham and Frasca [1998])

processor could find a temporal correlation and create a new complex event *Fire*. An event driven system could listen to those events and trigger automatically actions or notifications [Eckert and Bry 2009]. One could imagine an emergency shutdown of a system, the notification of administrators in case of application response time anomalies, or, in case of the fire example, the activation of the fire sprinkling system.

2.2.2 Processing Events

One mentioned possibility to process events is the use of event query languages. They can be used to recognize and specify complex events. In this case the event patterns are already known. The difference between event queries and usual database queries is shown in Figure 2.2. The database queries are processed and replied to immediately. However, an event query is continuously evaluated in the CEP engine. The used language has to be expressive enough to support, inter alia, composition, accumulation, and temporal correlations. Furthermore, defined rules can be deductive (create new events) or reactive (react on events) [Eckert and Bry 2009].

A second possibility to process events is to recognize so far unknown patterns within the given events. This approach uses artificial learning and data mining and can therefore be classified in the domain of artificial intelligence. However, this approach is not as explored as the former [Eckert and Bry 2009].

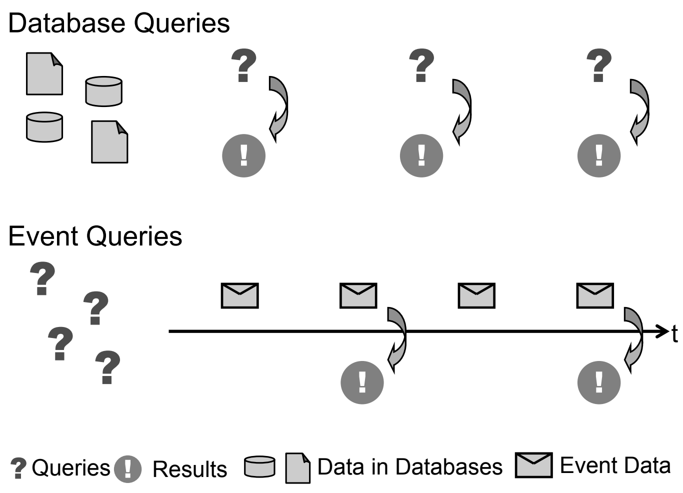


Figure 2.2. Comparison of database and event queries (Adapted from Eckert and Bry [2009])

2.2.3 Impact on Modern Architectures

Although still a relatively new separate domain in industry [Eckert and Bry 2009] , CEP can be an important tool to design high scalable and dynamic systems. Due to the nature of CEP, the event providers do not have to know about the event receivers and vice versa [Buchmann and Koldehofe 2009]. This approach leads therefore to weak coupling between the single components. According to Buchmann and Koldehofe [2009], CEP is more and more used during "the development of monitoring and reactive applications". They mention, that "CEP already plays an important role in many application areas like logistics, energy management, finance, or manufacturing processes". Eckert and Bry [2009] mention business activity monitoring, sensor networks, and market data as important scope of application.

2.2.4 Active Research Topics

Currently there is still a strong need for the research of query languages with respect to actual projects. Although CEP is, for example, highly utilised by algorithmic trading, only few information are actually available. Most of the projects in algorithmic trading are strictly confidential. It is also difficult to compare different CEP approaches and query languages, due to the lack of suitable benchmarks. Topical research domains are also inaccurate events and data mining [Eckert and Bry 2009].

Buchmann and Koldehofe [2009] notice also, that another important topic would be security in event based systems, which would be only addressed by few researchers. They also remarked that most of the solutions used in industrial scopes still rely on centralized processing, which stands in contrast to the mentioned benefits.

In summary, we can say that CEP can be a powerful framework to design robust and distributed systems. The usage of events leads naturally to loose coupling between the components. CEP has, however, also drawbacks due to the fact that there are still research topics, which are not fully understood yet.

2.3 The Kieker Framework

In this section, we present an overview of the important parts of Kieker. The information in this section are, if not otherwise noted, from Kieker Project [2013b]. As we describe the technical architecture of Kieker, we also use overlapping information and excerpts from the source code in the Kieker repository.

2.3.1 Introduction

The Kieker software is both a framework and a collection of tools, developed at the Software Engineering Group at the Kiel University and the University of Stuttgart. It offers the possibility to dynamically monitor and analyse, amongst others, Java, Visual Basic 6, COBOL, and .Net based applications [Kieker Project 2013b; c]. It emerged from a diploma thesis (Focke [2006]) in 2006. Since first published on Sourceforge in 2008, Kieker is constantly being developed and refined. The tool is used not only in academical, but also in industrial scopes [Kieker Project 2013c].

Kieker's architecture is shown in Figure 2.3. It consists of two parts. One for the monitoring and one for the analysis. The monitoring part can be divided in three types of components: Monitoring probes, monitoring controllers, and monitoring writers. The analysis part can be divided in: Analysis components, analysis controllers, and monitoring readers. Kieker uses monitoring records to exchange data between the two parts.

We briefly explain both parts in this section. As the analysis architecture, however, is more important for the rest of this thesis, it is explained in greater depth in the following Section 2.3.2.

Monitoring To monitor running programs, Kieker provides various possibilities to add the necessary monitoring code into the source or binary code. This can be done, inter alia, using manual probes, samplers based on Sigar [VMware 2013], or Aspect Oriented Programming (AOP) based on *AspectJ* [Eclipse Foundation 2013a]. The monitoring life cycle can be seen in Figure 2.4a. So called probes are responsible for collecting monitoring data, which can be, for example, method calls or simply the CPU utilization. After the data has been collected, it is stored into records (Step 1.1). Those records are then sent to the monitoring controller (Step 1.2). The controller relays it to a configured writer (Step 2). Said writer can write the record on the file system, in a database, or sent it to

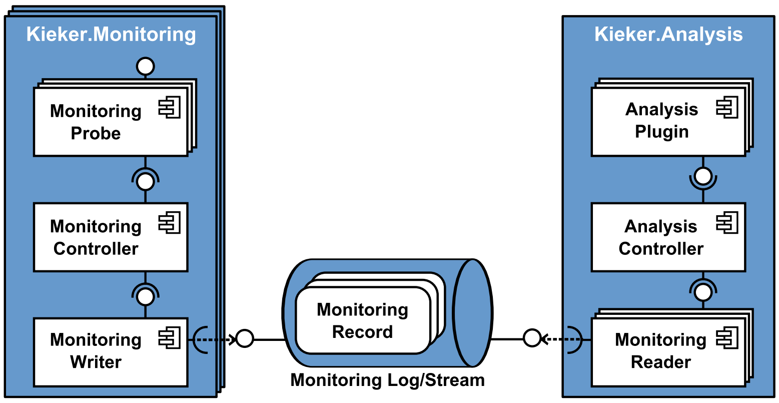
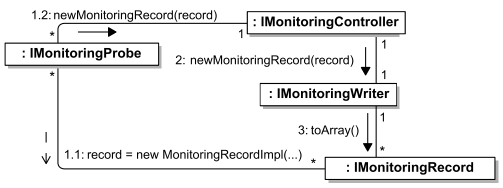
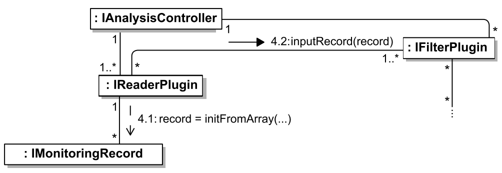


Figure 2.3. Overview of Kieker's architecture (Adapted from Kieker Project [2013b])


1.2: newMonitoringRecord(record)
: IMonitoringController
: IMonitoringProbe
2: newMonitoringRecord(record)
: IMonitoringWriter
3: toArray()
1.1: record = new MonitoringRecordImpl(...)
: IMonitoringRecord

(a) Monitoring Life Cycle



(b) Analysis Life Cycle

Figure 2.4. Overview of the life cycle within Kieker's architecture [Kieker Project 2013b]

other systems via network (e.g., with Java Message Service (JMS) or Java Management Extensions (JMX)).

Analysis The analysis life cycle can be seen in Figure 2.4b. The analysis part extracts the monitoring data from the records (Step 4.1) using either a suitable file system or network reader (based on JMS or JMX as well). Alternatively a reader can also generate records like a load driver. The collected data is sent to filters from a prepared analysis (Step 4.2). During this process, the analysis can, for example, produce dependency graphs or reconstruct the architecture of the monitored application. A more detailed view on the analysis is shown in the following section.

2.3.2 The Analysis Architecture

Most of the analysis part of Kieker is implemented as a typical pipes and filters architecture as described by Hohpe and Woolf [2004]. It bases on analysis components, which are elements called readers, filters, and repositories. They are supervised by an analysis controller. This controller is, amongst others, responsible to manage the life cycle and

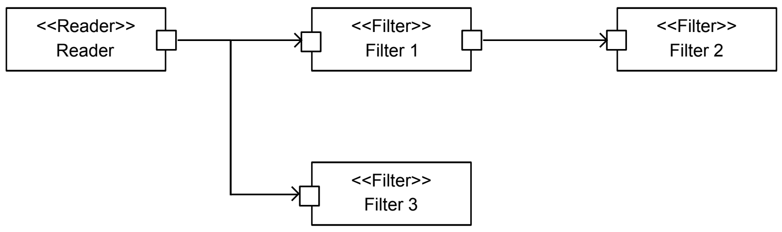


Figure 2.5. Example pipes and filters configuration

the connections of the components. It also provides methods to start the analysis or to shut it down.

The analysis components represent the filter part of the pipes and filters architecture, while the connections are synonymous with the pipes. To avoid confusion, we will use the term "filter" in the following, if not otherwise noted, to refer to Kieker's filter analysis component though.

An analysis within Kieker can contain multiple readers. Those are the components, which are responsible for reading, receiving, or generating monitoring records. The records are then sent to connected filters. Filters can work on the given data (e.g., they can filter or enrich the incoming data) and sent the results to their successors. Both readers and filters can be connected with repositories. Repositories are in Kieker simply implemented as shared data storages.

To explain the data flow between the components in more detail assume the situation in Figure 2.5. The record read by the reader is sent to Filter 1 and Filter 3 in order to be processed. The resulting data (in form of an object) from Filter 1 is then sent to Filter 2. An UML sequence diagram showing the resulting forwarding sequence with one record being send can be seen in Figure 2.6. The trace is marked bold. It can be seen that the reader reads the record and sends it with a synchronous call 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 as well 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.

One can see that Filter 1 and Filter 3 are independent from each other, but do not work in parallel. Instead the record is processed sequentially. Filter 3 has to wait until Filter 1 (and all its successors) processed the record, before it can process the record itself.

In the following we explain and describe some important aspects of the analysis part in more detail.

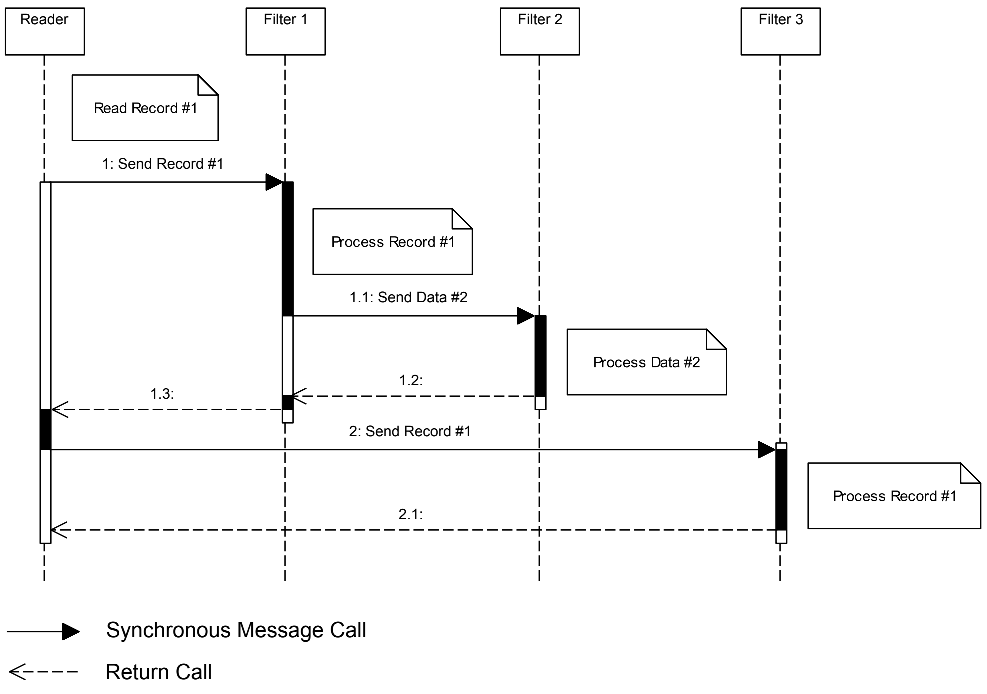


Figure 2.6. Data forwarding sequence for the Kieker analysis of Figure 2.5

Analysis Controller The analysis controller is at the core of an analysis, as it controls and manages the life cycle of all other components. All further analysis components are directly registered with the controller during creation. The controller makes sure that the components are correctly initialized during start and it makes also sure that an analysis is terminated in a controlled way. Furthermore, it verifies the correctness of connections between the components.

For a single analysis, only one analysis controller is necessary and allowed. Using a meta model, which we explain later, it is possible to store and load the whole configuration of an controller.

Reader Readers are components without input ports but (usually) with at least one output port. Those components are responsible for reading or creating monitoring records from a defined source. The readers deliver the records to connected repositories or filters. As each reader is started in an own thread and one analysis can contain multiple readers, it is already possible to use concurrency within an analysis. However, it is, as mentioned before, not fully supported by the architecture. It is also not possible to start filters in own threads.

As the readers are the driving components within analyses, they also determine the point of shutdown. An analysis is terminated by the analysis controller once all readers are finished.

Currently available readers can read (compressed) monitoring data from the file system and databases. They can also receive records via JMS, JMX, and in-memory-pipes. Some readers can furthermore simply read and deliver the current system time in periodic intervals.

Filter Filters are components within the analysis performing most of the work. They can have multiple input, repository, and output ports. They can receive data, perform any processing, and can deliver new output to connected filters. An exemplary filter would be a `CountingFilter`, which increments an internal counter when new records are received. It relays the received data, before delivering the current counter value to another output port.

Repository The repositories within the architecture are components which can store data shared by multiple components. Such a repository could, for example, contain a reconstructed system model, which is enriched by different filters and stored on the file system after the analysis. Although readers and filters are connected with repositories using so called repository ports, the interface to these repositories is not specified. Unlike filters and readers they do not receive their data in a specified way.

Repositories are rather unusual components in a pipes and filters architecture. Filters should be able to process independent from each other.

The Analysis Meta Model The analysis meta model is an Eclipse Modeling Framework (EMF) [Eclipse Foundation 2013b] based meta model with two main purposes. On the one hand it describes the existing analysis components and their relationship. However, it is not used for the actual implementation of the analysis components. On the other hand it is used to store and load configurations for the analysis controller. During the load and store processes, the model instances and their properties are mapped to and from the actual Java classes inside Kieker.

Ports and Connections Kieker provides three types of ports. Input, output, and repository ports. The first two can be connected with each other, which means, that the input port registers as an observer (as described by Gamma et al. [2009]) for the output port. When a plugin sends data to an output port, the data is checked and then sent to all registered input ports. However, the repository ports are not using an observer pattern. They are rather establishing a dynamic connection between plugins and repositories. Data is not sent in a specified way from and to the connected plugins.

All of the pipes between the filters are implicitly existent due to the implementation. They are not implemented as first-class entities. The declaration of ports is done using Java 1.5 style annotations at the method or the class respectively, as can be seen in

Listing 2.1. The declaration of two output ports is shown in the lines 3–4. Each output port has a name and several event types. The event types are the potential types of outgoing data. They are used by the analysis controller to validate the compatibility of ports. Line 13 shows how a component can send data through an output port.

The input ports are annotated methods each with one parameter. This can be seen in the lines 8–9. Analogous to output ports, input ports have names and event types.

Listing 2.1. Simplified excerpt from `CountingFilter`

```
 1  @Plugin(
 2    outputPorts = {
 3      @OutputPort(name = "relayedEvents", eventTypes = { Object.class }),
 4      @OutputPort(name = "currentEventCount", eventTypes = { Long.class })
 5  })
 6  public final class CountingFilter extends AbstractFilterPlugin {
 7
 8    @InputPort(name = "inputEvents", eventTypes = { Object.class })
 9    public final void inputEvent(final Object event) {
10      // Do something with the incoming event
11      ...
12      // Relay incoming data to following filters
13      super.deliver("relayedEvents", event);
14    }
15    ...
16  }
```

A simplified excerpt presenting the repository ports can be seen in Listing 2.2. The repository port is defined in the lines 3–4. Repository ports have names as well, but no event types. Instead they have a field determining the allowed type of connected repository.

Listing 2.2. Simplified excerpt from `TraceReconstructionFilter`

```
1  @Plugin(...,
2    repositoryPorts = {
3      @RepositoryPort(
4        name = "systemModelRepository", repositoryType = SystemModelRepository.class)
5  })
6  public class TraceReconstructionFilter extends AbstractTraceProcessingFilter {
```

In the following Listing 2.3 we show how the analysis controller is used to connect plugins with each other. In line 6 two filters are connected. The connecting of a filter with a repository can be seen in line 8. Connecting two filters requires the names of the output port and the input ports which should be connected. Connecting a filter with a repository requires only the name of the repository port to connect.

Listing 2.3. Simplified example presenting how to connect components

```java
// Create the analysis controller
final IAnalysisController ac = new AnalysisController();
// Create and configure the filters
...
// Connect two filters with each other
ac.connect(traceReconstructionFilter, "executionTraces", countingFilter, "inputEvents");
// Connect a filter with a repository
ac.connect(traceReconstructionFilter, "systemModelRepository", systemModelRepository);
```

Plugin and Analysis Configuration In order to support the storing and loading of analysis configurations, it is necessary to supply the plugins with their configuration in a specified way. While the available property keys (including their default values) are introduced using annotations, as it can be seen in Listing 2.4, Kieker uses a `Configuration` class to deliver the actual configuration (e.g. loaded from a configuration file).

Listing 2.4. Simplified excerpt from `TraceReconstructionFilter`

```java
@Plugin(...,
  configuration = {
    @Property(name = "timeunit", defaultValue = "NANOSECONDS"),
    ...
})
public class TraceReconstructionFilter extends AbstractTraceProcessingFilter {
```

In Listing 2.5 it can be seen how a configuration object is used to create and configure a filter. The shown filter constructor in Line 9 has to be provided by every analysis component, as it is internally used during the configuration loading process.

Listing 2.5. Simplified example presenting how to configure components

```java
1  // Create the analysis controller
2  final IAnalysisController ac = new AnalysisController();
3
4  // Prepare the configuration for the filter
5  final Configuration configuration = new Configuration();
6  configuration.setProperty("timeunit", "SECONDS");
7
8  // Create the filter using the analysis controller and the prepared configuration
9  final TraceReconstructionFilter filter = new TraceReconstructionFilter(configuration, ac);
```

Termination Similar to the forwarding sequence, the termination sequence is strictly sequential. The analysis controllers waits until all reader threads have finished reading, before the termination sequence starts. Alternatively the analysis can be terminated by calling the termination method of the controller.

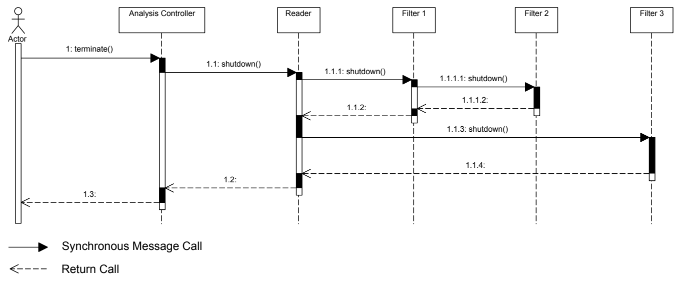


Figure 2.7. Simplified termination sequence for the Kieker analysis of Figure 2.5

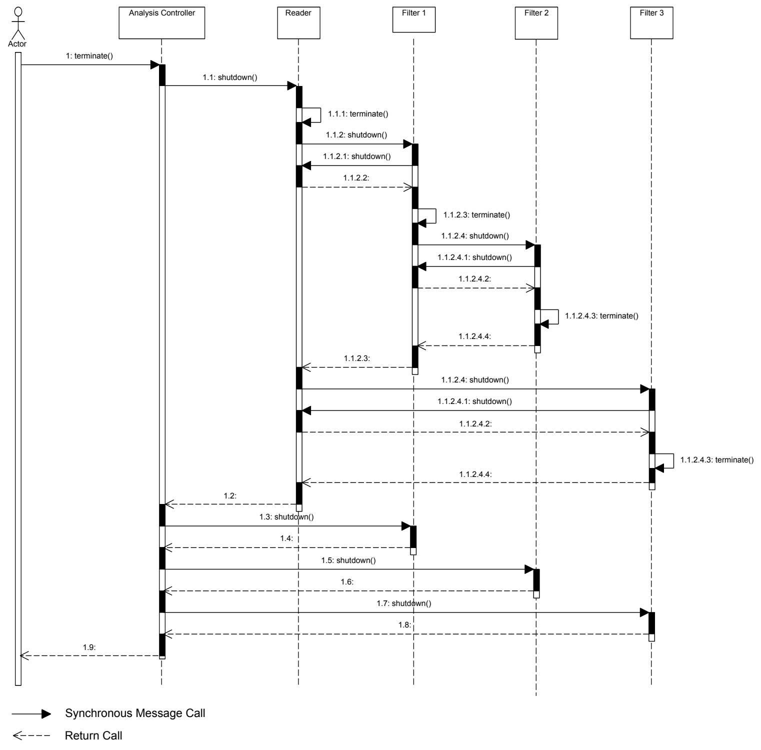


Figure 2.8. Full termination sequence for the Kieker analysis of Figure 2.5

The controller calls the termination method of all readers and all further filters. Each of this termination calls makes sure that all incoming and all outgoing plugins (with respect to the connected ports) are being terminated. Assume again the situation shown in Figure 2.5. The resulting termination sequence can be seen in Figure 2.7. The termination sequence is simplified, as each of the filters would also try to terminate its predecessors. The full termination sequence with all method calls can be seen in Figure 2.8.

Chapter 3

Development of a Concurrent and Distributed Analysis Framework for Kieker

In this chapter, we describe our approach and implementation. As a suitable process model we use GQM (as described by Basili et al. [1994]) to define the research questions from Section 1.3 in detail. After the definition of the goals, we present our approaches to the concurrent and the distributed parts of the new framework.

3.1 Definition of the Research Questions

Goal Question Metric (GQM) is a measurement mechanism to create measurable goals for a project. The concept of the approach is the specifications of goals. Various questions characterize how the goals are achieved and which particular qualities are important for the goals. Each of the questions is then associated with multiple metrics. These metrics can either be objective or subjective and should be able to answer the questions quantitatively [Basili et al. 1994]. The below listed goals and questions influence the actual implementation of the framework parts. We use the listed metrics furthermore to verify our results in Chapter 4.

`G1` Develop Kieker to support concurrent analyses

The first goal, which we designate as `G1`, is to extend the present pipes and filters analysis architecture of Kieker in order to support concurrent analyses.

The main question for this goal is whether the modifications we develop, allow to use concurrent analyses (`Q1.1`). We validate this by checking the existence of this part in the framework (`M1.1.1`). A second metric is a subjective evaluation, which checks also the degree of usability (`M1.1.2`).

Another important question is whether we improve the performance of an analysis with our modifications (`Q1.2`). This can be checked with the speedup (`M1.2.1`), the throughput (`M1.2.2`), and the efficiency (`M1.2.3`) of an analysis. We observe furthermore the memory consumption of our implementation (`M1.2.4`).

Goal	`G1`	Develop Kieker to support concurrent analyses
Question	`Q1.1`	Do the modifications allow to use concurrent analyses?
Metrics	`M1.1.1`	Existence of concurrency in the framework
	`M1.1.2`	Subjective evaluation by the author
Question	`Q1.2`	Do the modifications improve the performance of an analysis?
Metrics	`M1.2.1`	Speedup $= \frac{\text{Present execution time}}{\text{New execution time}}$
	`M1.2.2`	Throughput $= \frac{\text{Records}}{\text{Second}}$
	`M1.2.3`	Efficiency $= \frac{\text{Speedup}}{\text{Used Processors}}$
	`M1.2.4`	Maximum memory consumption of the program
Question	`Q1.3`	How significant are the modifications with respect to the API?
Metrics	`M1.3.1`	Number of classes, that are part of the public API, and were modified
	`M1.3.2`	Number of methods, that are part of the public API, and were modified
	`M1.3.3`	Subjective evaluation by the author
Question	`Q1.4`	Is it possible to perform a clean shutdown of a running analysis?
Metrics	`M1.4.1`	Existence of termination method
	`M1.4.2`	Subjective evaluation by the author

Table 3.1. Definition of GQM goal `G1`

An important framework constraint is to modify the present Application Programming Interface (API) as little as possible. Question `Q1.3` is therefore how significant our modifications are. This is checked with the number of API classes (`M1.3.1`) and with the number of API methods (`M1.3.2`) that were modified. As not all modifications can be expressed in numbers, we also perform a subjective evaluation (`M1.3.3`).

The last question is whether it is possible to perform a clean shutdown of an analysis (`Q1.4`). The reason for this question is derived from the architecture of Kieker as described in Section 2.3. The current implementation assumes that the analysis can be terminated once all readers have been terminated. If some parts of the analysis run concurrently and independently from the readers, this is no longer the case and therefore another challenge. We van verify this by checking the existence of a termination method (`M1.4.1`). We measure the effort with a subjective evaluation (`M1.4.2`) at this point as well.

A breakdown of the mentioned goal, questions and metrics is listed in Table 3.1.

Goal	G2	Develop Kieker to support distributed analyses
Question	Q2.1	Do the modifications allow to use distributed analyses?
Metrics	M2.1.1	Existence of distribution in the framework
	M2.1.2	Possibility to distribute repositories
	M2.1.3	Subjective evaluation by the author
Question	Q2.2	Do the modifications improve the performance of an analysis?
Metrics	M2.2.1	Speedup $= \frac{\text{Present execution time}}{\text{New execution time}}$
	M2.2.2	Throughput $= \frac{\text{Records}}{\text{Second}}$
	M2.2.3	Efficiency $= \frac{\text{Speedup}}{\text{Used Processors}}$
	M2.2.4	Maximum memory consumption of the program
Question	Q2.3	How significant are the modifications with respect to the API?
Metrics	M2.3.1	Number of classes, that are part of the public API, and were modified
	M2.3.2	Number of methods, that are part of the public API, and were modified
	M2.3.3	Subjective evaluation by the author
Question	Q2.4	Is it possible to perform a clean shutdown of a running analysis?
Metrics	M2.4.1	Existence of termination method
	M2.4.2	Subjective evaluation by the author

Table 3.2. Definition of GQM goal G2

G2 Develop Kieker to support distributed analyses

The second goal G2 is to extend the Kieker framework in order to support distributed running analyses.

Similar to G1 the main question is whether our modifications allow to distribute an analysis (Q2.1). We check the existence (M2.1.1) of this part and use a subjective evaluation (M2.1.3) as well. However, as explained in Section 2.3, the repositories, unlike the remaining plugins, do not have a defined interface or generic input ports. It will be therefore more difficult to access the repositories in a distributed analysis. Thus we also check whether the distribution of repositories is possible (M2.1.2).

The second question (Q2.2) is the same as in G1 and concerns the actual performance improvement. We use the same metrics (M2.2.1, M2.2.2, M2.2.3, M2.2.4).

The third question concerns the API modifications as well (Q2.3) and will be measured

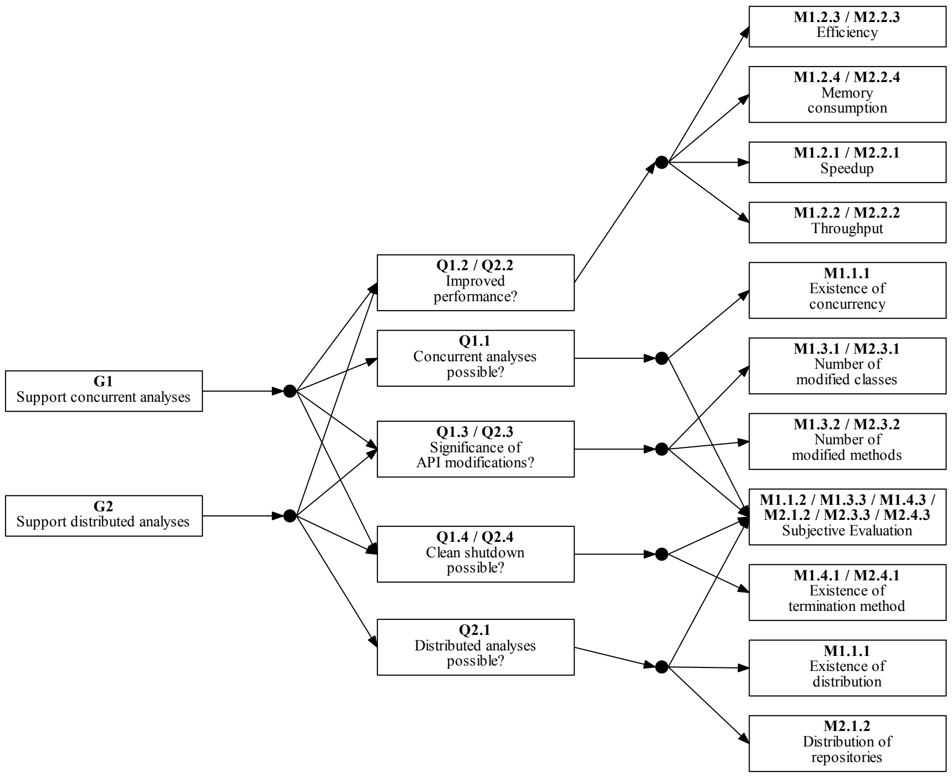

M1.2.3 / M2.2.3
Efficiency
M1.2.4 / M2.2.4
Memory consumption
M1.2.1 / M2.2.1
Speedup
M1.2.2 / M2.2.2
Throughput
M1.1.1
Existence of concurrency
M1.3.1 / M2.3.1
Number of modified classes
M1.3.2 / M2.3.2
Number of modified methods
M1.1.2 / M1.3.3 / M1.4.3 / M2.1.2 / M2.3.3 / M2.4.3
Subjective Evaluation
M1.4.1 / M2.4.1
Existence of termination method
M1.1.1
Existence of distribution
M2.1.2
Distribution of repositories
Q1.2 / Q2.2
Improved performance?
Q1.1
Concurrent analyses possible?
Q1.3 / Q2.3
Significance of API modifications?
Q1.4 / Q2.4
Clean shutdown possible?
Q2.1
Distributed analyses possible?
G1
Support concurrent analyses
G2
Support distributed analyses

Figure 3.1. Visual representation of the GQM goals

using the same metrics as in G1 (M2.3.1, M2.3.2, M2.3.3).

The question, how a running analysis will be terminated (Q2.4) is the same as in G1. We use also the same metrics (M2.4.1, M2.4.2).

A breakdown of the mentioned goal, questions and metrics is listed in Table 3.2. A visual representation can be seen in Figure 3.1.

3.2 Implementation Details

To present the actual implementation details, we use vastly simplified code listings in the following sections. Although it is still Java code, we renounce unimportant code parts. The complete code can be found in the Kieker repository on the attached DVD.

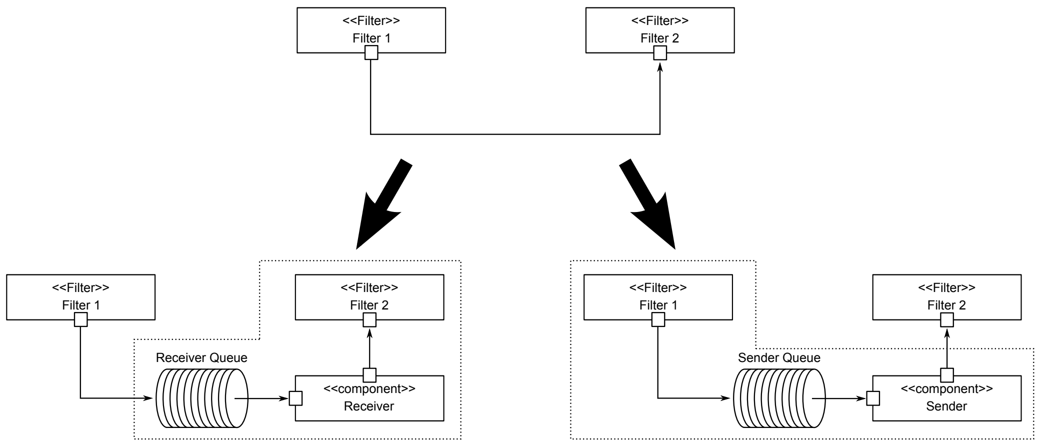


Figure 3.2. The basic concurrent design

3.2.1 Concurrent Part

Basic Concurrent Design For this part of the thesis, we want to enable concurrent plugins in the analysis. To achieve this, it is necessary that plugins can send their data asynchronously to further plugins. A possible approach is to use a First In, First Out (FIFO) buffer between the components. Instead of sending data directly to its neighbours, a plugin writes the data into the buffers. The receiving plugin can read the data later. This results in a temporal decoupling of the plugins. A filter does not have to wait until its neighbours have processed given data. The basic design, which we use to add concurrency to the existing pipes and filters architecture, can be seen in Figure 3.2. The connection between two filters is enriched by two sub-components. The first part is a queue, which stores the data to be received or sent. The second part is responsible for the actual sending or receiving of data. It is implemented as a concurrent thread.

The parts can be slightly different, depending on whether the input or the output port of a filter is set into an asynchronous mode. In the case of an asynchronous input port, the thread reads blocking from the queue and sends it then synchronously to its corresponding filter. In the case of an asynchronous output port, the threads reads the data to be sent from the queue and delivers it synchronously to the connected input ports.

Data Forwarding As mentioned in Chapter 2, the connections in the current implementation of Kieker are not first class entities. In order to enrich them, it is necessary to intercept the sent data. This will be done within the `deliver` method of the `AbstractPlugin` class. This method is responsible for sending data via an output port to other input ports. The API signature of this method is `protected boolean deliver(String outputPortName, Object data)`. The first argument is the name of the output port. The second argument is the data to be sent. In order to decide whether the sent data has to be intercepted by the concurrent part or whether it is sent directly by a sending thread, we add another flag. The signature of this method is `private boolean deliver(String outputPortName, Object data, boolean checkForAsynchronousMode)`. The old method simply wraps the new `deliver` method with `checkForAsynchronousMode` set to true. This avoids modifications at the API.

The new method has now to check the data forwarding from the output and to the input ports. The resulting code can be seen in Listing 3.1. A check (lines 5–8) tests whether the output port is asynchronous. If this is the case, the data is put into the queue (line 6) and the method returns immediately. However, it is assumed that the delivering does not fail (line 7). The check for asynchronous input ports is done in the lines 11–19. For each of the connected ports it is checked whether it is asynchronous (lines 12–14). If this is the case, the data is put into the receiving queue (line 15) and the method processes the next port.

Listing 3.1. Excerpt from `AbstractPlugin` showing the new delivering

```
 1 private boolean deliver(String outputPortName, Object data,
 2     boolean checkForAsynchronousMode) {
 3   ...
 4   // Check whether the output port is asynchronous
 5   if (checkForAsynchronousMode && this.sendingQueues.containsKey(outputPortName)) {
 6     this.sendingQueues.get(outputPortName).add(data);
 7     return true;
 8   }
 9   ...
10   // Check whether the input port is asynchronous
11   for (PluginInputPortReference pluginInputPortReference : registeredMethodsOfPort) {
12     final Queue<Object> receivingQueue = receivingPlugin.receivingQueues.get(
13       pluginInputPortReference.getInputPortName());
14     if (receivingQueue != null) {
15       receivingQueue.add(data);
16       break;
17     }
18     ...
19   }
20   ...
21 }
```

Sender and Receiver Threads The actual implementation of the sender thread within the `AbstractPlugin` class is shown in Listing 3.2. As each sender thread belongs to an output port, the name and the queue of the port have to be stored (lines 5–6 and 8–11). A simple termination functionality is provided using a termination token (line 3) and a public method (lines 30–32). The important body of the thread consists of an infinite loop (line 15), which is only terminated, if the thread reads a termination token from the queue. The threads reads an element from the queue blocking (line 17) and delivers it via the given output port (line 23). The additional boolean flag makes sure, that the data is actually send, instead of stored in the output queue. Before terminating, the thread makes sure that the analysis controller is informed correctly about the shutdown of the corresponding filter (line 27). The meaning of this method will be shown in the shutdown part of this section.

Listing 3.2. Excerpt from `AbstractPlugin` showing the sending thread

```
 1  private class SendingThread extends Thread {
 2
 3    private final Object TERMINATION_TOKEN = new Object();
 4
 5    private final BlockingQueue<Object> queue;
 6    private final String outputPortName;
 7
 8    public SendingThread(final BlockingQueue<Object> queue, final String outputPortName) {
 9      this.queue = queue;
10      this.outputPortName = outputPortName;
11    }
12
13    @Override
14    public void run() {
15      while (true) {
16        // Get the next element from the queue
17        final Object data = this.queue.take();
18        if (data == this.TERMINATION_TOKEN) {
19          // If the element is the termination token, we interrupt the loop
20          break;
21        } else {
22          // Otherwise we deliver the element
23          AbstractPlugin.this.deliver(this.outputPortName, data, false);
24        }
25      }
26
27      AbstractPlugin.this.shutdownAsynchronousConnection();
28    }
29
30    public void terminate() {
31      this.queue.add(this.TERMINATION_TOKEN);
32    }
33  }
```

The receiver thread works similar and is shown in Listing 3.3. Instead of an output port name, the receiving thread needs the method, which belongs to the input port (line 6), as it is actually called. The rest of the thread works like the sender thread, except that the data is not sent using the internal Kieker mechanism, but rather by calling the method (line 26). The meaning of the meta signal processing (line 23) will be explained in the shutdown part of this section.

Listing 3.3. Excerpt from `AbstractPlugin` showing the receiving thread

```java
 1  private class ReceivingThread extends Thread {
 2
 3    private final Object TERMINATION_TOKEN = new Object();
 4
 5    private final BlockingQueue<Object> queue;
 6    private final Method method;
 7
 8    public ReceivingThread(final BlockingQueue<Object> queue, final Method method) {
 9      this.queue = queue;
10      this.method = method;
11    }
12
13    @Override
14    public void run() {
15      while (true) {
16        final Object data = this.queue.take();
17        // Get the next element from the queue
18        if (data == this.TERMINATION_TOKEN) {
19          // If the element is the termination token, we interrupt the loop
20          break;
21        } else if (data instanceof MetaSignal) {
22          // A meta signal has to be processed directly
23          AbstractPlugin.this.processAndDelayMetaSignal((MetaSignal) data);
24        } else {
25          // Otherwise we call the corresponding input port method
26          this.method.invoke(AbstractPlugin.this, data);
27        }
28      }
29
30      AbstractPlugin.this.shutdownAsynchronousConnection();
31    }
32    ...
33  }
```

Port Configuration We add the setting, which ports are configured to be asynchronous, to the `Configuration` objects for the plugins. We add two additional properties. The result can be seen in Listing 3.4. The new properties are in the lines 1–2. The name of the asynchronous ports are read from the configuration in the constructor (lines 10–11).

The lines 12 and 13 are just necessary for a later binary search. During the verification of the ports, the constructor makes sure that the listed ports are set into the asynchronous mode (lines 16 to 20 and 23 to 27). The called methods are responsible for adding the FIFO buffers and threads (this is exemplaric shown in the lines 33 to 37).

Listing 3.4. Excerpt from `AbstractPlugin` showing the additional configuration

```
 1 public static final String CONFIG_ASYNC_INPUT_PORTS = "async-input-ports";
 2 public static final String CONFIG_ASYNC_OUTPUT_PORTS = "async-output-ports";
 3 
 4 ...
 5 
 6 public AbstractPlugin(final Configuration config, final IProjectContext projectContext) {
 7   super(config, projectContext);
 8 
 9   // Get the names of the asynchronous ports and prepare them for a binary search
10   String[] asyncInputPorts = config.getStringArrayProperty(CONFIG_ASYNC_INPUT_PORTS);
11   String[] asyncOutputPorts = config.getStringArrayProperty(CONFIG_ASYNC_OUTPUT_PORTS);
12   Arrays.sort(asyncInputPorts);
13   Arrays.sort(asyncOutputPorts);
14   ...
15   // Set the output ports into an asynchronous mode
16   for (final OutputPort outputPort : annotation.outputPorts()) {
17     if (Arrays.binarySearch(asyncOutputPorts, outputPort.name()) >= 0) {
18       this.setOutputPortToAsynchronousMode(outputPort.name());
19     }
20   }
21   ...
22   // Set the input ports into an asynchronous mode
23   for (final Method method : this.getClass().getMethods()) {
24     final InputPort inputPort = method.getAnnotation(InputPort.class);
25     if (Arrays.binarySearch(asyncInputPorts, inputPort.name()) >= 0) {
26       this.setInputPortToAsynchronousMode(inputPort.name(), method);
27     }
28     ...
29   }
30   ...
31 }
32 
33 private void setOutputPortToAsynchronousMode(final String outputPortName) {
34   final BlockingQueue<Object> newQueue = new LinkedBlockingQueue<Object>();
35   this.sendingQueues.put(outputPortName, newQueue);
36   this.sendingThreads.add(new SendingThread(newQueue, outputPortName));
37 }
38 ...
```

Analysis Start As the analysis controller calls the `start` method of each plugins during initialization, we use this place to start the additional threads. This avoids further API modifications. The initialization can be seen in Listing 3.5.

Listing 3.5. Initialization of the asynchronous ports

```java
public final boolean start() {
  ...
  this.initAsynchronousPorts();
  return this.init();
}

private void initAsynchronousPorts() {
  for (final Thread t : this.sendingThreads) {
    t.start();
  }
  for (final Thread t : this.receivingThreads) {
    t.start();
  }
}
```

Analysis Shutdown The current termination sequence has not been developed for concurrent or even distributed analyses. It is therefore necessary to modify it. The basic idea consists of meta signals, which are sent through the regular pipes. They are intercepted and processed by the framework. Furthermore we reduce the influence of the analysis controller on the termination sequence. Instead of initiating the shutdown, the controller waits for all plugins to be finished.

This autonomous system makes use of the fact that the readers are the driving elements within an analysis. The readers send an initialization signal through the pipes before they start reading. The processing of the meta signals takes place in the `processAndDelayMetaSignal` method. It is either called within the modified `deliver` method or within the receiver thread. The filters remember the number of received signals. After the reading, each reader sends a termination signal. Once a filter has received all termination signals, it starts the shutdown method and notifies the analysis controller about its termination. An exception from this approach is necessary for filters with asynchronous ports. In this case the shutdown method can not wait for the threads to finish, as this could block the whole system. Instead the threads are responsible for the final notification of the analysis controller (this takes place in the mentioned `shutdownAsynchronousConnection` method).

This shutdown procedure does work for most cases. It has, however, various drawbacks. The main problem is the autonomy of the system. It is currently not possible to terminate it from the outside. Another problem is, that error cases are currently not handled correctly. A third problem affects filters without input ports. Those filters are rather unusual, but they do exist within Kieker. One of these filters, for example, is responsible for writing the system model to the file system after an analysis. In the original framework these filters are terminated after the reader and their successors have been terminated (This is shown in Section 2.3.2). In the new termination sequence such filters would block the analysis.

In Listing 3.6 the important modifications of the analysis controller are shown. After starting the readers (line 3), the controller simply waits for all readers and filters to finish (line 5–6).

Listing 3.6. Excerpt from `AnalysisController` showing the termination modifications

```java
1  public final void run() throws IllegalStateException, AnalysisConfigurationException {
2    ...
3    initializationLatch.countDown();
4
5    readerLatch.await();
6    filterLatch.await();
7
8    LOG.info("Analysis terminated");
9  }
```

In Listing 3.7 we show the modifications within the readers. Instead of simply calling the `read` method, implemented by the actual reader, the controller calls the new `startRead` method. Said method sends an initialization signal (line 2), executes the usual reading (line 3), and performs the usual shutdown (line 4). Finally the reader sends a termination signal (line 5).

Listing 3.7. Excerpt from `AbstractReaderPlugin` showing the new `startRead` method

```java
1  public final void startRead() {
2    beforeRead();
3    read();
4    shutdown(false);
5    afterRead();
6  }
7
8  private void beforeRead() {
9    sendMetaSignal(new InitializationSignal());
10 }
11
12 private void afterRead() {
13   sendMetaSignal(new TerminationSignal(false));
14 }
```

In Listing 3.8 we show the important modifications of the class `AbstractPlugin`. Within the `deliver` method we have to ensure that meta signals are always send through the pipes (line 6) independent of the types of the ports. The meta signals have also to be intercepted by the framework (lines 8–10). In case of asynchronous ports, the receiving thread has to handle the meta signals (this can be seen in Listing 3.3, line 20). The method sending the meta signals through the pipes can be seen in the lines 14–18. It simply sends the signal to all available output ports.

In order to process the meta signals, the plugin uses a counter to count the number of initialization and termination signals (lines 21–25). The plugin relays the incoming meta signals (line 31). If the plugin receives the last termination signal, the counter reaches zero. In this case the plugin shuts itself down, as can be seen in line 28. The asynchronous ports are terminated as well in line 34. The notification of the analysis controller about the filter's termination can be seen in line 39. In case the plugin uses asynchronous ports, the notification is not performed by the `processAndDelayMetaSignal` method. Instead the notification is performed by the asynchronous threads. This can be seen in Listing 3.2, line 27 and Listing 3.3, line 30. The last terminated port is responsible for the actual notification. This makes sure that the analysis controller is informed not until all threads have finished their work.

Listing 3.8. Excerpt from `AbstractPlugin` showing the meta signal handling

```
 1  private boolean deliver(String outputPortName, Object data,
 2      boolean checkForAsynchronousMode) {
 3    for (PluginInputPortReference pluginInputPortReference : registeredMethodsOfPort) {
 4      ...
 5      for (Class<?> eventType : eventTypes) {
 6        if (eventType.isAssignableFrom(data.getClass()) || (data instanceof MetaSignal)) {
 7          ...
 8          if (data instanceof MetaSignal) {
 9            receivingPlugin.processAndDelayMetaSignal((MetaSignal) data);
10            break;
11    ...
12  }
13
14  protected void sendMetaSignal(final MetaSignal metaSignal) {
15    for (String outputPort : outputPorts.keySet()) {
16      deliver(outputPort, metaSignal);
17    }
18  }
19
20  protected boolean processAndDelayMetaSignal(final MetaSignal data) {
21    if (data instanceof InitializationSignal) {
22      metaSignalCounter++;
23    } else if (data instanceof TerminationSignal) {
24      metaSignalCounter--;
25    }
26
```

```
  if (metaSignalCounter == 0) {
    shutdown(((TerminationSignal) data).isError());
  }

  sendMetaSignal(data);

  if (metaSignalCounter == 0) {
    shutdownAsynchronousPorts();
  }

  if (!activeThreads && (data instanceof TerminationSignal) &&
      (metaSignalCounter == 0)) {
    ((AnalysisController) projectContext).notifyFilterTermination(this);
  }

  return true;
}
```

Using the Concurrent Part of the Framework In order to use concurrent filters in an analysis, it is only necessary to add the additional configuration. Listing 3.9 shows how the ports of a `TypeFilter` are configured to be asynchronous. The only modifications necessary are in the lines 7–11.

Listing 3.9. Additional configuration to use the concurrent part of the framework

```
 1  // Create the analysis controller
 2  final IAnalysisController ac = new AnalysisController();
 3  ...
 4  // Prepare the configuration for the filter
 5  final Configuration tfConfiguration = new Configuration();
 6
 7  tfConfiguration.setProperty(AbstractPlugin.CONFIG_ASYNC_INPUT_PORTS,
 8      TypeFilter.INPUT_PORT_NAME_EVENTS);
 9  tfConfiguration.setStringArrayProperty(AbstractPlugin.CONFIG_ASYNC_INPUT_PORTS,
10      new String[] {TypeFilter.OUTPUT_PORT_NAME_TYPE_MATCH,
11                    TypeFilter.OUTPUT_PORT_NAME_TYPE_MISMATCH});
12
13  // Create the filter using the analysis controller and the prepared configuration
14  final TeeFilter teeFilter = new TeeFilter(tfConfiguration, ac);
15  ...
16  ac.run();
```

Refused Approaches The both methods to set a port into an asynchronous mode, `setOutputPortToAsynchronousMode` and `setInputPortToAsynchronousMode`, were originally public methods. It was intended that a programmer would call this methods on the plugins after the instantiation, but before starting the analysis. This idea was later refused, as it was counterintuitive with respect to the already existing configuration objects.

Another approach for the configuration was an additional flag at the port annotations. This can be seen in Listing 3.10. The flag in line 3 determines that the output port for the timestamps would be asynchronous. This idea was refused. It would mean that a filter could be exclusively used either concurrent or sequential. This solution was therefore not generic.

Listing 3.10. Additional flag at the port annotation

```
1  @Plugin(
2    outputPorts = {
3      @OutputPort(name = "timestamps", eventTypes = Long.class, asynchronous = true),
4    ...)
5  public final class TimeReader extends AbstractReaderPlugin {
```

3.2.2 Distributed Part

Basic Distributed Design For this part of the thesis, we implement a support for distributed analyses. In order to provide a better abstraction for an analysis, we aggregate filters and readers into computation nodes. Those nodes can either run in a distributed or in a local mode. If used in a distributed way, the basic design is similar to the concurrent one. We add FIFO buffers and use a sender and a receiver thread for each node. The sender reads blocking 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. The design is shown in Figure 3.3.

Message Delivery A possible approach for the message delivery would be to rely on a CEP engine. However, we use only a slimmed-down message engine and rely on Apache ActiveMQ [ASF 2013a] as Message Oriented Middleware (MOM). Although not providing a full query language, ActiveMQ has the advantage, that it can dynamically create new topics and connect to them.

Each distributed node is configured with a unique name in a distributed analysis. This name is used to create a topic with the same name. If a node should be connected with another node, it is registered for the topic. Assume, for example, two nodes with the names "Node 1" and "Node 2". If both are configured to run in a distributed way, they open topics with the name "Node 1" and "Node 2", respectively. If the output port of the first node should be connected with the input port of the second node, the second node simply registers for the topic "Node 1". The messages are delivered automatically by the MOM and the nodes do not have to know their successors.

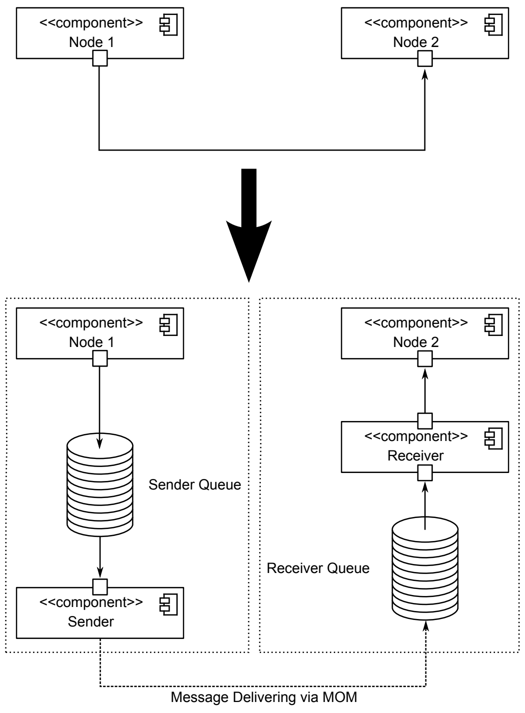


Figure 3.3. The basic distributed design

Repositories Within distributed analyses, repositories have to be specifically treated. As mentioned earlier, repositories do not have a formally specified interface for the access. We present therefore an approach which uses repository input and output ports. Listing 3.11 shows the repository input ports of repositories. Similar to usual input ports, we use annotations to mark the input port methods (lines 3–4). A difference is the additional return type, as repositories within Kieker usually return values.

Listing 3.11. Repository input ports for repositories

```
1  public class TypeRepository extends AbstractSystemSubRepository {
2    ...
3    @RepositoryInputPort(name = "lookupComponentTypeByNamedIdentifier",
4                         eventTypes = String.class, returnType = ComponentType.class)
5    public ComponentType lookupComponentTypeByNamedIdentifier(String namedIdentifier) {
6      ...
7    }
8  }
```

In order to access the repositories' ports, a plugin would now have to use various repository output ports with suitable types. We provide a new deliver method to access the repositories and receive the potential return values. Although this approach does work, it has some major drawbacks. One disadvantage is the amount of additional input ports. Every method within the repositories that should be accessed has to be annotated. Another disadvantage occurs when using the repositories in a distributed way. The plugins in Kieker often use the returned values from the repositories as keys for later accesses. When sending objects through the network, object identities go astray though. As solving the latter problem would go beyond the scope of this thesis, we consider this approach only theoretically.

Analysis Nodes We implement the analysis nodes as extensions of usual filter plugins. In order to simplify the message delivery, we add four ports to each node. One input, one output, one repository input, and one repository output port. Some of the ports and the configuration properties can be seen in Listing 3.12. In addition to the usual ports (lines 3 and 14–17) we use also new internal ports (lines 4–5 and 19–26). We use these internal ports to connect the filters contained in the node with the node itself. If, for example, the node receives a record via the input port (lines 14–17) it is simply send to the internal output port and therefore delivered to the internal filters. If an internal filter sends a record to the node's internal input port, the record is – if necessary – send to the MOM (line 23) and delivered via the usual output port (line 25). In case of a distributed node, the message is send by an asynchronous thread.

Necessary configurations for the nodes are the host name of the MOM (line 8), a flag whether the node is distributed (line 9), and an unique node name (line 10). The potential connection to the MOM is established within the constructor.

Listing 3.12. The ports and configuration of an analysis node

```
1  @Plugin(
2    outputPorts = {
3      @OutputPort(name = "sentEvents", eventTypes = Object.class),
4      @OutputPort(name = "internalOutputPort", eventTypes = Object.class,
5        internalUseOnly = true) },
6    ...
7    configuration = {
8      @Property(name = "server", defaultValue = "tcp://localhost:61616"),
9      @Property(name = "distributed", defaultValue = "false"),
10     @Property(name = "nodeName", defaultValue = "kieker-node"),
11 })
12 public class AnalysisNode extends AbstractFilterPlugin {
13   ...
14   @InputPort(name = INPUT_PORT_NAME_EVENTS, eventTypes = Object.class)
15   public final void inputPort(final Object data) {
16     super.deliver(INTERNAL_OUTPUT_PORT_NAME_EVENTS, data);
17   }
18
```

```
  @InputPort(name = INTERNAL_INPUT_PORT_NAME_EVENTS, eventTypes = Object.class,
    internalUseOnly = true)
  public final void internalInputPort(final Object data) {
    if (this.distributed) {
      this.sendQueue.add(data);
    }
    super.deliver(OUTPUT_PORT_NAME_EVENTS, data);
  }
  ...
}
```

We make sure that plugins can only be contained within one analysis node. This is ensured by by providing a suitable creation method. This is shown in Listing 3.13. A given plugin is created with the given configuration (lines 3–8). After the creation, it is registered with the analysis node in line 10.

Listing 3.13. The method to create new plugins within an analysis node

```
 1 public <T extends AbstractAnalysisComponent> T createAndRegister(Class<T> componentClass,
 2       Configuration configuration) {
 3   // Find the analysis default constructor
 4   Constructor<? extends AbstractAnalysisComponent> constructor =
 5       componentClass.getConstructor(Configuration.class, IProjectContext.class);
 6   // Create an instance ...
 7   AbstractAnalysisComponent concreteComponent =
 8       constructor.newInstance(configuration, this.projectContext);
 9   // ... and register it with this node
10   concreteComponent.registerWithinComponent(this);
11
12   // Return the newly created component
13   return (T) concreteComponent;
14 }
```

Analysis Start We do not provide a possibility to automatically start all nodes for a distributed analysis. It is necessary to manually start each distributed analysis.

Analysis Shutdown For the shutdown of an analysis we have to modify the termination sequence used in the concurrent part. As an analysis node has two input ports and one of them is for internal use only, the unmodified termination sequence could lead to an infinite feedback loop. We add an additional flag to the `sendMetaSignal` method in order to decide whether the signal comes from outside an analysis node or not. The method within `AbstractPlugin` works as before. However, we overwrite it in `AnalysisNode`. This can be seen in Listing 3.14. If the signal comes from outside the node, it is simply delivered via the internal output port (lines 3–4). If it comes from inside the node, the signal is sent to the MOM and to the external output port (lines 6–9).

Listing 3.14. The modified method to relay the meta signals

```java
@Override
protected void sendMetaSignal(MetaSignal metaSignal, boolean signalFromOutside) {
  if (signalFromOutside) {
    super.deliver(INTERNAL_OUTPUT_PORT_NAME_EVENTS, metaSignal);
  } else {
    if (this.isDistributed()) {
      this.sendQueue.add(metaSignal);
    }
    super.deliver(OUTPUT_PORT_NAME_EVENTS, metaSignal);
  }
}
```

Whether the signal comes from the inside or the outside is decided in `AbstractPlugin`. This can be seen in Listing 3.15. We make sure that meta signals are never refused by the framework (line 7). From line 10 on we intercept the meta signal. If the signal should be sent to the internal input port of an analysis node, the signal is marked as internal signal. This is checked and done in the lines 14–19.

Listing 3.15. The modified `deliver` method

```java
 1  private boolean deliver(String outputPortName, Object data,
 2      boolean checkForAsynchronousMode) {
 3    for (PluginInputPortReference pluginInputPortReference : registeredMethodsOfPort) {
 4      ...
 5      for (Class<?> eventType : eventTypes) {
 6        // Make sure that meta signals are always send through
 7        if (eventType.isAssignableFrom(data.getClass()) || (data instanceof MetaSignal)) {
 8          ...
 9          // Intercept the meta signals
10          if (data instanceof MetaSignal) {
11            boolean signalFromOutside = true;
12            // If the signal is send to the internal input port of a node, it has to
13            // marked as internal signal
14            if (receivingPlugin instanceof AnalysisNode) {
15              final String inputPortName = pluginInputPortReference.getInputPortName();
16              if (inputPortName.equals(AnalysisNode.INTERNAL_INPUT_PORT_NAME_EVENTS)) {
17                signalFromOutside = false;
18              }
19            }
20
21            receivingPlugin.processAndDelayMetaSignal((MetaSignal) data,
22                signalFromOutside);
23            break;
24            ...
25  }
```

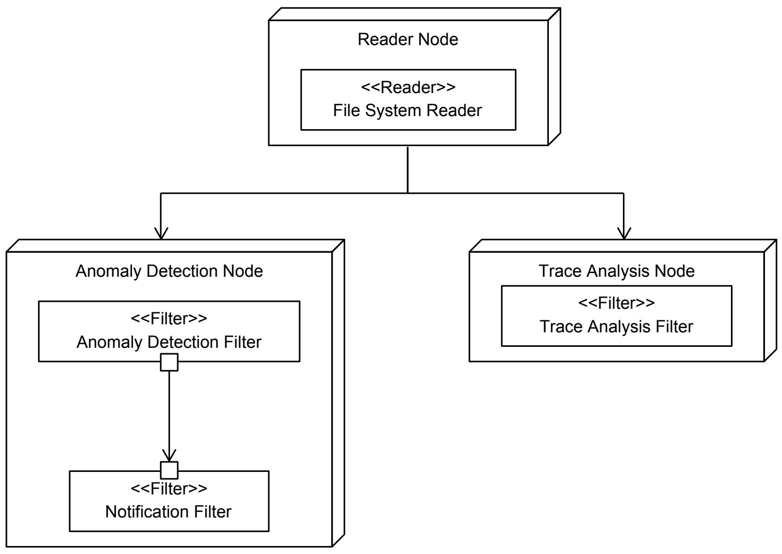


Figure 3.4. Exemplary distributed analysis for anomaly detection and trace analysis

Using the Distributed Part of the Framework In order to show the additional distributed configurations, we use the example from Chapter 1. Although Figure 3.4 shows only an example with hypothetical filters, it is suitable to present the creation and the configuration of a node.

As nodes are just special filters, they can be created like all other components (line 8). An additional property can be used to decide whether the node is distributed or local (line 4). The unique name of the node (line 5) and the host name of the MOM server (line 6) are configured as well.

In order to create new plugins that are contained in the node, we use the new creation method (line 10). The parameters are the class of the plugin to be created and its configuration. Plugins within nodes can be connected as usually (line 13). We provide additional methods to connect filters with the input or the output ports of the nodes (line 15). Connecting to other nodes works similar, but requires the unique name of the node (line 16).

Once the analysis has been created, it can be started as usually. This can be seen in line 18.

Listing 3.16. Configuration of the anomaly detection node of the example in Figure 3.4

```
1  final IAnalysisController ac = new AnalysisController();
2
3  final Configuration nodeConf = new Configuration();
4  nodeConf.setProperty(AnalysisNode.CONFIG_PROPERTY_NAME_DISTRIBUTED, "true");
5  nodeConf.setProperty(AnalysisNode.CONFIG_PROPERTY_NAME_NODE_NAME, "adNode");
```

```
nodeConf.setProperty(AnalysisNode.CONFIG_PROPERTY_NAME_MOM_SERVER, "tcp://blade1:61616");

final AnalysisNode adNode = new AnalysisNode(nodeConf, ac);

final ADFilter adFilter = adNode.createAndRegister(ADFilter.class, fsReaderConfig);
final NFilter nFilter = adNode.createAndRegister(NFilter.class, fsReaderConfig);

ac.connect(adFilter, "outputPort", nFilter, "inputPort");

adNode.connectWithInput(adFilter, "inputPort");
adNode.connect("readerNode");

ac.run();
```

Chapter 4

Evaluation

In this chapter we determine the performance and the memory consumption of the new framework implementations. For both the concurrent and the distributed part we perform different experiments. We detail our experimental environment and our statistical methods, before we describe the actual experiments and results. Finally we compare the results with our GQM goals from Chapter 3.

4.1 Experimental Environment

We perform all experiments with changeset 60ac6ef0bfd641f9f9b787f108c5bccdd3ec2075 of the project on various blade servers. The available blade servers and their configurations are listed in Table 4.1.

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

Table 4.1. Blade servers used for the experiments

We use the following Java Runtime Environment (JRE) for all experiments. Depending on the blade server, we use either the JRE for Solaris 64-Bit or for Solaris Sparc 64-Bit.

- ▻ Java Version 1.7.0_25
  - ▻ Java SE Runtime Environment (build 1.7.0_25-b15)
  - ▻ Java HotSpot 64-Bit Server VM (build 23.25-b01, mixed mode)

The JRE is started - if not otherwise noted - with the additional command line parameters `-Xms512m` and `-Xmx18G`. These arguments make sure that the Java Virtual Machine (JVM) provides enough heap space for the experiments.

4.2 Statistical Methods

Each experiment consists of five to six data sets and 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. For the comparison, we use diagrams with the execution time, the memory consumption, the CPU utilization, the speedup, the efficiency, and the throughput.

The plotted values in the execution time, memory consumption, and CPU utilization 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.

The speedup within the diagrams is plotted as a bar, as it can obtain different values due to the confidence intervals. 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. In a uniform manner we calculate and plot the throughput.

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 use only diagrams for the evaluation in this chapter. However, we provide the complete measurement data on a DVD in the appendix. This includes log files, raw data, and statistical prepared data.

4.3 Experiments for Concurrent Analyses

For the evaluation of the concurrent part we perform seven experiments. The test system for all experiments in this section is Blade 1.

We deactivate the file writing of one filter. This is necessary to avoid extensive file system activities during some of the experiments. Said filter is the sequence diagram filter. It is responsible for writing reconstructed traces as sequence diagrams on the file system. Instead we reprogram the filter to write the diagram into the memory and discard it.

We also reprogram the tool, executing the analyses, to repeat each execution five times. Those five repetitions are performed in the same JVM to provide an appropriate warm up. For the statistical analysis of the data sets we use only the fifth repetition of each execution.

During each of the experiments, we also execute Sigar 1.6.4 in a second JVM. Sigar is used to monitor and log the CPU and memory usage in 500 milliseconds intervals. For each of the ten runs of the configurations, we localize the minimal and the maximal memory usage. The difference of both values is used for the statistical analysis. For the CPU utilization analysis, we determine the average utilization across all processors during each of the runs.

To obtain the execution times, we use the log files, which are produced during each experiment. As we add additional log messages to the executing tool, we are able to read the needed times to the millisecond.

4.3.1 Experiment 1 - CPU and MemSwap Record Processing

Experimental Setup

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 a 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.

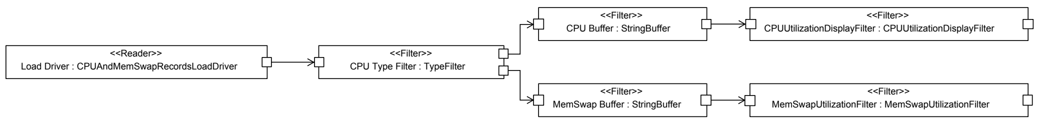
The load driver is configured to generate $2^{16}, \ldots, 2^{21}$ data sets. Each of these data sets consists of 16 CPU records and one MemSwap record. The driver generates therefore $17 \cdot 2^{16}, \ldots, 17 \cdot 2^{21}$ records in our setup.

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.

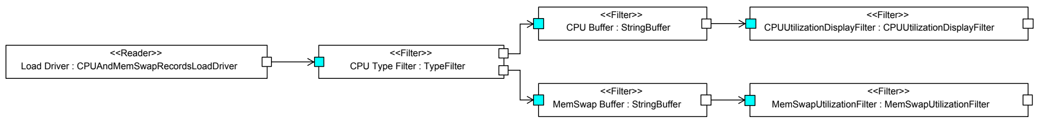
We compare three types of configurations. The first configuration uses no asynchronous ports at all (*Synchronous*). For the second configuration we set all available input ports into an asynchronous mode (*Full Asynchronous*). A third configuration sets only the input ports of the buffer filters into an asynchronous mode (*Asynchronous*). The detailed setup with all three configurations can be seen in Figure 4.1.

Memory Consumption

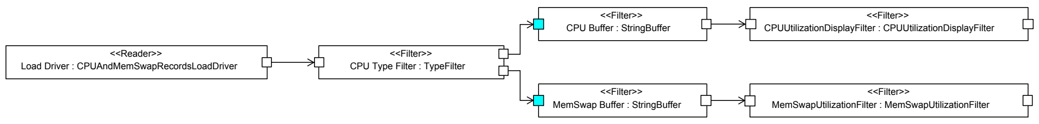
The average memory consumption during the experiment is shown in Figure 4.2a. The necessary memory for the synchronous configuration slightly fluctuates, but can be considered as constant. This is the expected behavior, as the data structures within the filters are filled with a constant amount of data after some hundred records. Both asynchronous configurations require much more memory during the executions though. The memory usage for the asynchronous configuration is up to ten times higher than


<<Reader>>
Load Driver : CPUAndMemSwapRecordsLoadDriver
<<Filter>>
CPU Type Filter : TypeFilter
<<Filter>>
CPU Buffer : StringBuffer
<<Filter>>
MemSwap Buffer : StringBuffer
<<Filter>>
CPUUtilizationDisplayFilter : CPUUtilizationDisplayFilter
<<Filter>>
MemSwapUtilizationFilter : MemSwapUtilizationFilter

(a) *Synchronous*


<<Reader>>
Load Driver : CPUAndMemSwapRecordsLoadDriver
<<Filter>>
CPU Type Filter : TypeFilter
<<Filter>>
CPU Buffer : StringBuffer
<<Filter>>
MemSwap Buffer : StringBuffer
<<Filter>>
CPUUtilizationDisplayFilter : CPUUtilizationDisplayFilter
<<Filter>>
MemSwapUtilizationFilter : MemSwapUtilizationFilter

(b) *Full Asynchronous*


<<Reader>>
Load Driver : CPUAndMemSwapRecordsLoadDriver
<<Filter>>
CPU Type Filter : TypeFilter
<<Filter>>
CPU Buffer : StringBuffer
<<Filter>>
MemSwap Buffer : StringBuffer
<<Filter>>
CPUUtilizationDisplayFilter : CPUUtilizationDisplayFilter
<<Filter>>
MemSwapUtilizationFilter : MemSwapUtilizationFilter

(c) *Asynchronous*

Figure 4.1. Setup of Experiment 1. The asynchronous ports are highlighted in turquois.

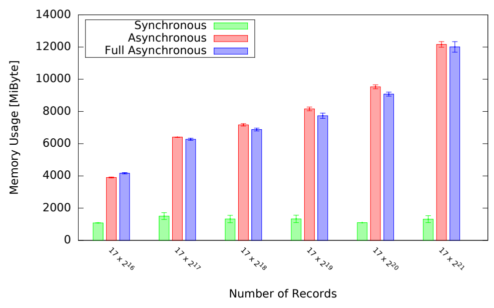
the one of the synchronous configuration. The higher memory requirement can be explained with the additional buffers between the filters. However, the increase in memory consumption is not uniform and can therefore not be considered as linear.

Noticeable is the fact, that the asynchronous configuration with more threads (and therefore with more buffers) does not have a significantly higher memory consumption than the other asynchronous configuration for most data sets. Under inclusion of the execution times, we assume that the full asynchronous configuration is able to process the contents of the queues faster.

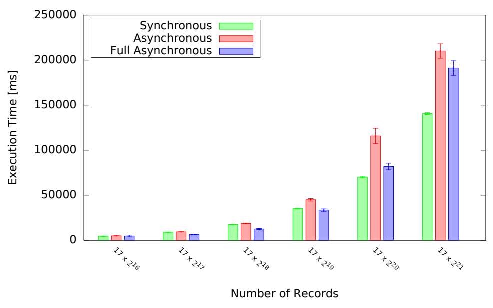
Performance

The execution times can be seen in Figure 4.2b. The first noticeable fact is that the full asynchronous configuration is faster than the other asynchronous configuration for most records. The configuration with more threads can process the records faster. When comparing the synchronous configuration with the asynchronous configurations, one can see, that the synchronous configuration is usually faster than the remaining configurations. For a specific amount of records however (between $17 \cdot 2^{17}$ and $17 \cdot 2^{18}$ records) the full asynchronous configuration is slightly faster.

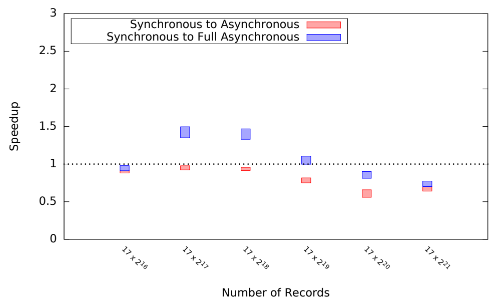
The speedup is shown in Figure 4.2c. Both speedups are near or below one for a higher number of records. For the mentioned specific area the framework can reach a slightly positive speedup with both asynchronous configurations though. As it is not expected that the speedup gets worse, we include the CPU utilization (Figure 4.2d) for the analysis. The dotted lines mark the theoretical upper limit for the utilization (e.g., the asynchronous configuration uses three threads and can theoretically use only $3/8$ of



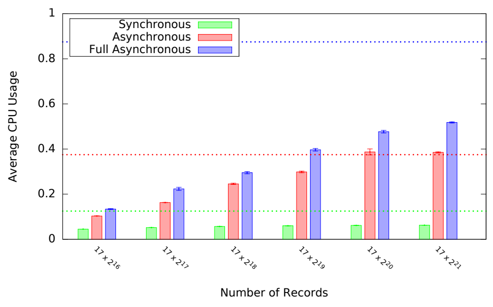
(a) Average memory usage



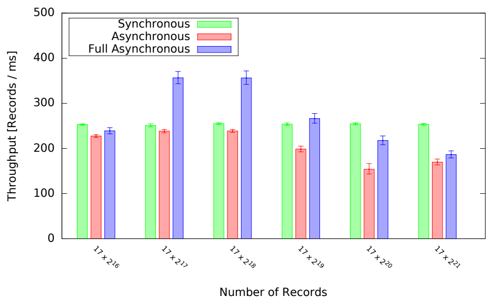
(b) Average execution time



(c) Speedup



(d) Average CPU usage



(e) Average throughput

Figure 4.2. Results of Experiment 1

the available computational power). It can be seen that the synchronous configuration uses only one of the available processors. The asynchronous configuration with two asynchronous ports does not only take full advantage of the theoretical computational power, but it even exceeds it slightly. We assume that the Garbage Collector (GC) of Java uses the additional computational power, which also explains the worse speedup.

The remaining asynchronous configuration does not reach its theoretical computational power. However, we assume due to the high memory usage, that the GC is similar active in this configuration.

The average throughput can be seen in Figure 4.2e. The synchronous configuration processes approximately 250 records per millisecond. The asynchronous configuration can process between 140 and 240 records per millisecond. The full asynchronous configuration processes between 170 and 350 records per millisecond. The full asynchronous configuration can therefore process at best approximately 100 records more than the synchronous configuration.

Note that the asynchronous configuration cannot take full advantage of the additional threads in this setup. For each of the MemSwap records, we produce 16 CPU records. This means that the display filter for the CPU records is way more busy than the display filter for the memory records. We repeat this experiment therefore with a modified ratio between both record types.

4.3.2 Experiment 2 - CPU and MemSwap Record Processing

Experimental Setup

The setup of this experiment is similar to the one of Experiment 1. The only difference is the configuration of the load driver. It generates now $2^{19}, \dots, 2^{24}$ data sets each consisting of one CPU record and one MemSwap record. The amount of records to process is not exactly the same as in Experiment 1, but are in the same order of magnitude. The ratio is now balanced and we expect a higher speedup.

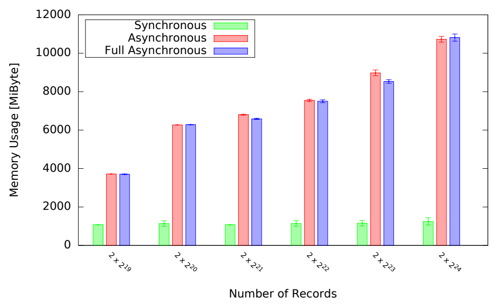
Memory Consumption

The memory usage can be seen in Figure 4.3a. The behavior is very similar to the one in Experiment 1. Both asynchronous configurations require still much more memory than the synchronous configuration. However, the asynchronous configurations require again almost the same amount of memory.

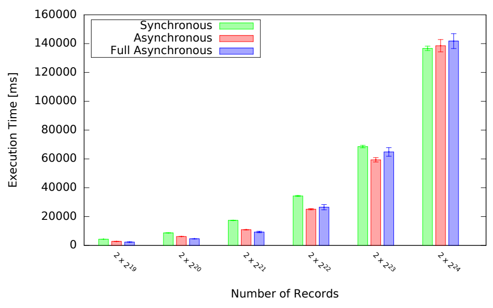
The difference in the values between Experiment 1 and Experiment 2 can be explained with the lower number of processors in the CPU records. Therefore the buffer filters and the display filters have to store less values.

Performance

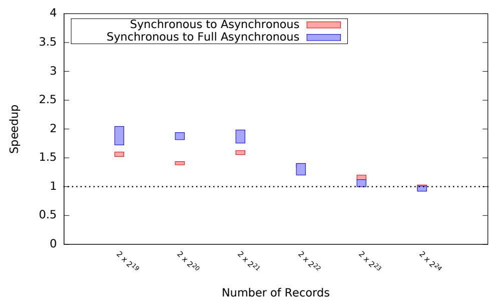
The runtime behavior is shown in Figure 4.3b. As expected, it is now better, but only for a lower number of records again. The initial speedup (Figure 4.3c) exceeds even two and is therefore much better. A comparison between the speedups from Experiment 1 and 2, however, shows that the initial better speedup quickly approaches the worse speedup from Experiment 1 (Figure 4.3d). It seems that the additional overhead of the



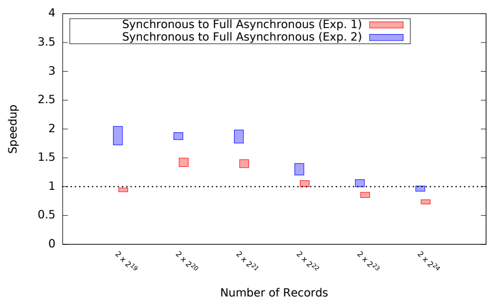
(a) Average memory usage



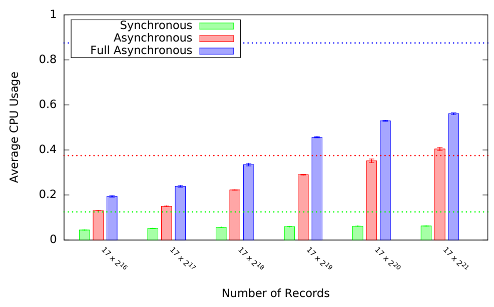
(b) Average execution time



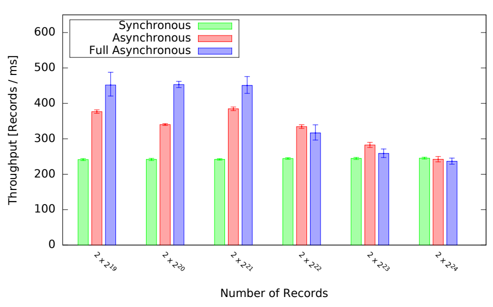
(c) Speedup



(d) Comparison of the Speedups from Experiment 1 and 2



(e) Average CPU usage



(f) Average throughput

Figure 4.3. Results of Experiment 2

concurrent part is only worthwhile within a specific range of records to process. We also have to assume that a lot of overhead results not only from communication between the processors, but also from the GC. This can be confirmed by the CPU utilization (Figure 4.3e), which is similar as the one in Experiment 1.

It can also be seen that the throughput (Figure 4.3f) is similar to the one in Experiment 1. The synchronous configuration processes almost the same amount of records per time unit as in Experiment 1. The full asynchronous configuration, however, benefits from the balanced ratio. At its best, it can process approximately 100 records per time unit more than before.

4.3.3 Experiment 3 - Trace Analysis

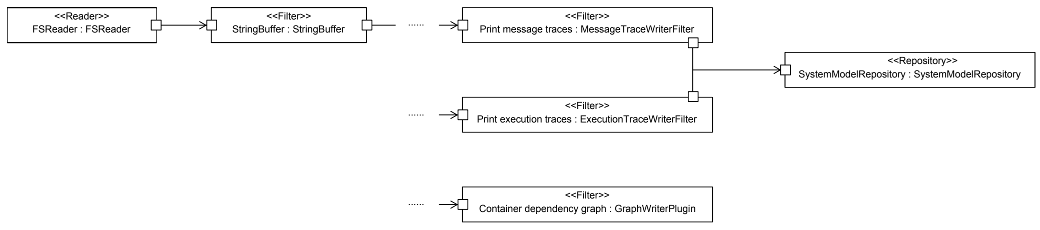
Experimental Setup

For the third experiment we use a modified configuration of Kieker's trace analysis tool. The configuration creates the following graphs and outputs during the analysis.

- ▻ Deployment component dependency graphs with response times
- ▻ Assembly component dependency graphs with response times
- ▻ Deployment operation dependency graphs with response times
- ▻ Assembly operation dependency graphs with response times
- ▻ Container dependency graphs
- ▻ Aggregated deployment call trees
- ▻ Aggregated assembly call trees
- ▻ Message traces
- ▻ Execution traces
- ▻ Invalid execution traces
- ▻ Deployment equivalence classes

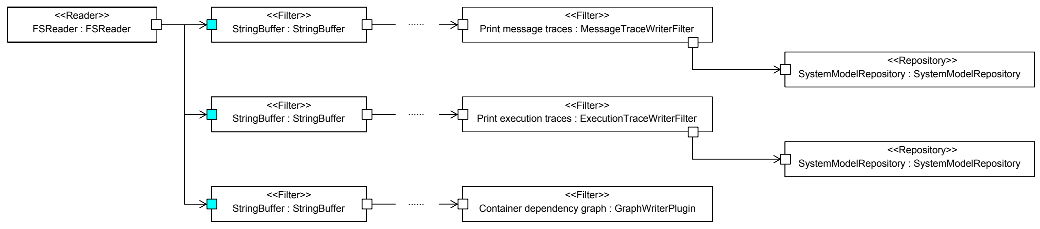
We test three types of configurations. A first configuration (*Synchronous*) uses the connections of filters as delivered by the trace analysis tool. As the used repositories within the analysis are not thread-safe at this point, we use a second configuration (*Asynchronous*) in which we clone the necessary 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. As this configuration performs more work than the first configuration, 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 filters contains no input port and is therefore difficult to use with out termination sequence. The simplified setup can be seen in Figure 4.4. The full setup contains more than hundred filters and is therefore difficult to picture.

The data sets for the experiment are taken from a monitoring of 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.



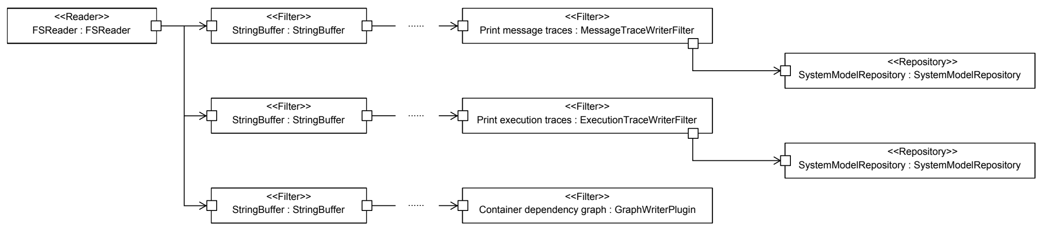
<<Reader>> FSReader : FSReader
<<Filter>> StringBuffer : StringBuffer
<<Filter>> Print message traces : MessageTraceWriterFilter
<<Repository>> SystemModelRepository : SystemModelRepository
<<Filter>> Print execution traces : ExecutionTraceWriterFilter
<<Filter>> Container dependency graph : GraphWriterPlugin

(a) *Synchronous*



<<Reader>> FSReader : FSReader
<<Filter>> StringBuffer : StringBuffer
<<Filter>> Print message traces : MessageTraceWriterFilter
<<Repository>> SystemModelRepository : SystemModelRepository
<<Filter>> StringBuffer : StringBuffer
<<Filter>> Print execution traces : ExecutionTraceWriterFilter
<<Repository>> SystemModelRepository : SystemModelRepository
<<Filter>> StringBuffer : StringBuffer
<<Filter>> Container dependency graph : GraphWriterPlugin

(b) *Asynchronous*



<<Reader>> FSReader : FSReader
<<Filter>> StringBuffer : StringBuffer
<<Filter>> Print message traces : MessageTraceWriterFilter
<<Repository>> SystemModelRepository : SystemModelRepository
<<Filter>> StringBuffer : StringBuffer
<<Filter>> Print execution traces : ExecutionTraceWriterFilter
<<Repository>> SystemModelRepository : SystemModelRepository
<<Filter>> StringBuffer : StringBuffer
<<Filter>> Container dependency graph : GraphWriterPlugin

(c) *Synchronous with Cloned Nodes*

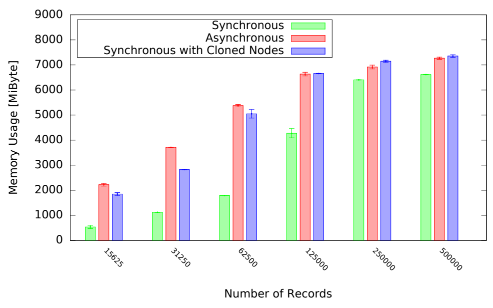
Figure 4.4. Simplified setup of Experiment 3. The asynchronous ports are highlighted in turquois.

Memory Consumption

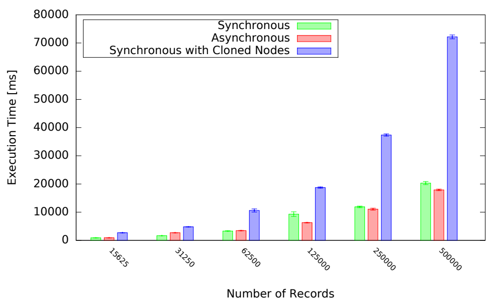
The average memory consumption is shown in Figure 4.5a. Noticeable is the fact, that the configurations with cloned nodes needs almost the same amount of memory as the asynchronous configuration. Although the asynchronous configuration uses additional buffers, it does not need more memory. Also, the memory consumption of the pure synchronous configuration without cloned nodes approaches the memory consumption of the remaining configurations. We assume that the additional filters and the buffers for the asynchronous ports become less important with a higher number of records.

Performance

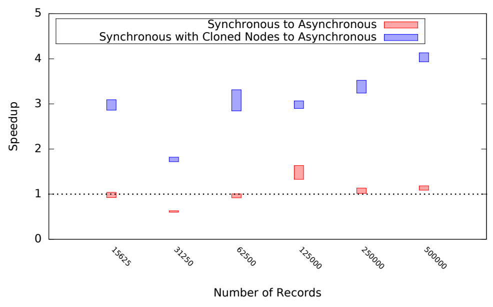
The CPU utilization in Figure 4.5d indicates already, that the asynchronous configuration does not take full advantage of the available processors. Although the configuration uses more than eight threads, it uses only half of the available processing power.



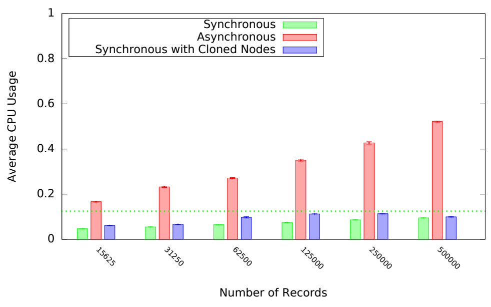
(a) Average memory usage



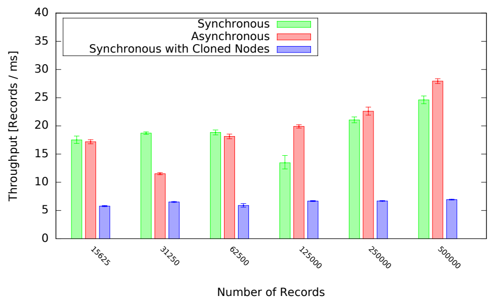
(b) Average execution time



(c) Speedup



(d) Average CPU usage



(e) Average throughput

Figure 4.5. Results of Experiment 3

The execution times are shown in Figure 4.5b. It can be seen that the asynchronous configuration is significantly slower than the first synchronous configuration for a lower number of records. The second synchronous configuration, however, performs the same (additional) work and is much slower that the asynchronous configuration. This can also be seen in the speedup diagram (Figure 4.5c). Within a specific range, the asynchronous configuration can be used to speedup the trace analysis. With regard to the synchronous configuration with cloned nodes, however, the speedup exceeds even four. The leap in the speedup is conspicuous, though. We repeat the experiment with new monitoring data to examine whether this is a data anomaly or not.

The throughput (shown in Figure 4.5e) is much lower than in the previous experiments. This is the expected behaviour, as the records are read from the file system instead of created in memory. However, this points out the strong influence of the reader on the performance.

4.3.4 Experiment 4 - Trace Analysis

Experimental Setup

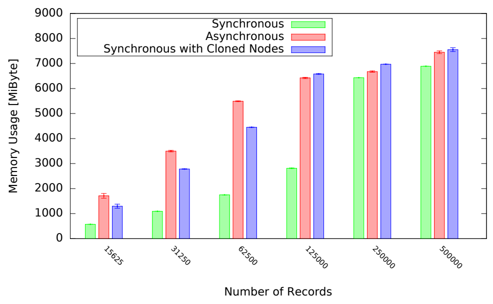
The setup of this experiment is similar as the one for Experiment 3. The only difference are the monitoring logs, which are now taken from a monitoring of the iBATIS JPetStore example. Again, the noted number of records is not precise. We remove several records per data set in order to avoid invalid traces.

Memory Consumption

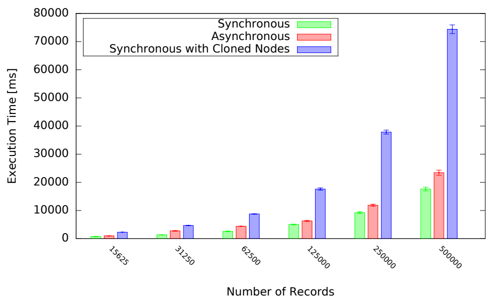
We show the memory consumption in Figure 4.6a. The behaviour is very similar to the one in Experiment 3. The synchronous configuration needs, for a higher number of records, almost the same amount of memory as the remaining configurations.

Performance

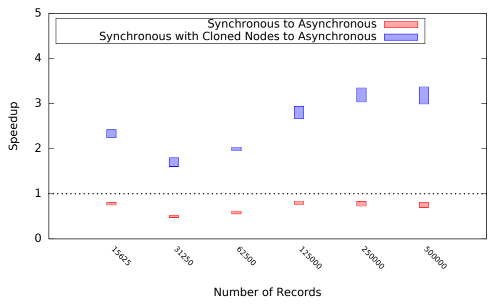
The CPU utilization, shown in Figure 4.6d, is similar as the one from the previous experiment. The asynchronous configuration does not take full advantage of the available processors. The execution time (Figure 4.6b) behaves also similar, but shows a slightly worsening of the asynchronous configuration. Using the speedup diagram (Figure 4.6c), we can see that the speedup is worse. The additional threads do not improve the performance of the trace analysis. We can therefore conclude that the actual speedup can be strongly influenced by the actual data. As the diagram shows also no conspicuous leap in this experiment, we assume that the leap from Experiment 3 is a data anomaly.



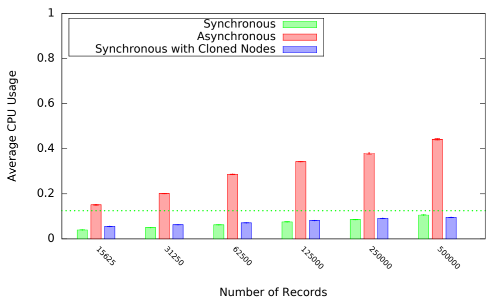
(a) Average memory usage



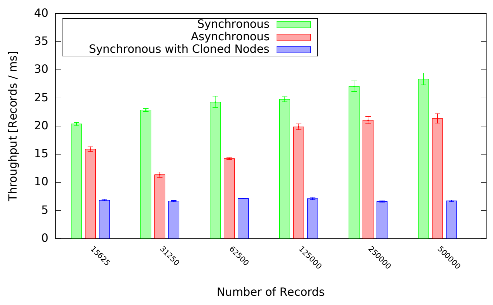
(b) Average execution time



(c) Speedup



(d) Average CPU usage



(e) Average throughput

Figure 4.6. Results of Experiment 4

4.3.5 Experiment 5 - Trace Reconstruction

Experimental Setup

For the fifth experiment a trace reconstruction is performed. Incoming records are transformed and used to reconstruct message traces. The traces are sent to a sequence diagram filter. A first configuration, we evaluate, performs this without any asynchronous ports (*Synchronous*). A second configuration (*Asynchronous*) uses a load balancer, which distributes the work based on the trace ID to four different computing nodes. Each of the computing nodes has an asynchronous input port. A third configuration (*Full Asynchronous*) uses five additional asynchronous input ports. The additional asynchronous ports are the input port of the load balancer and the input ports of the filters, transforming trace event records to execution and message traces. A visual representation of all three setups can be seen in Figure 4.7. The representation is simplified, as we omit some unnecessary ports.

The data sets for the experiment are taken from a monitoring of the Kieker.WebGUI project.

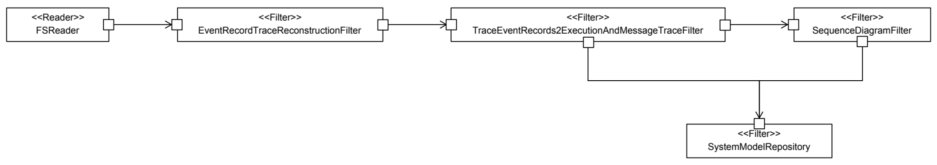
Memory Consumption

The memory usage can be seen in Figure 4.8a. Although some of the values tend to wild fluctuations, we can again recognize the strongly increased memory utilization of the asynchronous configurations. We can consider the memory consumption of both configurations as equal. The memory requirements of the synchronous configuration is constant.

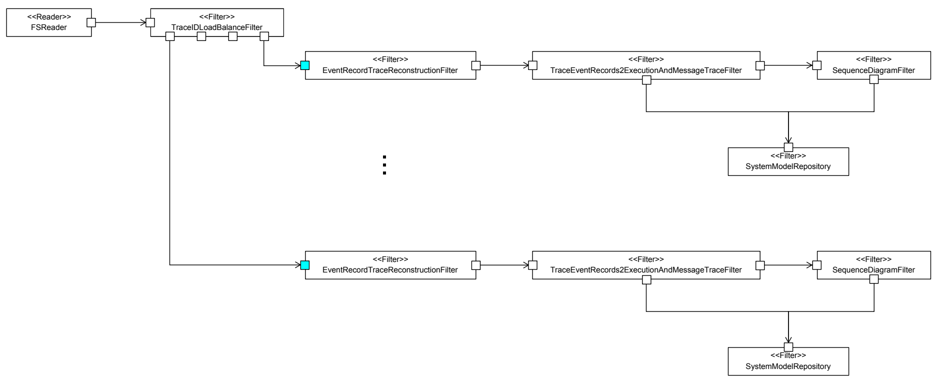
Performance

The average execution time can be seen in Figure 4.8b. Although both asynchronous configurations can be considered as equally fast, they are both significantly faster than the synchronous configuration. This results in a positive speedup (Figure 4.8c). The CPU diagram, however, indicates that additional computing power is hardly taken advantage of.

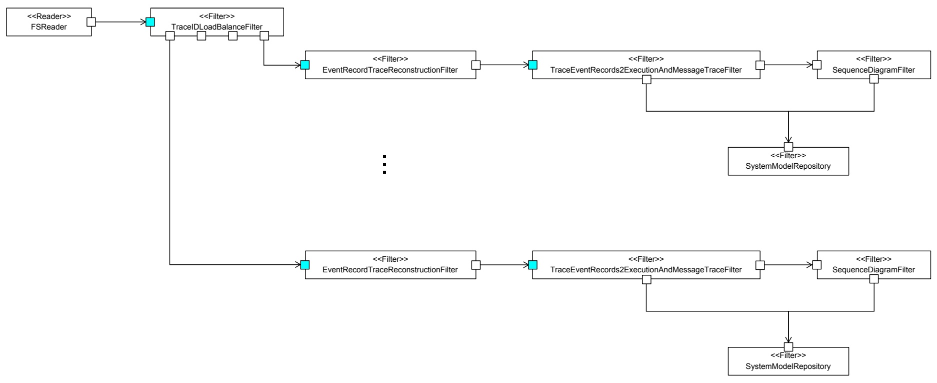
The throughput, as it can be seen in Figure 4.8e, is in the same order of magnitude as the one of the Experiments 3 and 4. However, it is a little bit higher. The asynchronous configurations can process between 40 and 50 records per millisecond.



(a) *Synchronous*

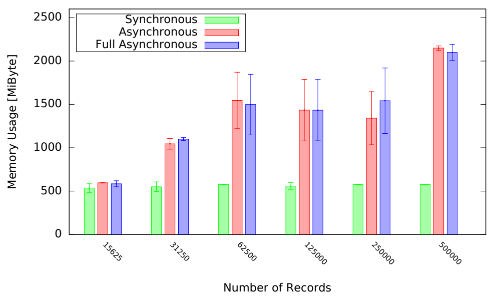


(b) *Asynchronous*

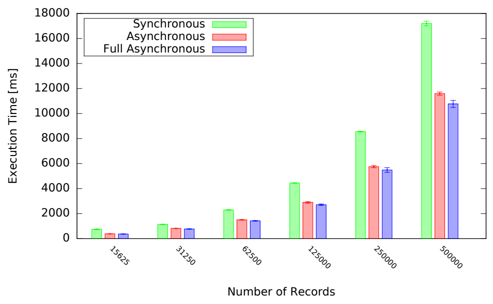


(c) *Full Asynchronous*

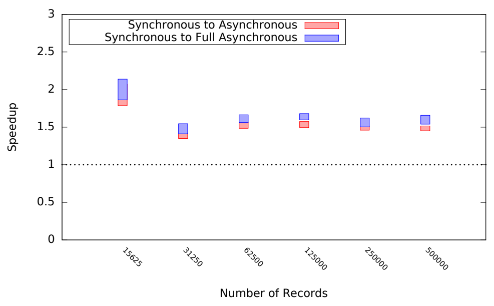
Figure 4.7. Setup of Experiment 5. The asynchronous ports are highlighted in turquois.



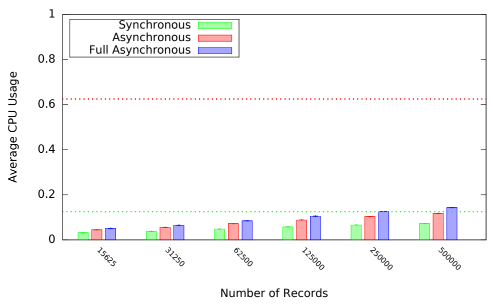
(a) Average memory usage



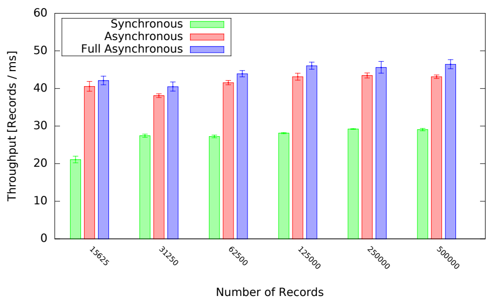
(b) Average execution time



(c) Speedup



(d) Average CPU usage



(e) Average throughput

Figure 4.8. Results of Experiment 5

4.3.6 Experiment 6 - Trace Reconstruction

Experimental Setup

The setup of this experiment is the same as the one for Experiment 5. The difference are the monitoring logs, which are now beforehand randomly generated. They consist of traces with a maximal depth of 100 calls. The name of the called operation changes every 100th trace.

Memory Consumption

The memory usage can be seen in Figure 4.9a. Both asynchronous configurations have again a higher memory consumption than the synchronous configuration. It is, however, noticeable, that the memory consumption of the synchronous configuration approaches the one of the remaining configurations.

Performance

The average execution time is shown in Figure 4.9b. Again both asynchronous configurations can be considered as equally fast. The synchronous configuration is also a little bit slower again. Using the deeper traces, the speedup (Figure 4.9c), however, is only slightly positive. The performance depends a lot on the actual monitoring data as well.

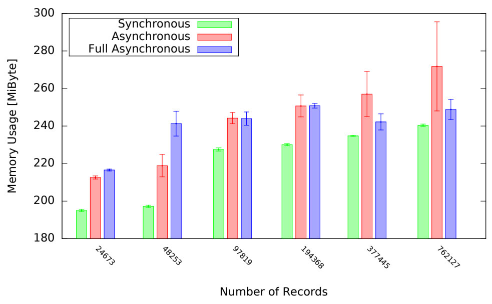
4.3.7 Experiment 7 - Trace Reconstruction

Experimental Setup

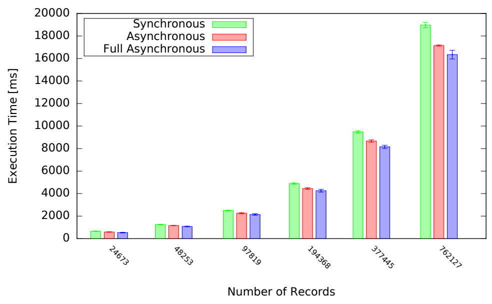
The setup of this experiment is similar as the one for Experiment 6. The difference is that we use various load balancers with different number of outputs. We vary the number of outputs from one to four. As we use also a synchronous configuration with one output, we vary the number of threads therefore from one to five. For the evaluation of this experiment we consider only the efficiency.

Results

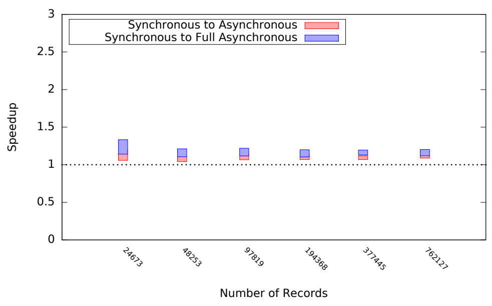
The efficiency can be seen in Figure 4.10a. The dotted lines show the theoretical minimal efficiency. It can be seen that the efficiency can be considered as relatively constant for each number of threads. It can also be seen that it is only slightly above the theoretical minimum. The framework can therefore take only very slight advantage of the additional parallel architecture during this experiment.



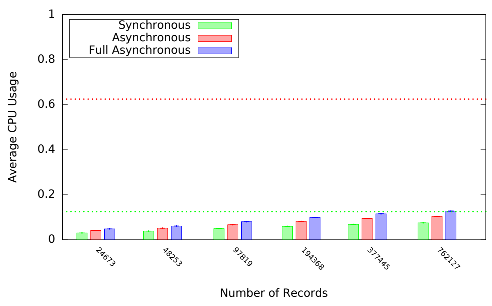
(a) Average memory usage



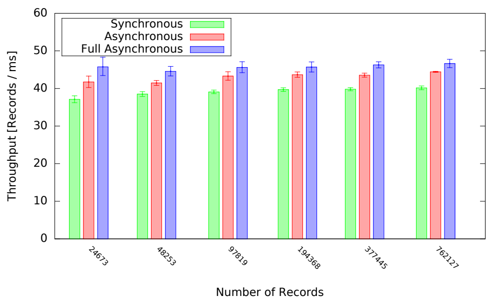
(b) Average execution time



(c) Speedup

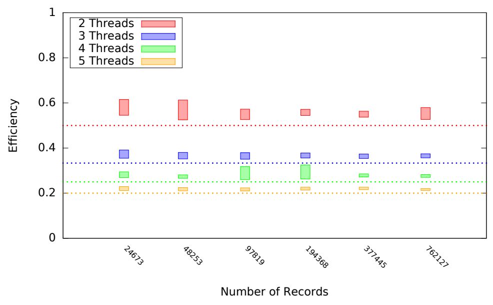


(d) Average CPU usage



(e) Average throughput

Figure 4.9. Results of Experiment 6



(a) Efficiency

Figure 4.10. Results of Experiment 7

4.4 Experiments for Distributed Analyses

For the evaluation of the distributed part we perform six experiments. The used test systems are listed for each experiment.

We program the analysis configurations to repeat each execution five times. Those five repetitions are performed in the same JVM to provide an appropriate warm up. For the statistical analysis of the data sets we use only the fifth repetition of each execution. In order to provide an automated test setup, we use waiting times between each repetition and execution. The reader node, for example, waits several minutes before rebuilding the reader components and sending the initialization signals. This waiting time makes sure that the remaining nodes can finish their work and can rebuild themselves. The precise amount of time is listed for each experiment.

During each of the experiments, we also execute Sigar 1.6.4 in a second JVM for each blade server. Sigar is used to monitor and log the CPU and memory usage in 500 milliseconds intervals. For each of the ten runs of the configurations, we localize the minimal and the maximal memory usage. The difference of both values is used for the statistical analysis. As we use waiting times between the runs, we renounce a CPU utilization analysis.

To obtain the execution times, we use the log files, which are produced during each experiment. As we add additional log messages to the executing tool, we are able to read the needed times to the millisecond.

4.4.1 Experiment 8 - CPU and MemSwap Record Processing

Experimental Setup

The setup is based on Experiment 1 for the concurrent running analyses. As a node can only have one output port, we use two nodes on the reader side. We designate them Node 1-1 and Node 1-2. The load driver sends the records to the type filter, which relays

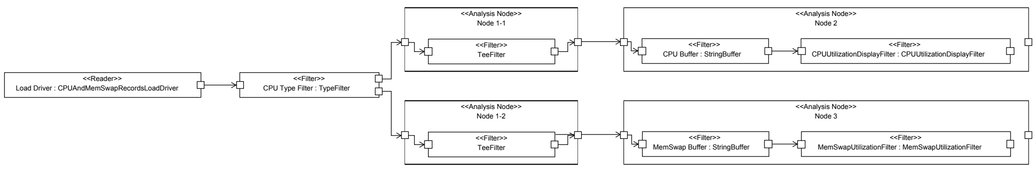

<<Reader>>
Load Driver : CPUAndMemSwapRecordsLoadDriver
<<Filter>>
CPU Type Filter : TypeFilter
<<Analysis Node>>
Node 1-1
<<Filter>>
TeeFilter
<<Analysis Node>>
Node 1-2
<<Filter>>
TeeFilter
<<Analysis Node>>
Node 2
<<Filter>>
CPU Buffer : StringBuffer
<<Filter>>
CPUUtilizationDisplayFilter : CPUUtilizationDisplayFilter
<<Analysis Node>>
Node 3
<<Filter>>
MemSwap Buffer : StringBuffer
<<Filter>>
MemSwapUtilizationFilter : MemSwapUtilizationFilter

Figure 4.11. Setup of Experiment 8

them to the correct node. The nodes itself contain just a filter to relay the data from the internal output to the internal input port. Node 2 contains the buffer filter and the display filter for the CPU records. Node 3 contains the remaining filers for the MemSwap records. The setup can be seen in Figure 4.11. Table 4.2 shows the deployment of the nodes on the blade servers. We use 120 seconds as waiting time between the executions.

Nodes	Blade Server
MOM	Blade 1
Load Driver, Type Filter, Node 1-1, Node 1-2	Blade 6
Node 2	Blade 0
Node 3	Blade 5

Table 4.2. Node deployment of Experiment 8

As the MOM is too slow to process the amount of records used during Experiment 1, we reduce the number of records and use the throughput in order to compare the experimental setups.

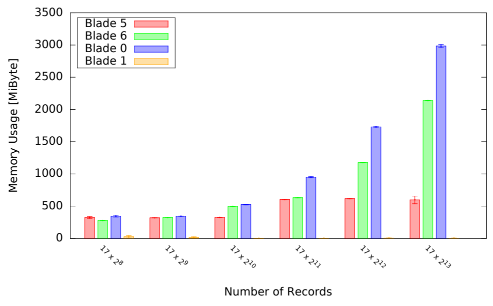
Memory Consumption

The memory consumption can be seen in Figure 4.12a. Blade 1, containing the MOM, requires only a very low amount of memory. Blade 6, containing the load driver, and Blade 0, containing the CPU display filter, require both more memory than Blade 5. This can be explained with the imbalanced ratio between memory and CPU records.

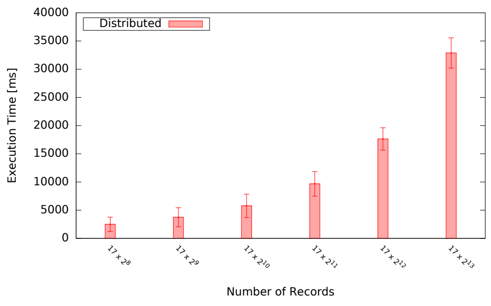
We do not compare the memory consumption of this experiment with the one of Experiment 1, as we use a greatly reduced amount of records.

Performance

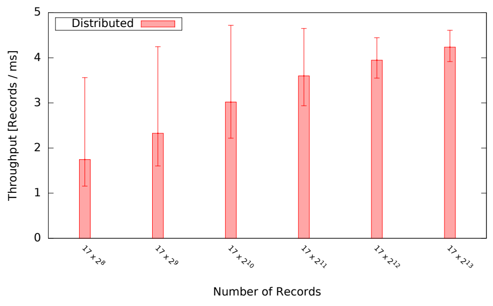
We show the throughput in Figure 4.12c. As the experiment processes only very few records per time unit, the values tend to wild fluctuations. For a higher number



(a) Average memory usage



(b) Average execution time



(c) Average throughput

Figure 4.12. Results of Experiment 8

of records, the values get more stable and are between three and five records per millisecond. Compared with the results of Experiment 1, the throughput is of another order of magnitude. It can be considered as 50 to 100 times lower.

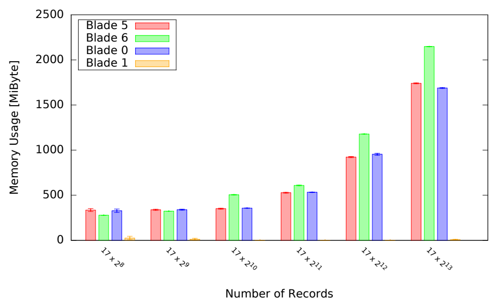
4.4.2 Experiment 9 - CPU and MemSwap Record Processing

Experimental Setup

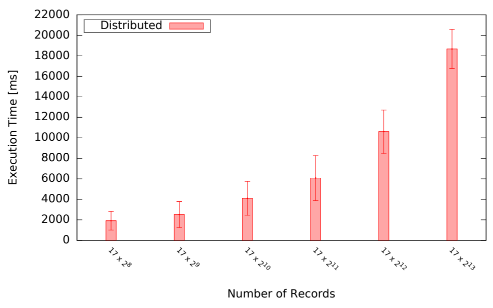
The setup of this experiment is similar to the one of Experiment 8. The only difference is the load driver. It is now configured for a balanced ratio between the two record types as in Experiment 2.

Memory Consumption

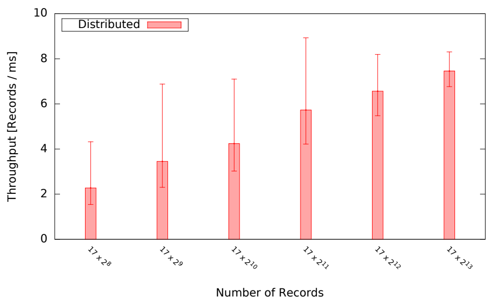
The memory consumption is shown in Figure 4.13a. The required memory of both working nodes (on Blade 0 and Blade 5) is now balanced as well.



(a) Average memory usage



(b) Average execution time



(c) Average throughput

Figure 4.13. Results of Experiment 9

Performance

The resulting throughput (shown in Figure 4.13c) is higher than in the previous experiment. It stabilizes between six and eight records per millisecond. Although the throughput is now twice as high, it is still very low, compared with the pure concurrent configurations.

4.4.3 Experiment 10 - Trace Analysis

Experimental Setup

This experiment is based on Experiment 3. We use the same monitoring log files, but renounce the last dataset, as it would require a very high waiting time. Each of the trace analysis tasks is performed by one node. The mapping can be seen in Table 4.3. The deployment of the nodes is shown in Table 4.4. The waiting time for this experiment is 210 seconds. We start the JVMs with the parameter `-Xmx10G`. The reduced heap size is necessary, as the available memory of the blade servers would otherwise be exceeded.

Node	Task
Node 2	Assembly component dependency graphs with response times
Node 3	Container dependency graphs
Node 4	Deployment equivalence classes
Node 5	Message traces
Node 6	Deployment component dependency graphs with response times
Node 7	Execution traces
Node 8	Aggregated assembly call trees
Node 9	Aggregated deployment call trees
Node 10	Assembly operation dependency graphs with response times
Node 11	Deployment operation dependency graphs with response times
Node 12	Invalid execution traces

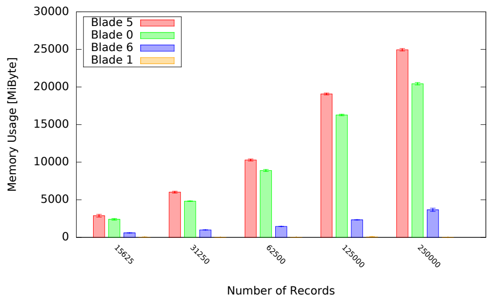
Table 4.3. Mapping between trace analysis tasks and nodes of Experiment 10

Nodes	Blade Server
MOM	Blade 1
Node 1	Blade 6
Node 2 – 6	Blade 0
Node 7 – 12	Blade 5

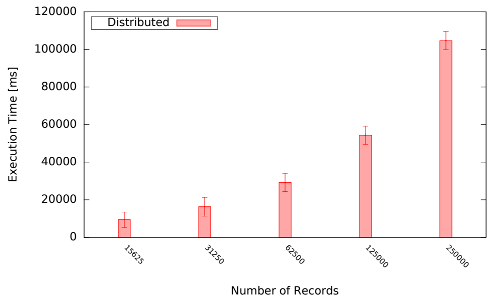
Table 4.4. Node deployment of Experiment 10

Memory Consumption

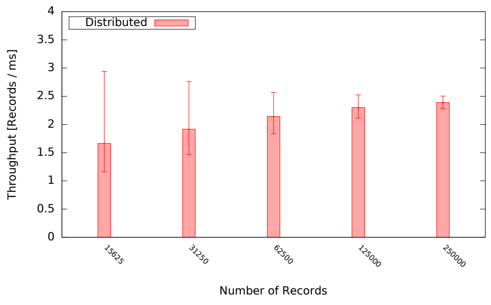
The memory consumption is shown in Figure 4.14a. The memory consumption of the MOM is, as in the previous experiments, very low. Compared with the remaining nodes, the requirements of the MOM can be neglected. The reader node has, compared with the working nodes on Blade 0 and Blade 5, a relatively low memory utilization. The nodes for the trace analysis tasks have a very high memory requirement. The analysis needs several times more memory than the plain asynchronous configurations.



(a) Average memory usage



(b) Average execution time



(c) Average throughput

Figure 4.14. Results of Experiment 10

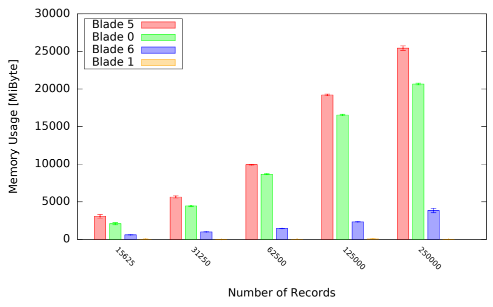
Performance

The throughput is shown in Figure 4.14c. The distributed analysis can not be used to accelerate the trace analysis. Compared with the results from Experiment 3, the processing is very slow. However, the performance is in a similar order of magnitude as the synchronous configuration with clones nodes from Experiment 3.

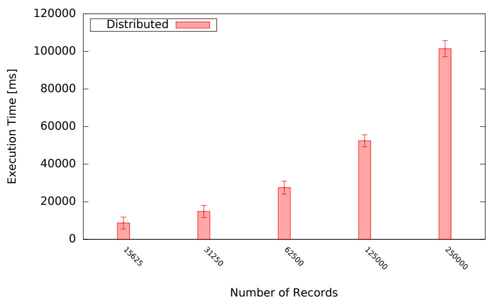
4.4.4 Experiment 11 - Trace Analysis

Experimental Setup

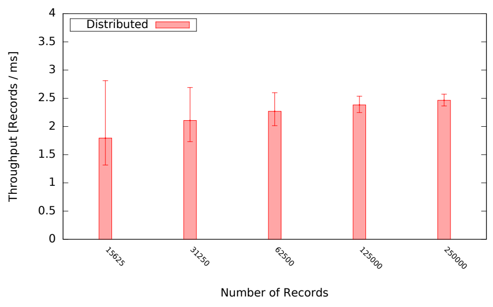
The setup of this experiment is similar to the one of Experiment 10. The difference is the monitoring data, which is now the same as the one in Experiment 4. We renounce the last dataset as well.



(a) Average memory usage



(b) Average execution time



(c) Average throughput

Figure 4.15. Results of Experiment 11

Memory Consumption

The memory consumption can be seen in Figure 4.15a. Both the reader and the MOM require again only a low amount of memory. The remaining nodes have a similar memory utilization as in the previous experiment.

Performance

The throughput is shown in Figure 4.15c. The performance is very similar to the one in Experiment 10. Unlike in Experiment 4, the actual throughput is not very sensitive to the actual monitoring logs. There is no leap in the throughput, neither in this nor in the previous experiment.

4.4.5 Experiment 12 - Trace Reconstruction

Experimental Setup

This experiment is based on Experiment 5. We use the same monitoring logs, but renounce the last data set. The reader and the load balancer are contained in Node 1. As a distributed node has only one output port, we use four analyses nodes in Node 1, each responsible for one output port of the load balance filter. The trace reconstruction is performed on the identical Nodes 2–5. Each of the nodes receives data from one output port of the load balancer. The deployment of the nodes is shown in Table 4.5. The waiting time for this experiment is 210 seconds.

Nodes	Blade Server
MOM	Blade 1
Node 1	Blade 6
Node 2 – 3	Blade 0
Node 4 – 5	Blade 5

Table 4.5. Node deployment of Experiment 12

Memory Consumption

The memory consumption is shown in Figure 4.16a. The MOM node requires again only very few memory. The remaining nodes, however, require more memory. Node 1 has a slightly higher consumption than the four working nodes. We assume that the additional memory is necessary, because Node 1 uses more threads, buffers, and components.

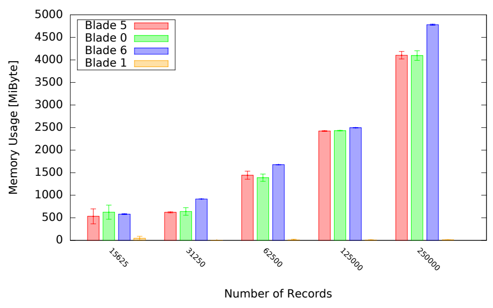
Performance

The throughput can be seen in Figure 4.16c. It stabilizes between seven and nine records per millisecond. The throughput of the synchronous configuration from Experiment 5 varies between 20 and 30 records per millisecond. However, it should be noted that the performance is again in the same order of magnitude.

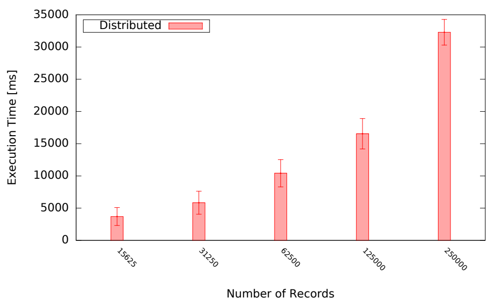
4.4.6 Experiment 13 - Trace Reconstruction

Experimental Setup

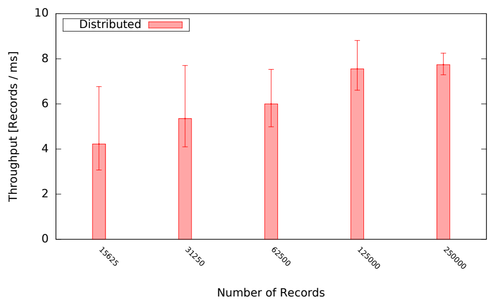
The setup of this experiment is similar to the one of Experiment 12. The only difference are the monitoring logs, which are now the one used for Experiment 6. Again we



(a) Average memory usage



(b) Average execution time



(c) Average throughput

Figure 4.16. Results of Experiment 12

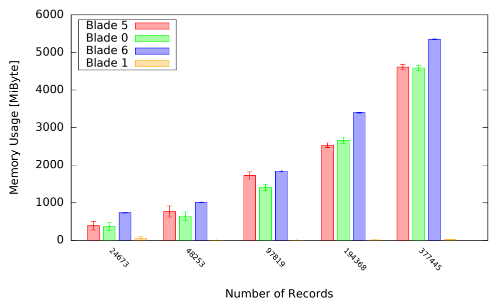
renounce the last data set in order to avoid a high waiting time between the experiments. The waiting time and the deployment are retained unchanged.

Memory Consumption

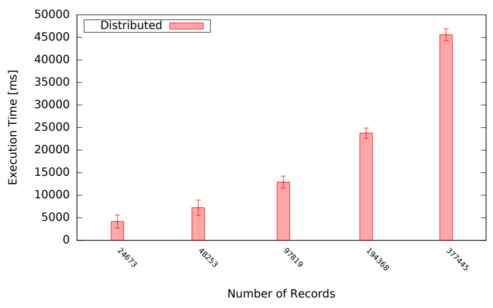
The memory consumption, shown in Figure 4.17a, is very similar to the one of the previous experiment. The blade server containing the MOM requires a negligible amount of memory. The working nodes on Blade 0 and Blade 5 have a very similar memory consumption. Blade 6, with the reader and the load balancer, requires again a slightly higher amount of memory for the sending and receiving threads and for the additional components.

Performance

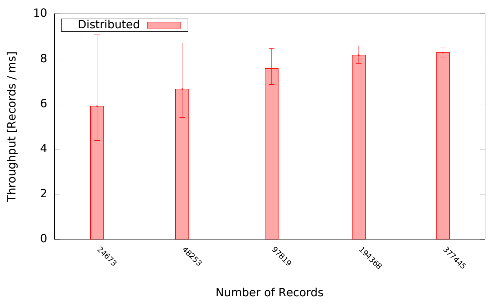
The throughput can be seen in Figure 4.17c. It is very similar to the one of the previous experiment as well. It can again be seen that the actual throughput is only slightly influenced by the monitoring data.



(a) Average memory usage



(b) Average execution time



(c) Average throughput

Figure 4.17. Results of Experiment 13

4.5 Comparison with the GQM Goals

In this section we verify whether the goals from Chapter 3 are achieved. We answer the corresponding questions to the goals using the listed metrics.

G1 Develop Kieker to support concurrent analyses

Q1.1 Do the modifications allow to use concurrent analyses?

The modifications allow to use additional threads for filters. It is possible to set filter ports into an asynchronous mode. As we use the already existing configuration objects, the concurrent part of the framework is easy to use and configure. Only little effort is necessary to extend an existing analysis configuration in order to use additional concurrency. The main drawback is the usage of repositories, which is not covered by the concurrent part of the framework. Also, the modifications of the termination sequence do not allow to use filters without input ports.

Q1.2 Do the modifications improve the performance of an analysis?

In some cases it is possible to improve the performance. In most cases, however, the additional communication overhead is too much in comparison of the work the filters perform. The efficiency is not very good either. The actual speedup depends very much on the actual scenario and the monitoring logs. It can be assumed that the additional overhead pays off for computationally intensive analyses with only very few records.

However, it should also be noted, that the additional buffers require a lot of space. As the buffers are not bounded, this can even lead to crashing analyses. The additional buffers result therefore in poor scalability.

Q1.3 How significant are the modifications with respect to the API?

The modifications are not very significant. An existing analysis configuration does not have to be changed to be still runnable with the modifications. The only method with changed behavior is the termination method. The number of public API classes and methods that changed is therefore one.

Q1.4 Is it possible to perform a clean shutdown of a running analysis?

The modifications of the shutdown sequence make sure, that the analysis runs and terminates autonomically. It is, however, not possible to terminate a running analysis from the outside. The new termination sequence is furthermore not designed for error cases.

Conclusion

Although the new framework does have some drawbacks, we can consider the goal as achieved.

Goal Achieved: ✓

G2 Develop Kieker to support distributed analyses

Q2.1 Do the modifications allow to use distributed analyses?

It is possible to distribute analyses. An existing analyses can be used by the new framework without modifications. However, the new framework has some constraints. Distributed nodes have only one input and one output port each. It is also difficult to start distributed analyses, as the single nodes have to be started manually in the correct order. Furthermore, it is currently not possible to distribute repositories.

Q2.2 Do the modifications improve the performance of an analysis?

The modifications do not improve the performance in our experiments. The MOM is too slow for hundreds of thousands of records. The additional components for the nodes require an increased amount of memory.

It would be possible the achieve performance improvements for very specific analyses with only very few communication between the nodes.

Q2.3 How significant are the modifications with respect to the API?

The creation and distribution of nodes is independent from the usual API. The only modified method is the termination method of the analysis controller. The number of public API classes and methods that changed is therefore one as well.

Q2.4 Is it possible to perform a clean shutdown of a running analysis?

The termination sequence works similar to the one for the concurrent part. It is not possible to terminate a running analysis from the outside. The new termination sequence is furthermore not designed for error cases. Especially crashed nodes could prevent an analysis from terminating.

Conclusion

The new framework has major drawbacks and does not improve the performance. However, as the goal was to support distributed analyses, we can consider the goal as achieved.

Goal Achieved: ✓

Chapter 5

Related Work

In this chapter we present research related to this master's thesis. This mainly includes frameworks for CEP, but also some further work at the software engineering group of Kiel University and some generally related research topics.

5.1 CEP and CEP Related Frameworks

For this thesis, we take various CEP and CEP related frameworks into account to discover the suitability for routing and delivering messages within the distributed part of the architecture. Most of the following frameworks and tools have been suggested in Kieker Project [2013a].

Apache S4

According to its description, the S4 platform [ASF 2013c] provides, amongst others, scalability, faul-tolerance, and a plugin oriented implementation. It is in active development since October 2010 and has already been used in productive systems. S4 is licensed under the Apache 2 License. However, we do not use S4 for this thesis, as the programmers describe the tool as an alpha version. It is also not guaranteed that there is backwards-compatibility until the 1.0 release. Therefore, we use S4 not even as a prototype and are unable to provide further information about this platform.

Siddhi CEP

Siddhi CEP [Siddhi 2012] is classified by the programmers as full CEP engine, licensed under the Apache 2 License. Originally started as an academical research project at the University of Moratuwa, Sri Lanka, the project has later been improved by the WSO2 Inc. under the new name WSO2 CEP. Although the project was being further developed by the middleware production firm WSO2, the usage of the WSO2 CEP turn out to be rather complicated. Due to further technical problems we decide to use another message engine.

Esper

Another suggested framework is Esper [Esper 2013]. It provides an easy usable and intuitive API to use CEP within Java (and .NET). We do not use this framework due to licensing issues. Esper is licensed under the GNU General Public License (GPL), which is incompatible with the Apache 2 License of Kieker [ASF 2013b].

RabbitMQ

The message broker RabbitMQ [GoPivotal 2013] was used during most of the development of the distributed framework. It is licensed under the Mozilla Public License and provides a dynamic exchange system. It is possible to create topics during runtime and use flexible message routing based on routing keys. The RabbitMQ server is written in Erlang [Erl 2013]. The Erlang environment was not available for Solaris though. Due to further technical problems, occurring while compiling the environment, we decided to use ActiveMQ instead.

5.2 Further Work

Master's Project 2013

The master's project 2013 of the software engineering group was performed during this thesis. We can therefore only access internal documents, which are not published yet.

The goal of the project was the combination of cloud computing with Kieker using a master worker pattern. Multiple analysis nodes, representing the workers, would collect various records and assemble partial traces. The number of said nodes would be dynamically scaled in order to support a proper utilization of available capacity. A master analysis node would finally aggregate these partial traces into complete traces [Weißenfels et al. 2013].

The main goal of this project, distributing an analysis, is similar to the one of this thesis. However, there are some differences. The main difference is that the project aims to distribute only a specific trace analysis. In this thesis, we develop a more generic approach to support the distribution of any desired Kieker analysis. Furthermore, the project focuses on a dynamic scaling of the computation nodes. This is a research topic, which is not addressed in this thesis.

Kahn Process Networks

Kahn Process Networks (KPNs) 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 [Kahn 1974; Lee and Parks 1995].

Although being only a MoC, the KPNs 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). Research about KPNs and related MoCs could therefore also be useful for the Kieker framework.

It should be noted though, that our approach is still slightly different than the one of the KPNs. 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 KPNs can only wait for data on one of the available input channels [Kahn 1974]. It is furthermore difficult to formally describe our networks using mathematical equations.

Chapter 6

Conclusion and Outlook

Summary

In this thesis we presented an approach to support concurrent and distributed analyses in Kieker. Our main motivation was allowing online trace analysis. Therefore we outlined the benefits of trace analyses for software engineers during maintenance and development tasks.

We presented various metrics to measure the performance of our framework modifications and discussed some theoretical limits under inclusion of Amdahl's Law. We introduced not only Kieker and its detailed analysis architecture, but also CEP as possible distribution approach.

Our support for concurrent analyses is implemented by using additional FIFO buffers and sender/receiver threads between filters. For the distributed support we decided to use the MOM ActiveMQ for the message delivering instead of actually relying on CEP. We aggregated filters into nodes and enabled to use them in a distributed way. In order to support a termination of the analysis, we implemented an autonomous shutdown based on meta signals.

We measured our approach in various experiments with different metrics. All in all we performed thirteen experiments on blade servers.

Results

The 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 of analyses. The resulting communication overhead is too high. We also realize that the unbounded buffers between the filters lead to disproportionate high memory consumption. Furthermore it seems like repositories are difficult to handle within a concurrently running analysis.

The distributed part of the framework does not lead to any speedup at all within the performed experiments. The communication between the nodes is too slow for practical use.

However, the framework modifications could theoretically be used as a solution for very specific analyses. The usual Kieker analyses are already performant enough. The additional communication overhead would simply not pay off.

Outlook

In order to improve the performance of the concurrent part, we suggest a more suitable data structure for the buffers. Specialized multi-producer one-consumer queues could speedup the framework. It could also be 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 centralized 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. Note also that it would be possible to use a CEP engine or MOM to deliver messages between all available filters instead of aggregate filters into virtual nodes. However, this would probably decrease the performance drastically.

Conclusion

All in all we suggest to carry on with research in the direction of a concurrent framework. A reduction of the communication overhead and the usage of bounded buffers could lead to a performant, scalable, and easy-to-use analysis framework.

Appendix A

DVD

The attached DVD contains the sourcecode, the experimental setups, and the measured data from the experiments. The sourcecode is given in the form of a Git repository. The branch, which has been used for this Thesis, is named "distributedAnalysis". The measured data is given in the form of raw log files and in a modified form.

Bibliography

[Amdahl 1967] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: *Proceedings of the AFIPS Spring Joint Computer Conference*. ACM. Atlantic City, New Jersey, USA, Apr. 1967, pages 483–485. (Cited on page 8)

[ASF 2013a] Apache Software Foundation. ActiveMQ Website. last access: 06.09.2013. URL: `http://activemq.apache.org/`. (Cited on page 34)

[ASF 2013b] Apache Software Foundation. Apache License v2.0 and GPL Compatibility. last access: 25.04.2013. URL: `http://www.apache.org/licenses/GPL-compatibility.html`. (Cited on page 72)

[ASF 2013c] Apache Software Foundation. S4 Website. last access: 25.04.2013. URL: `http://incubator.apache.org/s4/`. (Cited on page 71)

[Basili et al. 1994] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric Approach. In: *Encyclopedia of Software Engineering*. Feb. 1994, pages 528–532. (Cited on page 21)

[Bisbal et al. 1997] J. Bisbal, D. Lawless, B. Wu, J. Grimson, V. Wade, R. Richardson, and D. O'Sullivan. An Overview of Legacy Information System Migration. In: *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC)*. IEEE. Clear Water Bay, Hong Kong, Dec. 1997, pages 529–530. (Cited on page 1)

[Buchmann and Koldehofe 2009] A. Buchmann and B. Koldehofe. Complex Event Processing. *it-Information Technology* 51.5 (Sept. 2009), pages 241–242. (Cited on pages 10 and 12)

[Colombet and Desbat 1998] L. Colombet and L. Desbat. Speedup and Efficiency of Large-Size Applications on Heterogeneous Networks. *Theoretical Computer Science* 196.1–2 (Apr. 1998), pages 31–44. (Cited on page 8)

[Eager et al. 1989] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers* 38.3 (Mar. 1989), pages 408–423. (Cited on page 8)

[Eckert and Bry 2009] M. Eckert and F. Bry. Complex Event Processing (CEP). *Informatik-Spektrum* 32.2 (Apr. 2009), pages 163–167. (Cited on pages 10–12)

[Eclipse Foundation 2013a] Eclipse Foundation. AspectJ Website. last access: 13.05.2013. URL: `http://www.eclipse.org/aspectj/`. (Cited on page 13)

[Eclipse Foundation 2013b] Eclipse Foundation. Eclipse Modeling Framework Project (EMF) Website. last access: 26.05.2013. URL: http://www.eclipse.org/modeling/emf/. (Cited on page 17)

[Erl 2013] Erlang Website. last access: 28.09.2013. URL: http://www.erlang.org/. (Cited on page 72)

[Esper 2013] Esper contributors & EsperTech Inc. Esper Website. last access: 09.09.2013. URL: http://esper.codehaus.org/. (Cited on page 72)

[Focke 2006] T. Focke. Performance Monitoring von Middleware-basierten Applikationen. Diploma Thesis. Unversität Oldenburg, Mar. 2006. (Cited on page 13)

[Gamma et al. 2009] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 37th. Addison-Wesley, Mar. 2009, pages 293–303. (Cited on page 17)

[GoPivotal 2013] GoPivotal. RabbitMQ Website. last access: 09.09.2013. URL: http://www.rabbitmq.com/. (Cited on page 72)

[Gustafson 1988] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM* 31.5 (May 1988), pages 532–533. (Cited on pages 9, 10)

[Hennessy and Patterson 2007] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. 4th. Morgan Kaufmann, 2007, page 39. (Cited on page 7)

[Hohpe and Woolf 2004] G. Hohpe and B. Woolf. Enterprise Integration Patterns. 4th. Addison-Wesley, Aug. 2004, pages 70–77. (Cited on page 14)

[Holten et al. 2007] D. Holten, B. Cornelissen, and J. van Wijk. Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views. In: *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. Banff Centre, Alberta, Canada, June 2007, pages 47–54. (Cited on page 1)

[IEEE and ISO/IEC 2010] IEEE and ISO/IEC. Systems and software engineering - Vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (2010), page 224. (Cited on page 2)

[Kahn 1974] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In: *Proceedings of the 6th IFIP Congress*. Stockholm, Sweden, Aug. 1974, pages 471–475. (Cited on page 73)

[Kieker Project 2013a] Kieker Project. Feature wish: Integrate CEP engine. last access: 04.05.2013. 2013. URL: https://kieker.uni-kiel.de/trac/ticket/809. (Cited on page 71)

[Kieker Project 2013b] Kieker Project. Kieker 1.7 User Guide. Software Engineering Group, Kiel University, Kiel, Germany. Apr. 2013. URL: http://kieker-monitoring.net/documentation/. (Cited on pages 2, 13, 14)

[Kieker Project 2013c] Kieker Project. Kieker Website. Mar. 2013. URL: http://kieker-monitoring.net/. (Cited on pages 2 and 13)

[Kleinrock and Huang 1992] L. Kleinrock and J.-H. Huang. On Parallel Processing Systems: Amdahl's Law Generalized and Some Results on Optimal Design. *IEEE Transactions on Software Engineering* 18.5 (May 1992), pages 434–447. (Cited on pages 8, 9)

[Lee and Parks 1995] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE* 83.5 (May 1995), pages 773–801. (Cited on page 73)

[Luckham and Frasca 1998] D. C. Luckham and B. Frasca. Complex Event Processing in Distributed Systems. *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998). (Cited on pages 10, 11)

[Moe and Carr 2001] J. Moe and D. A. Carr. Understanding Distributed Systems via Execution Trace Data. In: *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*. IEEE. Toronto, Ontario, Canada, May 2001, pages 60–67. (Cited on page 1)

[Rauber and Rünger 2010] T. Rauber and G. Rünger. Parallel Programming for Multicore and Cluster Systems. Ext. Englisch language translation of German language edition: Parallele Programmierung (2nd. edn.) Springer, 2010, pages 163–164. (Cited on page 8)

[Reiss and Renieris 2005] S. P. Reiss and M. Renieris. JOVE: Java as it Happens. In: *Proceedings of the ACM Symposium on Software Visualization (SoftVis)*. Saint Louis, Missouri, USA: ACM, May 2005, pages 115–124, 211. (Cited on page 1)

[Robins 2010] D. B. Robins. Complex Event Processing. Feb. 2010. (Cited on page 10)

[Siddhi 2012] Siddhi Website. last access: 04.05.2013. URL: `http://siddhi.sourceforge.net/`. (Cited on page 71)

[Sun and Chen 2010] X.-H. Sun and Y. Chen. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing* 70.2 (Feb. 2010), pages 183–188. (Cited on page 10)

[Van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Research Report. Kiel University, Nov. 2009. (Cited on page 2)

[VMware 2013] VMware. Sigar Website. last access: 13.05.2013. URL: `http://www.hyperic.com/products/sigar`. (Cited on page 13)

[Weißenfels et al. 2013] B. Weißenfels, S. Mahmens, S. Finke, J. Beye, F. Biß, and E. Koppenhagen. Master's project 2013. Software Engineering Group, Department of Computer Science, Kiel University. 2013. (Cited on page 72)

[Wolf et al. 2004] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces Through Successive Refinement. In: *Proceedings of the 10th European Conference on Parallel Processing (Euro-Par)*. Springer. Pisa, Italy, Aug. 2004, pages 47–54. (Cited on page 1)

[Wong et al. 1995] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural Redocumentation: A Case Study. *IEEE Software* 12.1 (Jan. 1995), pages 46–54. (Cited on page 1)