Non-Intrusive Monitoring of Stream Processing Applications

Michael Vögler*, Johannes M. Schleicher*, Christian Inzinger†, Bernhard Nickel*, and Schahram Dustdar*
*Distributed Systems Group, TU Wien, Austria, {lastname}@dsg.tuwien.ac.at
†s.e.a.l. – software evolution & architecture lab, University of Zurich, Switzerland, {lastname}@ifi.uzh.ch

***Abstract*—Stream processing applications have emerged as a popular way for implementing high-volume data processing tasks. In contrast to traditional data processing models that persist data to databases and then execute queries on the stored data, stream processing applications continuously execute complex queries on incoming data to produce timely results in reaction to events observed in the processed data. To cope with the request load, components of a stream processing application are usually distributed across multiple machines. In this context, performance monitoring and testing are naturally important for stakeholders to understand as well as analyze the runtime characteristics of deployed applications to identify issues and inform decisions. Existing approaches for monitoring the performance of distributed systems, however, do not provide sufficient support for targeted monitoring of stream processing applications, and require changes to the application code to enable the integration of application-specific monitoring data.**

In this paper we present MOSAIC, a service oriented framework that allows for in-depth analysis of stream processing applications by non-intrusively adding functionality for acquiring and publishing performance measurements at runtime, to the application. Furthermore, MOSAIC provides a flexible mechanism for integrating different stream processing frameworks, which can be used for executing and monitoring applications independent from a specific operator model. Additionally, our framework provides an extensible approach for gathering and analyzing measurement data. In order to evaluate our solution, we developed a scenario application, which we used for testing and monitoring its performance on different stream processing engines.

I. Introduction

Modern information systems need to process an ever-increasing volume of data from various sources while providing timely responses to requests from stakeholders. Traditionally, such systems persist incoming data in a database and then execute queries on the stored data to perform required analyses and produce desired results, but are increasingly incapable of coping with the sheer volume of data to process, often making it infeasible to even store all incoming raw data [1]. The requirement for timely responses to complex queries over continuous streams of high-volume data led to the emergence of stream processing systems that do not rely on traditional data processing models. Stream processing systems are designed to continuously process and analyze incoming data streams to produce results in reaction to events observed in the incoming data, as opposed to separately triggered requests to analyze previously stored data.

Due to their intrinsic requirements, performance testing and monitoring [2] of stream processing applications are inherently important for stakeholders to assess and understand the status and runtime characteristics of deployed applications. Since the capabilities of single machines are insufficient for providing the necessary processing power for handling these huge amounts of data, stream processing applications have to scale computations across multiple machines and face the challenge of becoming inherently distributed [3]. In addition to several design and management challenges, this also leads to increased complexity when dealing with performance testing and monitoring. Especially for applications where monitoring was not considered initially, gathering meaningful measurements is challenging.

While there is a host of existing monitoring systems for gathering performance data about applications [4], [5], [6] and their runtime infrastructure [7], [8], to the best of our knowledge, there is no solution that specifically targets stream processing applications and their structure in a way that allows for detailed examination and comparison of their runtime characteristics, independent of the used stream processing framework. Furthermore, emitting data to monitoring systems usually requires code changes in applications that specifically target the used monitoring system.

In this paper, we introduce MOSAIC, a service oriented framework for monitoring the runtime performance of distributed stream processing applications, independent from a particular operator model (e.g., query, graph, or API). In addition, we present an approach that allows adding monitoring functionality to existing JVM-based applications, without changing the applications' code or requiring recompilation. MOSAIC allows for the integration of different stream processing engines for executing and monitoring applications, and provides a flexible mechanism for gathering and analyzing performance measurements based on a generic domain model. We illustrate the feasibility of our approach by monitoring and analyzing the performance of a representative stream processing application deployed on two different stream processing engines.

The remainder of this paper is structured as follows: In Section II we further motivate our work and outline the specific problem and requirements in the context of our work in the smart city domain. In Section III we introduce the MOSAIC framework and accompanying toolset to address the identified problems. We provide a detailed evaluation of framework characteristics and capabilities in Section IV, discuss relevant related research in Section V, followed by a conclusion and an outlook on future research in Section VI.

II. Motivation

The rapid adoption of the smart city paradigm combined with the extensive growth of today's metropolises have led to a significant increase of monitoring and control system deployments [9]. These system penetrate all vital areas of today's cities, including building monitoring and management, traffic control, as well as energy management systems via smart meters. Naturally, these system rely on stream processing applications that allow to rapidly process and react to relevant events that occur in associated, large-volume data sources. Performance, availability, and reliability of these systems has become a critical factor in the operation of modern city infrastructures.

However, the enormous scale combined with the distributed and heterogeneous nature of the deployed stream processing applications poses significant challenges for monitoring mechanisms. Due to the wide variety of available stream processing frameworks and used processing models, finding the right framework for any given task and objectively assessing the resulting application are not trivial. In order to enable a holistic approach to monitor such applications, we argue that the following requirements must be met:

- *Distributed Operation*: A monitoring mechanism must be able to handle the inherent distributed nature of modern stream processing applications. Since single machines cannot provide the necessary processing power for running such applications, computations must be scaled across a distributed infrastructure. Apart from the overhead of managing and provisioning these infrastructures, also monitoring gets more complex as performance measurements of multiple resources have to be considered and appropriately handled.
- *Non-Intrusiveness*: A monitoring mechanism should allow to extend the set of measured metrics beyond traditional performance monitoring, in order to provide an in-depth look into relevant application characteristics without requiring code-level changes, or having to rebuild the stream processing application itself. Furthermore, it should also be possible to acquire measurement data from applications, where monitoring was not considered from the beginning.
- *Model Independence*: A monitoring mechanism should provide means to monitor applications independent from a particular processing model or execution environment, in order to enable an integrated and holistic monitoring concept. Current stream processing frameworks employ various models to define application logic (e.g., queries based on domain-specific languages (DSLs), operator graphs, or programming APIs), which should be supported by the monitoring framework without requiring changes to the application code.

To provide accurate performance monitoring, a solution that respects the requirements defined above should also allow for the analysis of application performance behavior, which comprises the following steps: (i) Acquisition: the process of measuring performance data, (ii) Publication: the process of publishing acquired data, (iii) Management: the process of managing and storing collected data, and (iv) Analysis: the process of extracting information from monitoring data.

III. Approach

In order to address the previously defined requirements, we present MOSAIC, a framework for non-intrusive monitoring of stream processing applications. The overall architecture of our approach is depicted in Figure 1 and consists of the following components: (i) a Stream Processing Environment, (ii) MOSAIC Base, and (iii) MOSAIC, a cloud-based middleware framework. In the following, we discuss these components in more detail, and present the overall approach of weaving, deploying, and monitoring stream processing applications.

A. Stream Processing Environments

Usually stream processing applications consist of various processing steps, such as validation, transformation, aggregation, and analysis [1]. Such applications are used to process data in order to extract or create information. Since stream processing applications have to deal with an ever-growing amount of data, the actual processing work is then distributed across multiple worker resources. In order to manage this distributed processing more efficiently, stream processing applications are deployed and executed on top of stream processing engines that provide seamless provisioning of worker nodes.

To allow our framework to monitor stream processing applications that are executed on top of stream processing engines, we focus on frameworks deployed on the Java Virtual Machine (JVM), such as Apache Spark [10] and Apache Storm [11], [12]. We facilitate aspect-oriented programming (AOP) to weave components of our framework into the actual stream processing application in order to add the necessary monitoring functionality without requiring changes to the underlying application.

Next, to represent a stream processing application and associate respective monitoring data, we introduce a domain model that consists of the following elements. The central element is a `Node`. A node is a representation of a worker performing a specific task in a distributed stream processing application. In order to identify the node, it has a `nodeId`. In addition, a `nodePurpose` attribute describes what the node is actually doing. This approach allows grouping nodes according to their functionality, but also supports distinguishing single nodes. For example, consider an aggregation operation that is intense in computation, and the application requires multiple instances for this specific operation. In such cases, several nodes that share a common purpose are executed, but have different identifiers. Monitoring information that is associated with nodes is commonly referred to as `Measurement`, where we distinguish between `RuntimePerformance` and `JvmProfile`. RuntimePerformance represents runtime measurements and allows monitoring the runtime of one or several operation steps. Furthermore, since an operation can consist of multiple steps, we introduced a `sequence` attribute that connects these records. The JvmProfile is used to monitor resource statistics of the underlying JVM.

B. MOSAIC Base

In order to allow for detailed and application-specific monitoring, we split our framework in two parts. One part, the actual monitoring and analysis framework (MOSAIC) is deployed in the cloud. The other part (MOSAIC Base) is

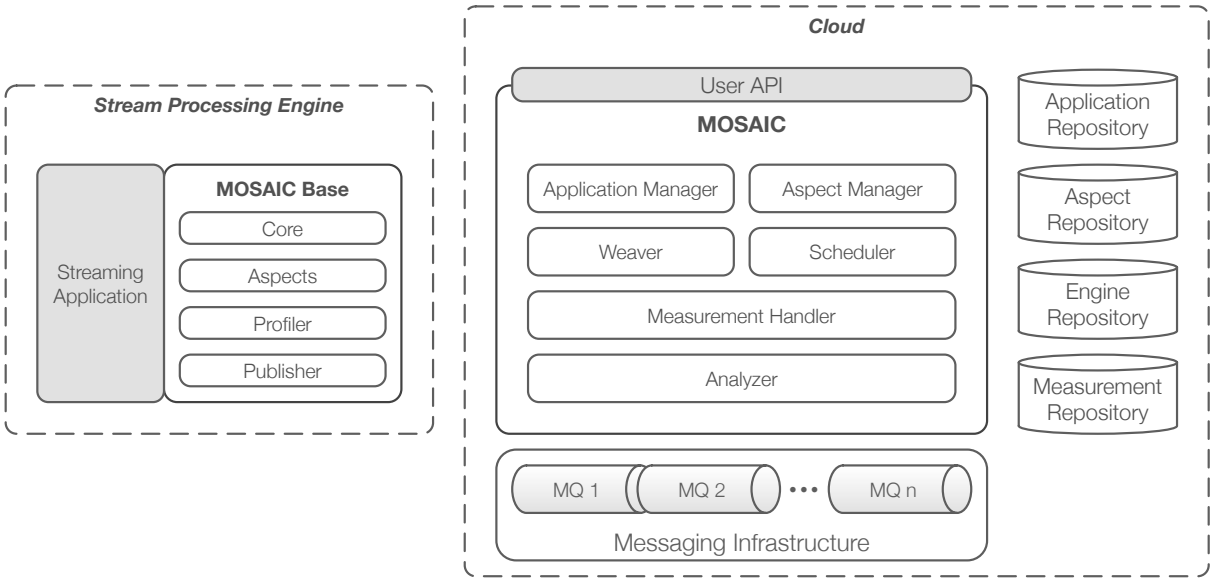


Fig. 1: Framework – Overview

depicted on the left-hand side in Figure 1 and contains the application-specific components that need to be integrated into the application in order to acquire the required monitoring information.

1) Core: The `core` component is the centerpiece of the framework. It contains the domain model, the basic runtime performance measurement functionality, and an abstraction for transferring as well as storing acquired measurement information. Additionally, the component provides out-of-the-box integrations for several stream processing engines. The core component also offers extension APIs to allow for easy integration of additional stream processing engines.

2) Aspects: In order to acquire measurement data, we use aspects respectively advised code that is woven into the target application, as provided by the AspectJ [13] AOP framework. The basic approach of obtaining measurement data consist of the following three steps: First, save the start-timestamp before the monitored code block. Second, save the end-timestamp after the monitored code block is finished. Finally, publish the data. By following the aspect oriented programming principle this can be done using advices of type `Around` (an entire method is wrapped around a pointcut), or `Before` and `After` (two advised methods are invoked) advices. We implemented two different abstract aspects for measuring runtime performance. Additionally, to connect runtime measurements of processing steps (i.e., to establish a sequence of measurements), we need a correlation identifier for a sequence, which must be passed on from one processing step to the next. Since we cannot assume that data that is used within an application provides such a sequence identifier, we add and pass on sequence identifiers within an application. This approach comprises a static crosscutting advice to add a sequence identifier to data objects, and two dynamic crosscutting aspects that create and pass on sequences.

3) Profiler: The `profiler` component contains all classes for monitoring the JVM resources. In addition, the component provides functionality required for profiling JVM resources for a particular code block in order to create execution profiles.

4) Publisher: The `publisher` groups different built-in mechanisms for publishing and storing monitoring information. Currently, we provide functionality to log and persist measurement information for files, JDBC, JMX, and log4j. Data is then distributed via the Java Message Service (JMS).

Since the integration must be as flexible as possible, aspects that acquire performance measurements are woven into the target application. These woven aspects use functionality provided by the core component and publish the measurement data using the publisher component. Once an aspect is woven into the target application, the advised code interacts with the provided functionality of MOSAIC Base when executed.

C. MOSAIC

The enabling framework for monitoring and analyzing stream processing applications is depicted on the right-hand side in Figure 1. MOSAIC is a cloud-based framework and the overall design follows the micro service architecture [14]. This approach enables developing a scalable and evolvable framework. In addition, it allows the flexible management and scaling of components, which is important for MOSAIC when handling stream processing applications deployed at large scale. In the following, we will introduce the main components of MOSAIC.

1) Messaging Infrastructure: For handling the multitude of monitoring information, MOSAIC uses a distributed messaging fabric that minimizes unnecessary network traffic compared to a centralized message bus. Additionally, it allows components of the system to freely choose the most suitable

(e.g., closest) connection point. For transmitting and consuming data, MOSAIC uses a publish/subscribe mechanism, where producing components publish data in the messaging infrastructure to a specific routing key, which represents the application's unique id. Consuming components of MOSAIC, which are mostly analyzer components, consume data by subscribing to the according routing key. This approach provides a flexible and easily extendable mechanism to handle monitoring information.

2) Repositories: In order to manage registered stream processing applications, developed aspects for acquiring measurement data, and monitoring data, MOSAIC provides several repositories.

- **Application repository.** To allow our framework to monitor stream processing applications, operators of such applications need to register them via the provided `User API`. During the registration process operators upload the application as a jar package, add the required runtime configuration, and finally state the intended execution environment (i.e., stream processing engine). Next, based on the stated execution environment, operators can define which monitoring information they are interested in and state the required granularity. For example, operators can define if they are just interested in runtime performance of the application, or also additional JVM-specific profiles. All this application-specific information is managed and stored in the application repository.
- **Aspect repository.** The framework provides abstract aspects that can be used by defining concrete pointcuts. These abstract aspects can be extended to integrate arbitrary execution environments that are used for running stream processing applications. In addition to abstract aspects, MOSAIC also provides built-in integrations for Apache Spark Streaming [10] and Apache Storm [11]. Any additional or new aspects developed and registered by an operator are stored and managed in the aspect repository.
- **Engine repository.** In order to allow our framework to deploy stream processing applications on their intended stream processing engine, we need engine-specific drivers. As for aspects, we define an abstract driver that can be extended to integrate additional, currently not supported, execution environments. The built-in drivers and additional drivers are stored and managed in the engine repository.
- **Measurement repository.** Since our framework does not only allow monitoring of stream processing applications, but also analyzing gathered measurement data, MOSAIC provides a measurement repository for persisting gathered monitoring information.

3) Application Manager: The application manager is responsible for handling registered stream processing applications and associated information in the application repository. Furthermore, during the application registration process, the manager takes care of verifying that the provided information and application package are valid, and if a suitable engine-driver is available. If an engine-driver is available, this driver is then associated with the application and will be used for subsequent deployment. If no suitable engine-driver is present, the operator will be notified and is then required to provide a driver via the user API. Furthermore, to keep track of deployed applications and associated monitoring information, the application manager handles the state of applications.

4) Aspect Manager: Since we want to create and acquire a broad variety of measurement data, MOSAIC needs to provide suitable aspects. To handle these aspects efficiently the aspect manager is used. As already described, MOSAIC provides built-in aspects and allows operators to add their custom-developed aspects by extending the abstract aspects. In addition to aspects, the manager is also responsible for handling and creating configuration files that define pointcuts for concrete aspects. Based on this configuration the `Weaver` component then weaves the aspect code at the defined places in the application to provide the expected monitoring information.

5) Weaver: To provide a flexible and extensible weaving component that allows using different aspect-oriented programming frameworks, we define a weaver interface and provide a concrete implementation that relies on AspectJ [13]. Based on AspectJ, we decided to use load-time weaving by facilitating the Java agent provided by AspectJ. This approach allows us to include the Java agent as an argument during application startup and add a configuration file to the classpath. The Java agent is then responsible for weaving the aspects at application load-time. This weaving option itself causes comparatively little overhead [15], since an advice, once weaved, behaves like a usual method call. However, this weaving approach increases the time needed for starting the application, which is acceptable since we consider long running stream processing applications.

6) Scheduler: The scheduler component is responsible for deploying the stream processing application, the required functionality for weaving, and MOSAIC Base on the actual execution environment. In order to deploy an application and start the monitoring process, the scheduler contacts the application manager and receives the application package and the corresponding information, stating the intended execution environment and required monitoring information. Based on this information, the aspect manager is contacted, which provides the tailored MOSAIC Base containing the required aspects and configurations. Next, the weaver provides the specific functionality that is needed for weaving MOSAIC Base in the application. Finally, the correct engine driver is loaded and the complete package is deployed on the stream processing environment, where the application is started. During startup the weaving process is triggered, which integrates the monitoring-specific code into the application. Once the application is successfully deployed, the scheduler notifies the application manager that the application is running.

7) Measurement Handler: To handle published monitoring information, the measurement handler is used. Based on the currently registered and running applications, the handler starts application-specific data handlers that consume the published data and store them in the measurement repository. This approach allows for flexible and efficient management of measurements.

8) Analyzer: In addition to collecting application-specific performance measurements, we also want to analyze this data to obtain actionable insights. Therefore, we provide an abstract analyzer and corresponding implementations that, based on

our domain model, allows analyzing metrics like runtime per processing step, overall runtime of a sequence of processing steps, and the latency between processing steps.

D. Weaving and Monitoring Approach

Since we cannot show a complete list of provided monitoring capabilities of our framework due to space constraints, we will focus on one specific example and explain how the overall process, from scheduling an application to receiving monitoring data, works.

For this example, we want to measure the runtime of a processing step in an Apache Storm stream processing application. Listing 1 shows a stub for measuring the runtime performance, where an around advice is used. An abstract pointcut, which has to be defined when using this abstract aspect is used for advising the monitoring code. The advice continues the execution at the given join point and measures invocation time.

```
@Aspect
public abstract class
    AbstractRuntimePerformanceAspect {
    @Pointcut
    public abstract void scope();

    @Around("scope() && this(jpo)")
    public Object around(ProceedingJoinPoint
        pjp, Object jpo) throws Throwable {
        ...
    }
}
```

Listing 1: Abstract Runtime Performance Aspect

To use this abstract aspect we now specify that we want to measure the runtime of a processing step (a bolt) in our Apache Storm application. According to this definition, MOSAIC creates a configuration file that defines the pointcut for the aspect, as shown in Listing 2. The concrete-aspect element defines an aspect and a new name for the aspect. Furthermore, the extends attribute defines the abstract aspect. The expected pointcuts of the abstract aspect are set using the pointcut element.

```
<aspectj>
  <aspects>
    <concrete-aspect name="
        BoltRuntimePerformanceAspect" extends="
        AbstractRuntimePerformanceAspect">
      <pointcut name="scope" expression="
          execution(* backtype.storm.topology.
          IRichBolt.execute(..))" />
    </concrete-aspect>
  </aspects>
</aspectj>
```

Listing 2: *aop.xml* example

After defining what and how we want to monitor an application, the framework has to deploy and execute the application on the intended execution environment. As discussed above, the scheduler component is responsible for deploying the application and the additional tailored MOSAIC Base components, as well as the needed weaving functionality, on the execution environment. Next, the scheduler starts the application on the target environment, which triggers the load-time weaving process using the weaving functionality provided by our framework. Once MOSAIC Base is weaved into the target application, the execution is initiated and the application starts processing data. While processing incoming data, the monitored processing step of our target application will be invoked. Since we weaved our framework into the application, we can monitor the processing step as depicted in Figure 2.

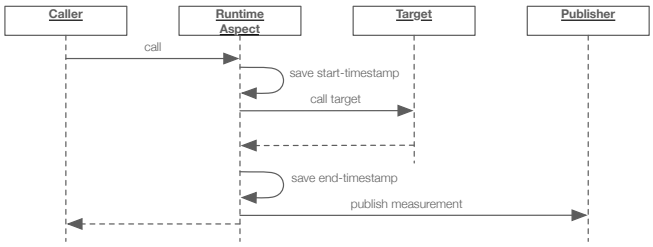


Fig. 2: Monitoring of a Processing Step with a Runtime Aspect

As illustrated in Figure 2 before the actual processing step (target) of our application is called, the runtime aspect is invoked, which saves the start-timestamp and then executes the actual processing step. When the processing step is finished, the runtime aspect saves the stop-timestamp, and publishes the measurement using the publisher provided by MOSAIC Base. The measurement is then consumed on the cloud-based framework, stored in a repository, and can then be further analyzed.

IV. Evaluation

To evaluate our approach we chose a representative scenario and implemented it on top of two stream processing engines. Next, we created a test setup in the cloud using several virtual machine (VM) instances for hosting our framework, as well as the stream processing engines.

In the remainder of this section we give an overview of the chosen scenario and the developed applications, discuss the concrete evaluation setup, present different evaluation scenarios, and analyze the gathered results.

A. Scenario

To illustrate the feasibility of the MOSAIC approach, we will use a modified version of the well-known Traveling salesman problem (TSP) [16] as a scenario application. A TSP graph G is a complete weighted undirected graph specified by a pair (N, d), where N is a set of nodes and d is a function that translates the distance between two nodes in a numerical value. d satisfies two conditions: (1.) Symmetry: $d(i,j) = d(j,i), \forall i,j \in N$. (2.) $d(i,j) >= 0, \forall i,j \in N$. A $path$ of the TSP graph G is a set of edges that describes a path containing each node exactly once (i.e., a Hamiltonian graph). The path distance is the sum of distances of all edges. The solution for our modified traveling salesman problem is a path with the minimal possible path distance.

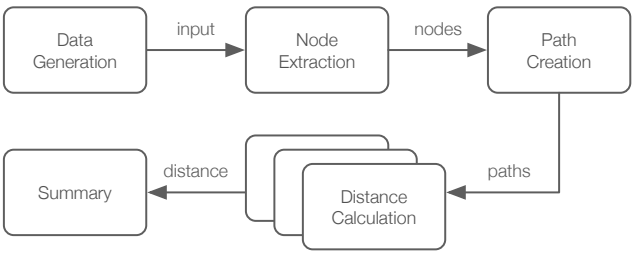


Fig. 3: Sample application – Overview

B. Sample Application

Based on the described TSP scenario we implemented a stream processing application for both, Apache Spark and Apache Storm. In order to make the gathered monitoring results comparable, the implementations are designed to share the same basic architecture.

The overall architecture of the sample application is split into five processing steps as shown in Figure 3. In the first step the input data for the application is created. Processable input data for our scenario is a random string containing vertices in a specified format (i.e., x- and y-coordinates in a two-dimensional space), where before and after each vertex there can be random characters. For the purpose of this scenario, we assume that the coordinate data is embedded in a text document and coordinates must be extracted from the input data in a separate step. The result of this extraction step is a list of Node objects with an id to identify the node, as well as x and y coordinates. In the path creation step, all paths are created. This step also allows splitting paths into subsets to subsequently pass them on to multiple workers for parallel distance calculation. In the last step, the minimal distance of the received set of paths is calculated. Finally, the summary step is responsible for summarizing the partial results of the distance step, which represents the minimal distance of all paths.

C. Apache Spark Streaming Implementation

Following the aforementioned application architecture, we implemented the described scenario based on Apache Spark Streaming. First, we implemented a stream `Receiver` that creates data for the application to consume. For all other steps we implemented `Functions`, which are used by map or flat map transformations and one output operation. Figure 4a shows an overview of our Apache Spark Streaming application. The Receiver creates data, which is stored in Spark RDD's over time. We decided to use a time window so that a RDD is created in a defined interval, containing all data records stored in this time interval. The second step, extraction, is done via a map function, where lists of nodes are extracted from the input data in a RDD. Paths are created by using a flat map transformation. Using an identifier, each path list can be associated to its source node list. The path creation step is followed by a map transformation, which calculates distances for each path list and determines the minimal distance. The summary function determines the absolute minimum distance for an input string by aggregating all received minimum distances with the same identifier.

D. Apache Storm Implementation

As mentioned above, we also implemented the described scenario according to the architecture description using Apache Storm as depicted in Figure 4b. For each step described above, we implemented a Storm-based component. The first step, data creation, is implemented as a `Spout` that creates and emits data to the topology. All other steps are implemented as `Bolts`, linked together using Storms shuffle grouping method. Path bolts are different to other bolts since they might emit multiple tuples, where each tuple contains a set of paths. Path lists can be associated with their source, and thus with each other, by using an identifier.

E. Evaluation Scenarios

In order to gather sufficient monitoring information from our sample application, we defined two scenarios. In the first scenario we executed 25 test runs with a break of 1 second between each run. In the second run, we changed the break to 10 milliseconds. For each run we generated a random input string with a size of approximately 17MB. With these two scenarios we simulate different load patterns for the application to highlight notable differences.

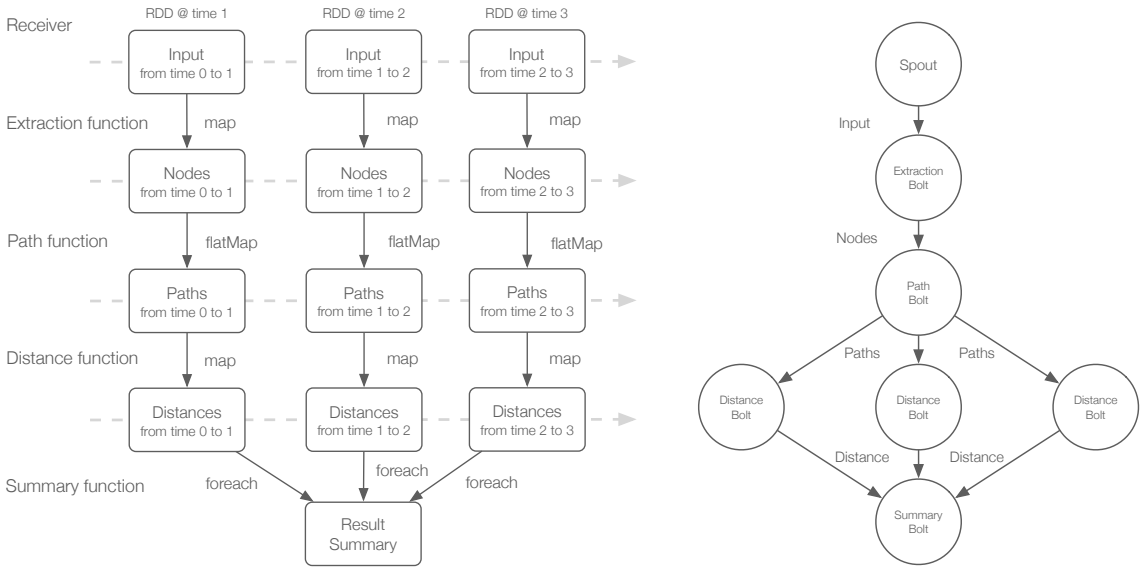
F. Setup

To create analyzable data, we executed both described implementations based on the defined scenarios. Since the main focus of our investigation is on the engineering perspective of how performance can be monitored, and not a performance analysis itself, we used the following setup in our private OpenStack [17] cloud.

MOSAIC is deployed on one instance using the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB disk space). For implementing our messaging infrastructure we use a RabbitMQ [18] cluster consisting of 2 VM nodes using Ubuntu 14.04 and the m1.small flavor (1920MB Ram, 1 VCPUs and 40GB disk space). Next, for hosting the Apache Spark Streaming and Apache Storm engine, we created two separate VM nodes, each using the m2.flavor (5760MB RAM, 3 VCPUs and 40GB disk space). The physical machines hosting the VMs are connected with regular 1000BASE-T ethernet links.

G. Results

In this section, we analyze the measurement data gathered from the test runs. By comparing the results of both implementations, we illustrate the benefit of our framework. The common data model for monitoring different stream processing applications, implemented using different frameworks, allows for a direct comparison of monitored processing steps. Figure 5 shows an overview of runtime measurements for each processing step of our application. The plotted durations (end time − start time) are aggregated by node purpose (i.e., processing step) and node identifier. The node identifier allows to associate a record with the particular run or implementation (10 ms or 1 second break, and Spark or Storm implementation). The



(a) Spark (b) Storm

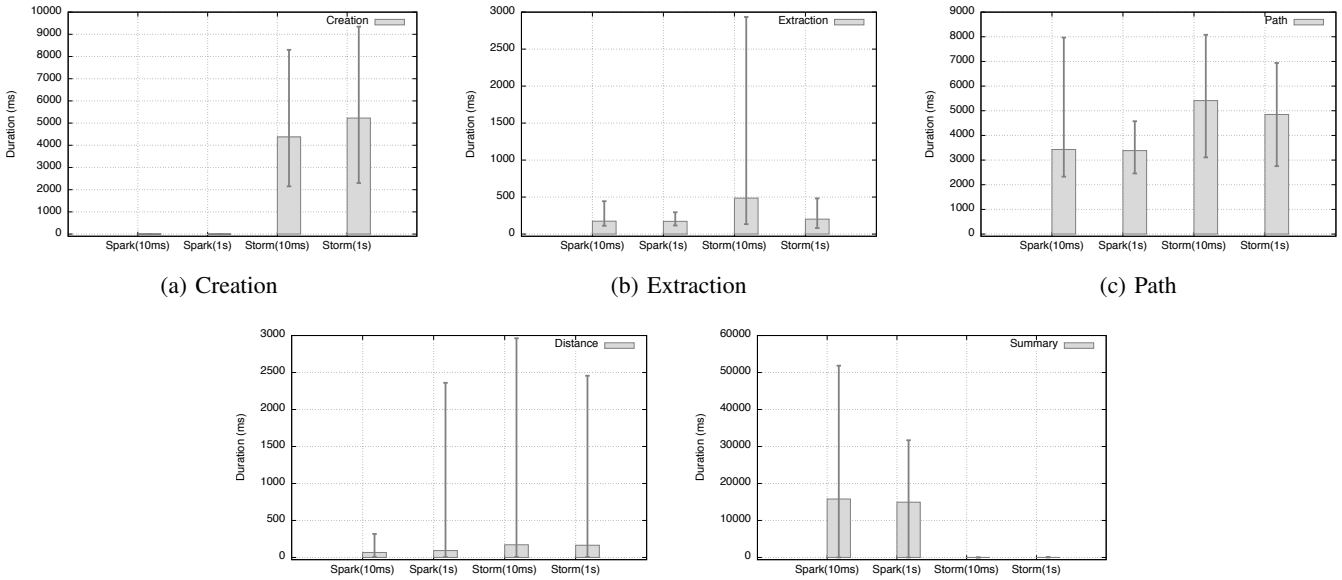
Fig. 4: Sample application – Implementation

different sub-figures show the different processing steps. In the following, we discuss notable results of the comparison between the Spark and Storm application.

In Figure 5a we can see the runtime of the first processing step of our application. By looking at the figure we notice that measuring the invocation time of the store method of a Spark Receiver, does not reflect the time consumed for actually creating, reading or receiving the data. Only the time used for transferring the data to Spark is measured. This explains the large gap when comparing Spark and Storm results of the first step (Creation). Figure 5b, Figure 5c and Figure 5d show the gathered measurement data for the extraction, duration, and distance processing steps. We see that the Spark implementation performs better regarding runtime measurements at these processing steps, compared to the Storm implementation. Figure 5e shows the runtime of the last processing step, namely summary. When comparing the results, we notice a large difference between Spark and Storm. The explanation for this gap is as follows: Spark immediately starts the invocation of output operations for each time slot, even when no data was received within the period of a time slot. However, when no data is received, Spark blocks the output operations function invocation and waits for a considerable amount of time. This behavior distorts the results for the last processing step, since for the first few time slots no data has been passed to the output operation as the path calculations have not been finished. These records have a considerable impact on the aggregated data.

Considering these results, it might appear that Spark performs significantly better. However, simply looking at runtimes of single processing steps does not include the time that has been consumed by the framework or for transferring data from one processing step to another. Using the injected correlation identifiers, runtime measurement records of different processing steps can be connected to each other and the time between the end time of a step and start time of a following step can be determined. Figure 6 depicts the latency between the processing steps.

We notice that Spark exhibits significantly higher inter-step latencies. However, these differences can be explained by the looking at the different processing models used by Spark and Storm. First, Spark creates micro batches over time, which in combination with a window operation means that after the store method of the Receiver is invoked, it can take some time until Spark passes the created data record to the next processing step. Second, since Spark combines data records in RDDs according to the window operation, the number of records that are transferred from the receiver to the first mapping function (processing step) can be considerable in size. This especially applies to the scenario with a data creation delay of only 10 milliseconds. In comparison, Storm emits a tuple as soon as it arrives at its topology. Finally, Spark and Storm employ different task scheduling models. Both engines have a fixed number of task executors. However, Spark reserves task executors for Receivers, which means that a Receiver is running continuously, whereas functions are scheduled and executed when a task executor becomes available. In Storm, Spouts are treated equally to Bolts, which means that their executions are also paused when there are no task executors available. For Spark this means that the extraction step is only executed when a task executor is available, thus processing steps may be paused. In contrast, a Receiver is running all the time, which adds an additional delay between these two processing steps. In the processing model of Storm, where Spouts are scheduled and paused as well, the longest running processing step is the bottleneck, which is the path step in our application. After an extraction task is executed, it might take some time until a task executor becomes available for running the path step of the preceded



(a) Creation (b) Extraction (c) Path

(d) Distance (e) Summary

Fig. 5: Duration per processing step

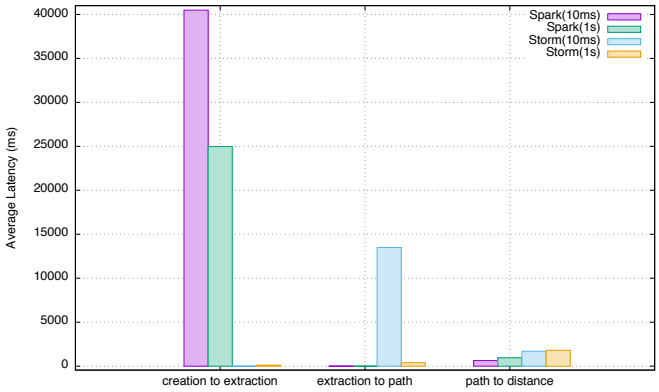


Fig. 6: Latency between processing steps

extraction result, as the task executors might be busy with running other queued path tasks. When all task executors are busy with executing path tasks, no more tuples are emitted by Spouts, thus there is no increased latency between the Spout and the Bolt used for extraction.

To also discuss the total runtime of our application, Figure 7 shows the time between the absolute minimum start time that has been recorded for the creation processing step and the absolute maximum end time of the summary processing step. We notice that the Spark implementation performed better for the test run with the 10 millisecond delay between data creation, whereas the Storm implementation was faster for the test run with the 1 second delay between data creation.

Based on the gathered results, we showed that our approach enables acquiring performance measurements that reflect the differences between the discretized stream processing model of Spark and the continuous operation processing model of Storm. Whereas the advantages and disadvantages of these processing models are not the subject of our approach, the analysis of the gathered results proves the applicability and purpose of our framework.

V. Related Work

With the advent of big data with ever growing data volumes, it is important that applications that deal with this data perform well in order to deliver the intended benefit. Therefore, monitoring of applications is crucial as it enables organizations to analyze and assess the performance of their applications. Ganglia [7] is a monitoring framework for distributed systems. It centrally collects certain metrics, such as CPU usage, memory as well as process information of nodes in a distributed system and allows visualizing collected data. Ganglia is based on a hierarchical design and relies on a multicast-based protocol. In contrast to our approach, Ganglia is strictly bound to a defined list of metrics, which for example does not allow monitoring runtime performance of single process steps of an application. Imamagic et al. [19] present Nagios, a open source solution for monitoring network services in order to detect failures. A service in Nagios can be represented as a host, a network or a service metric (e.g., process runtime). Nagios is built as a distributed system consisting of a server that collects data from sensors by using a plugin. Compared to our approach that allows collecting and analyzing performance measurements, Nagios focuses on states, which means that any performance measurement acquired, must be reduced to a state by defining thresholds for metrics. Additionally, Nagios does not provide any mechanism for monitoring applications that do not measure any performance metrics by default. Di Nitto et al. [20] propose MODAClouds a platform for monitoring that automatically improves quality of service attributes of cloud-based services. The overall approach consists of a monitoring

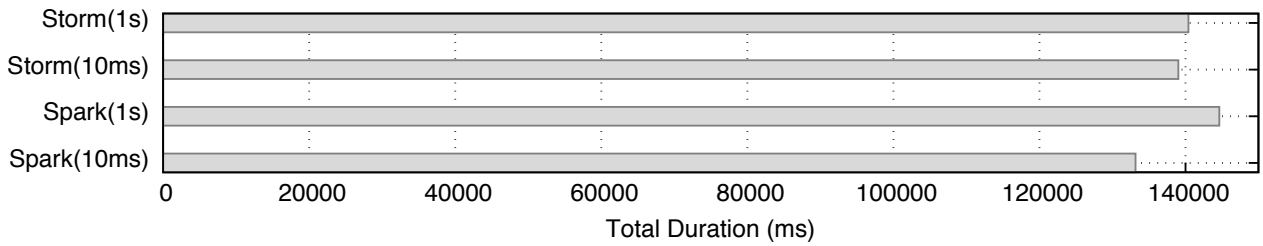


Fig. 7: Absolute total duration ($max(endtime) - min(starttime)$)

platform, a self-adaption platform, and an execution platform. The monitoring platform gathers and analyzes data, collected by data collectors. Compared to our approach, data collectors do not create measurement data, but collect data that has already been created by some other component or application of the system. Moldovan et al. [21] introduce MELA, an approach for monitoring and analyzing elastic services deployed in the cloud. Based on monitored metrics, the authors focus on determining relationships among performance, cost, and resource usage of a service. In order to do that, the authors propose elasticity relationships of elastic services, which are used for applying analyses techniques. However their approach is also capable of collecting and analyzing monitoring data of services respectively applications, the authors rely on data that is either emitted by the service or the execution environment. In contrast, our approach allows collecting and analyzing measurements from applications that do not provide any built-in monitoring capabilities.

Leitner et al. [4] propose an approach for monitoring high-level performance metrics of cloud applications based on complex event processing. The approach is based on a multi-step event correlation approach, which in combination with a hierarchy of predefined events, allows specifying and monitoring application performance metrics. Although the authors provide a flexible and extensible monitoring approach, they solely focus on measurement data created by applications or infrastructure components. Thus, their approach does not allow to monitor applications that do not provide this feature. Yuen et al. [22] present a scalable network for monitoring distributed applications. The network consists of proxies that collect performance data from applications and then report this data to distributed monitors in order to aggregate the data. To reduce the delay caused by the monitoring approach, the authors introduce a monitoring algorithm called SMon, which continuously adapts the network in real-time. However, this work shares similarities with our approach, it focuses on the actual collection of performance data and does not provide a mechanism to acquire and analyze the gathered results. Frischbier et al. [5] discuss ASIA, an approach for monitoring distributed event-based enterprise systems. ASIA provides a mechanism for effectively monitoring the state of a distributed system by dynamically integrating functionality for monitoring into components of the system at runtime, which is based on aspect-oriented programming. Even though this work is similar to our approach, the authors do not provide a flexible and extensible model for defining measurement data. Funika et al. [23] introduce a system for monitoring performance of distributed applications. By using semantic information about monitored components allows automated guidance in order to fine-tune measurements. This can be used for identifying possible performance flaws faster and enables reacting on certain events more efficiently. In contrast to our approach, this work only considers applications that provide built-in monitoring functionality and additionally does not allow distributing and collection measurement data. Li et al. [24] present Sparkbench, a platform specifically built for evaluating Spark-based stream processing applications. Although Sparkbench shares similarities with our approach, it is targeted for Spark and therefore not applicable for other stream processing platforms.

Next to systems that are specifically built for acquiring monitoring information, most stream processing engines already provide built-in monitoring functionalities. Apache Storm [11] provides special bolts to collect and publish metrics [25]. In essence the pre-defined metrics are categorized into system metrics (e.g., memory usage) and topology metrics (e.g., topology statistics such as tuples emitted per minute). For adding custom metrics Storm provides an API. However, compared to our approach, the functionality for acquiring custom metrics would require changes in the application code. Apache Spark [10] provides performance data of system components via Metrics [26]. Although Spark provides mechanisms for publishing measured data to various mediums, the monitoring capabilities are limited to engine-specific components of Spark.

VI. Conclusion

In general, the runtime performance of applications is a crucial aspect, since applications that can not fulfill their performance requirements do not provide their intended purpose. Especially in the era of big data with the ever-growing amounts of data, this is particularly demanding for stream processing applications. In order to allow this type of applications to deal with the immense load, they must be scaled to multiple machines, as single machines can not provide the necessary processing power. Scaling applications appropriately requires actionable performance measurements that need to be acquired by monitoring the application. Monitoring stream processing applications, however, is a challenging task, due to their distributed nature. In addition, stream processing applications often do not provide built-in monitoring functionality that allows gathering and analyzing their runtime performance. Furthermore, traditional runtime environments such as stream processing engines do not allow fine-grained monitoring of deployed applications, but are only capable of providing engine-specific runtime data, which is not sufficient for analyzing the

performance of an application appropriately. This calls for a structured approach that allows non-intrusive monitoring of stream processing applications in oder to acquire application-specific runtime performance data. In this paper, we introduce MOSAIC a cloud-based framework that provides a flexible approach for adding functionality for acquiring and publishing of performance measurements, at runtime to stream processing applications. The framework allows integrating different stream processing engines for deploying and executing applications, a generic domain model for storing and publishing measurements, and a mechanism for gathering and analyzing these measurements. To evaluate our approach, we developed a representative stream processing application, which we used for testing and monitoring its performance by using Apache Spark Streaming respectively Apache Storm as the underlying stream processing engines. Finally, we discussed the gathered results and showed that our approach provides actionable insights on the performance behavior of an application.

In our ongoing work, we plan to extend our framework to address further challenges. We will integrate additional stream processing platforms (e.g., [27]) to see possible limitations and investigate how our approach can be further improved. Additionally, we plan to integrate more sophisticated statistical methods and visualization techniques to provide deeper insights and help drawing better conclusions. Furthermore, based on the analysis of performance measurements, resource planning and stochastic models can be derived to evaluate application behavior under uncertain load. Since many organizations use monitoring tools (e.g., Ganglia [7], Nagios [19] and Splunk [28]) for monitoring applications and IT systems, we plan to provide appropriate interfaces to support these tools and ease integration with our framework. As the performance of stream processing applications also depends on the network performance, we see the necessity to also consider measuring network performance. This network-specific measurement data would allow deeper analysis in general and support root-cause analysis in the case of performance issues. Furthermore, we will integrate our framework with our overall efforts (e.g., [29], [30], [31], [32]) in designing, deploying, and managing complex, large-scale applications in the context of Smart City and IoT [33], to provide a comprehensive tool set for researchers and practitioners.

REFERENCES

[1] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the cloud," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 3, no. 5, pp. 333–345, 2013.

[2] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., feb 2009.

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *Proc. SOSP*, no. 1. ACM, 2013, pp. 423–438.

[4] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, "Application-Level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm," in *Proc. Intl. Conf. on Service-Oriented Computing and Applications*. IEEE, 2012, pp. 1–8.

[5] S. Frischbier, E. Turan, M. Gesmann, A. Margara, D. Eyers, P. Eugster, P. Pietzuch, and A. Buchmann, "Effective runtime monitoring of distributed event-based enterprise systems with ASIA," in *Proc. Intl. Conf. on Service-Oriented Computing and Applications*. IEEE, 2014, pp. 41–48.

[6] NewRelic RPM. [Online]. Available: http://www.newrelic.com

[7] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[8] Dropwizard Metrics. [Online]. Available: http://metrics.dropwizard.io/

[9] R. Kitchin, "The real-time city? Big data and smart urbanism," *GeoJournal*, vol. 79, no. 1, pp. 1–14, 2014.

[10] Apache Spark. [Online]. Available: http://spark.apache.org

[11] Apache Storm. [Online]. Available: http://storm.apache.org/

[12] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, and M. Fu, "Storm@twitter," in *Proc. SIGMOD Intl. Conf. on Management of Data*. ACM, 2014, pp. 147–156.

[13] AspectJ. [Online]. Available: https://eclipse.org/aspectj/

[14] S. Newman, *Building Microservices*. O'Reilly Media, Inc., Feb. 2015.

[15] M. Soares, M. Maia, and R. Silva, "Performance Evaluation of Aspect-Oriented Programming Weavers," in *Intl. Conf. on Enterprise Information Systems*. Springer, 2015, pp. 187–203.

[16] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, "An Analysis of Several Heuristics for the Traveling Salesman Problem," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 563–581, 1977.

[17] OpenStack. [Online]. Available: http://www.openstack.org

[18] RabbitMQ. [Online]. Available: http://www.rabbitmq.com

[19] E. Imamagic and D. Dobrenic, "Grid Infrastructure Monitoring System based on Nagios," in *Proc. Workshop on Grid Monitoring*. ACM, 2007, pp. 23–28.

[20] E. Di Nitto, M. A. A. da Silva, D. Ardagna, G. Casale, C. D. Craciun, N. Ferry, V. Muntes, and A. Solberg, "Supporting the Development and Operation of Multi-cloud Applications: The MODAClouds Approach," in *Proc. Intl. Symp. on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2013, pp. 417–423.

[21] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "MELA: Monitoring and Analyzing Elasticity of Cloud Services," in *Proc. Intl. Conf. on Cloud Computing Technology and Science*, 2013, pp. 80–87.

[22] C. H. P. Yuen and S. H. G. Chan, "Scalable real-time monitoring for distributed applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2330–2337, 2012.

[23] W. Funika, P. Godowski, P. Pegiel, and D. Król, "Semantic-Oriented Performance Monitoring of Distributed Applications," *Computing and Informatics*, vol. 31, no. 2, pp. 427–446, 2012.

[24] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15. New York, NY, USA: ACM, 2015, pp. 53:1–53:8.

[25] Apache Storm Metrics. [Online]. Available: https://storm.apache.org/documentation/Metrics.html

[26] Apache Spark Monitoring. [Online]. Available: http://spark.apache.org/docs/latest/monitoring.html

[27] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, pp. 348–355, 2011.

[28] Splunk. [Online]. Available: http://www.splunk.com

[29] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, and S. Dustdar, "MADCAT - A Methodology for Architecture and Deployment of Cloud Application Topologies," in *Proc. Intl. Symp. on Service-Oriented System Engineering*. IEEE, 2014, pp. 13–22.

[30] M. Vögler, J. M. Schleicher, C. Inzinger, S. Nastic, S. Sehic, and S. Dustdar, "LEONORE – Large-scale provisioning of resource-constrained IoT deployments," in *Proc. Intl. Symp. on Service-Oriented System Engineering*. IEEE, 2015, pp. 78–87.

[31] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "DIANE – Dynamic IoT Application Deployment," in *Proc. Intl. Conf. on Mobile Services*. IEEE, 2015, pp. 298–305.

[32] J. M. Schleicher, M. Vögler, C. Inzinger, and S. Dustdar, "Smart Fabric – An Infrastructure-Agnostic Artifact Topology Deployment Framework," in *Proc. Intl. Conf. on Mobile Services*. IEEE, 2015, pp. 320–327.

[33] J. M. Schleicher, M. Vögler, C. Inzinger, and S. Dustdar, "Towards the internet of cities: A research roadmap for next-generation smart cities," in *Proc. Intl. Workshop on Understanding the City with Urban Informatics*. ACM, 2015, pp. 3–6.