Elastic Stream Processing for the Internet of Things

Christoph Hochreiner, Michael Vögler, Stefan Schulte and Schahram Dustdar
Distributed Systems Group, TU Wien, Austria
{c.hochreiner, voegler, s.schulte, dustdar}@infosys.tuwien.ac.at

***Abstract*—Emerging trends like Big Data and the Internet of Things pose new challenges to established data stream processing engines. Especially, with the advent of the Internet of Things, the data that has to be processed can become very large. Since companies usually aim for cost efficiency, engines need to support resource elasticity to minimize the operational cost while maintaining real-time processing capabilities.**

In the work at hand, we propose and realize the distributed Platform for Elastic Stream Processing (PESP). An extensive evaluation demonstrates the practical feasibility and efficiency of the system design. The evaluation shows that PESP is able to reduce cost by 20% with minimal effects on the Quality of Service in comparison to an over-provisioning baseline. Compared to an under-provisioning baseline, PESP allows a Quality of Service improvement of 72%.

***Index Terms*—Stream processing, Distributed data processing, Hybrid clouds**

I. Introduction

Although data stream processing is an established research area, it gained additional momentum in recent years. The emerging Big Data trend, along with the advent of the Internet of Things (IoT), poses new challenges to traditional stream processing engines (SPEs) like System S [1], Borealis [2] or STREAM [3]. These challenges arise from the large amounts of data, which can be found in social networks [4], as well as in IoT-based scenarios that exhibit a distributed deployment of data sources [5], e.g., in a Smart City scenario. The huge amount of data requires massive parallel stream processing capabilities, which have been tackled by several SPEs, like MillWheel [6], Apache Storm[1] or Apache Spark[2]. These SPEs are designed to process large amounts of data within fixed processing topologies, i.e., a choreography of different data sources as well as stream processing operators like filters, transformations or even complex business logic [7], [8]. However, traditional topologies are provided initially with fixed computational resources and are not able to adapt at runtime.

These fixed resources are likely to render over-provisioning or under-provisioning scenarios for changing streaming data rates [9]. In an over-provisioning scenario, the SPE has more resources at its disposal than it needs to process the average streaming data rate, which renders unnecessary high cost during non-peak times. In an under-provisioning scenario, there are not enough computational resources, which results in delays, i.e., affects the real-time processing capabilities [10].

Even though modern SPEs already resolve several challenges for Big Data scenarios [4], there are still open challenges for SPEs in the context of IoT [9]: In contrast to other stream processing scenarios, e.g., for social networks [4], IoT-based topologies are often deployed across several geographical locations, since the data sources are only available where the observed activities take place.

First, this requires SPEs to support distributed deployment, e.g., across a hybrid cloud, to reduce the data transfer distance between data sources and the stream processing operators [11].

Second, IoT landscapes often evolve over time [9]. These evolutionary changes are triggered by failures, e.g., communication outtakes, or emerging data sources and processing operators. This requires SPEs to support self-configuration and integration of operators at runtime, since it is often not feasible to maintain a centralized configuration node [5].

Third, the IoT implies changing data rates, e.g., an induction loop-based sensor detecting cars driving past provides fewer data in the middle of the night than in the rush hour [12]. If an SPE is not able to adapt to changing data rates, a consistent level of Quality of Service (QoS), e.g., maximum processing time, is not achievable. This requires SPEs to be elastic in terms of processing capabilities.

Fourth, since companies usually aim for cost efficiency, the amount of computational resources that are used for processing streaming data needs to be minimized. This aspect is not only required to minimize the total cost, but also to reduce the ecological footprint of an SPE.

These four major challenges for IoT infrastructures pose new requirements towards SPEs. Up to now, to the best of our knowledge, these challenges were only tackled to some extent. Especially, there is no SPE tackling all of these challenges in a holistic approach (see Section III). In this work, we present the *Platform for Elastic Stream Processing* (PESP) which is suited to realize data stream processing solutions for the IoT. We implement PESP, evaluate it extensively, and demonstrate its capabilities as well as cost efficiency.

The remainder of this paper is structured as follows: First, we will provide a Motivational Scenario in Section II. Then we discuss the Related Work in Section III and present the System Design of PESP in Section IV. Section V presents the testbed-driven Evaluation and Section VI concludes the paper, and provides an outlook on our future work.

II. Motivational Scenario

In the following, we provide a representative scenario to illustrate and motivate our work. For this, we consider a scenario from the transportation domain. In this scenario, the provider of a worldwide operating taxi fleet wants to analyze

[1] https://storm.apache.org
[2] http://spark.apache.org

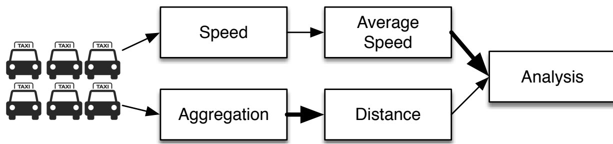

Fig. 1. Motivational Scenario

the rides of its taxis in order to optimize the operational planning in real-time. The taxi provider maintains several fleets in different cities across the world, whereas the operational and analytics center is located in Europe.

To realize a recommendation system for increasing the taxi usage rate as depicted in Figure 1, it is required to process a constant stream of location information from the taxis to derive metrics such as the *average speed* or *distance* of each single ride. These metrics are then *analyzed* to derive optimization measures. Before this can happen, preprocessing activities are needed, e.g., a *speed* calculation between two geolocations and an *aggregation* of all locations for a single ride to derive the distance. Each processing step depicted as a single entity in Figure 1 represents a dedicated stream processing operator for this stream processing topology.

Figure 1 also considers two different types of data flow connections among stream processing operators, which are depicted by fine respectively bold arrows. A fine arrow represents stream processing operations, where the operator emits the same amount of streaming data as it receives. Bold arrows represent aggregation operations, which aggregate the streaming data over a specific time window and emit fewer data after processing compared to the incoming data.

Since the taxis emit a new location information every second, it is necessary to process the data geographically as close as possible to the data providers, i.e., taxis, to reduce the data transfer duration between the data sources and the stream processing operators. Furthermore, there are also other limitations towards the deployment of the stream processing operators, e.g., the analytics operator must only be deployed within a private cloud on the premises of the taxi operator to protect the intellectual property of the algorithm. Last, the SPE also needs to be configurable at runtime to adjust to the ever changing taxi fleet and to allow updating and tweaking the analysis routines without stopping the data stream processing.

III. Related Work

Although Cherniack et al. [15] have outlined the foundations of scalable distributed stream processing already several years ago, to the best of our knowledge, there are still open challenges to realize a reliable and cost-efficient elastic SPE.

In general, the roots for data stream processing are located in the database area. Two of the first SPEs are Aurora [16] and its successor Borealis [2]. While Aurora is designed in a monolithic manner, Borealis already considers a distributed deployment on a cluster system. This allows to increase the fault tolerance of the engine and to provide a higher performance due to parallel data processing [17]. Since Borealis was designed before the appearance of the cloud paradigm, i.e., resource elasticity at runtime [10], it only considers load-shedding mechanisms to cope with changing data rates. Other established SPEs, like System S [1] or STREAM [3] pursue a centralized system design instead of a distributed one.

In contrast to the previously mentioned approaches, there are also more recent engines, which leverage elastic computational resources to process huge amounts of data in real-time, e.g., [6], [18], [19], [20]. A basic system design for integrating cloud resources and data stream processing has initially been proposed by Ishii and Suzumura [21]. Their approach integrates cloud resources to cope with changing data rates and to realize a more cost-efficient data stream processing solution compared to an over-provisioning solution with fixed resources.

While Ishii and Suzumura rely on a hybrid cloud model, there are also several SPEs that are designed to be deployed in a single cloud. The most prominent representatives are the open source engines Apache Storm, Apache Spark and Apache Samza[3]. These SPEs originate from the Big Data domain and are designed to efficiently process huge amounts of data in real-time. Up to now, these SPEs support only fixed stream processing topologies, and each modification of a topology needs to be redeployed with the updated topology configuration. This approach is not feasible for the volatile IoT.

In order to leverage geographically distributed computational resources, Cardellini et al. [14] propose an extension to Apache Storm. Their approach supports an adaptive scheduler, which places operators on the most suitable computational resources. More recent projects, like Spring Cloud Data Flow[4], pick up the concept of individual resource usage for each operator and allow that resources may be scaled individually, depending on the system load. Nevertheless, Spring Cloud Data Flow does not support reconfiguration at runtime nor stateful operators.

Besides open source projects, there are also corporate solutions like Amazon IoT[5] or Google Cloud Dataflow[6], which provide an SPE to connect their existing services and to apply simple queries on data flows. Although these systems provide similar functions as traditional stream processing systems, they operate on numerous topologies in parallel and are able to distribute the load on a large pool of resources [22].

While the scaling mechanisms for stateless stream processing operators are straightforward, there are still several challenges for stateful processing operators, since their state has to be synchronized among replicated operators [15]. Nevertheless, some preliminary work has already been presented in this area. Fernandez et al. [23] suggest a checkpointing mechanism and Gedik et al. [24] propose a key-value store to synchronize the state among multiple operators.

[3]http://samza.apache.org
[4]http://cloud.spring.io/spring-cloud-dataflow/
[5]https://aws.amazon.com/iot/
[6]https://cloud.google.com/dataflow/

TABLE I
COMPARISON OF SPES

	Borealis [2]	IBM System S [1]	StreamCloud [13]	Spring Cloud Data Flow	Apache Spark	Apache Storm	Distributed Apache Storm [14]	Google Cloud Dataflow	AWS IoT
Hybrid clouds				✓			✓	(✓)	(✓)
Reconfiguration at runtime			(✓)						
Elastic resources			✓	✓			✓	✓	✓
Cost efficiency								(✓)	(✓)

Finally, there are also open challenges regarding the scaling policies that are used to elastically scale the computational resources for the SPE. These challenges are picked up by Heinze et al. [25] who discuss the benefits and downsides of different scaling approaches, e.g., threshold-based approaches or scaling based on reinforcement learning.

Table I provides an overview on how existing SPEs cope with the four major challenges introduced in Section I. The checkmarks indicate that the SPE is capable of addressing the challenge. Although all of the SPEs support a distributed deployment, e.g., on a cluster, only some of them support the deployment within a hybrid cloud, which is distributed across more than one geographical location. Altogether, there are already solutions for individual challenges, but a holistic approach considering *all* challenges is still missing.

The table also shows that resource elasticity is available for some SPEs. Only the two commercial providers support cost efficiency. Nevertheless, it has to be mentioned that these providers operate on a large resource pool for numerous users, which makes it difficult to compare their capabilities to other engines that are only deployed for single users. Finally, it can be seen that none of the presented SPEs are capable of dealing with a reconfiguration at runtime, although StreamCloud already provides operation migration techniques which can be used to realize this [13].

IV. PLATFORM FOR ELASTIC STREAM PROCESSING

This section presents the system design of the Platform for Elastic Stream Processing (PESP) and the rationales behind the design decisions.

A. System Design

The system design of PESP aims at addressing IoT specific challenges as discussed in Section I, while respecting the requirements for real-time SPEs defined by Stonebraker et al. [26]. These requirements not only cover the aspects of *keeping the data moving* and to *integrate stored and streaming data*, but also to provide capabilities to *handle stream imperfections* and to deliver *reliable results*.

PESP is a distributed SPE, which consists of arbitrary many *Operator Nodes*. Each of these Operator Nodes represents one specific stream processing operator within a stream processing topology. The topology is configured in a publish-subscribe manner, where each Operator Node subscribes to relevant *Sensors* or other Operator Nodes. This flexible and decentralized configuration approach allows PESP to be reconfigured at runtime in contrast to centralized SPEs, where it is harder to update the topology configuration at runtime. Nevertheless it has to be mentioned that the reconfiguration at runtime also implies organisational challenges, e.g., synchronization challenges for stateful operators. Omitting a centralized orchestration component or message broker also avoids any single point of failure, which is present in established SPEs.

As depicted in Figure 2 (using an FMC Block Diagram), each Operator Node maintains at least one, but up to arbitrary many, *Processing Nodes*. The bold arrows indicate the message flow, while the fine ones represent the control flow within PESP. While the Operator Nodes maintain the communication, state management, and cloud resource management, the Processing Nodes host the Processing Logic, which performs the required operations. Both nodes are composed of several subcomponents, which are discussed in the following.

1) Operator Node:

Incoming Queue: The Incoming Queue is the entry point for streaming data to the Operator Node and is part of the communication infrastructure of PESP. This component obtains data from other data providers, such as preceding Operator Nodes or Sensors, and buffers the incoming data for processing. The buffering capabilities do not only provide the flexibility to cope with changing data rates, but also feature load distribution capabilities among the Processing Nodes. Each Processing Node fetches the streaming data individually and allows for a balanced system load across all Processing Nodes according to their processing capabilities, i.e., available computational resources. The Incoming Queue is the key component regarding the decentralized orchestration of the processing topology at runtime, since the list of data providers can be updated on the fly based on information stored in the Configuration component.

Usage Monitor: The Usage Monitor observes the load of the Incoming Queue, i.e., data throughput and currently buffered data. Furthermore, this component gathers the system load of the Processing Nodes, i.e., CPU and RAM usage, which are provided periodically by the System Monitor of the Processing Nodes. This monitoring data is preprocessed within the Usage Monitor to derive key metrics such as the *average system load/minute* and is used by the Reasoner.

Reasoner: The Reasoner is the core component for optimizing the throughput of data items while minimizing the amount of Processing Nodes to reduce the total cost for data stream processing. Therefore, the Reasoner obtains the usage data from the Usage Monitor and continuously analyzes the current resource usage (see Section IV-D).

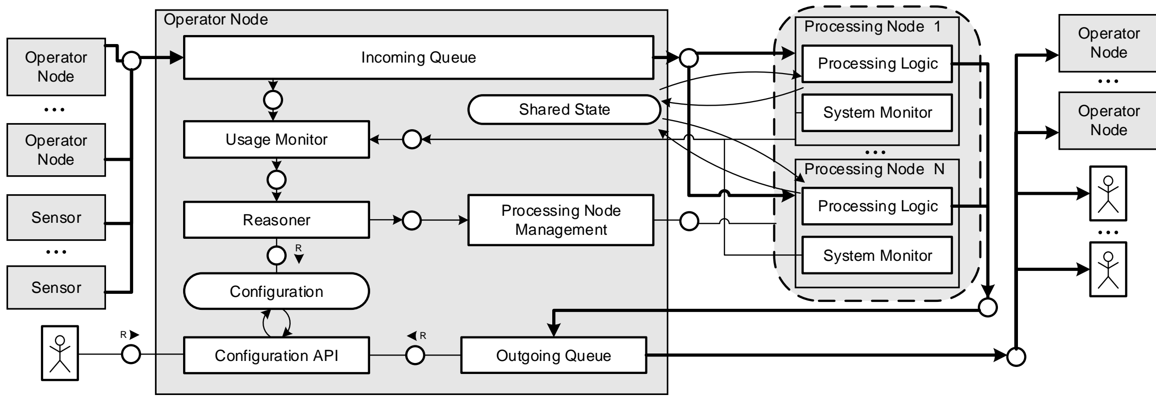

Fig. 2. System Design

Processing Node Management: This component takes care of provisioning new Processing Nodes to ensure the real-time processing capabilities of PESP. Therefore, the Processing Node Management allocates computational resources, e.g., a Virtual Machine (VM), and deploys the Processing Node on these resources. If Processing Nodes are not required anymore, the Processing Node Management informs the selected Processing Node that it will be removed and that the node must not fetch additional streaming data from the Incoming Queue. After a predefined grace period to finish the current stream processing operation, the Processing Node is removed and the respective computational resources are released.

Data Repository: The Data Repository allows the operator to maintain a synchronized state across all Processing Nodes, similar to the concept proposed by Gedik et al. [24]. This synchronized state enables PESP to implement stream processing operations for query windows [18], or to merge semantically annotated data across multiple data items. By using a shared data repository, there is no need to partition the data in advance to realize parallel data processing, as required for other solutions, e.g., StreamCloud [13].

Outgoing Queue: The Outgoing Queue gathers data from all Processing Nodes of a specific operator and forwards the data, based on the configuration, to all subscribed entities of the stream processing topology, e.g., other Operator Nodes or subscribed Users. The Outgoing Queue acts as a short-term buffer to realize a basic fault tolerance mechanism, whenever subscribers are currently not available.

Configuration API: The Configuration API is used to update the topology, i.e., update the list of subscriptions, as well as to update the thresholds, which are used within the Reasoner to tune cost efficiency and QoS at runtime. The data is stored locally on the Operator Node to be either accessed by the Reasoner to configure the elastic resource provisioning or the Outgoing Queue to obtain the recipients of the data.

2) Processing Node:

Processing Logic: The Processing Logic represents the actual processing functionality of the stream processing operator. This functionality comprises simple filtering operations over SQL-like aggregations, or complex business logic [8]. PESP provides a toolkit for developers to access the shared state as well as a basic set of aggregation and filtering functionalities. This eases the implementation of arbitrary stream processing operators to realize stream processing topologies for the IoT.

System Monitor: The System Monitor records the current CPU and RAM usage of the Processing Node and reports this data to the Usage Monitor of the Operator Node in a configurable interval.

B. Data Flow

Within PESP, two layers of data flow are relevant for the system design: The first layer constitutes the data flow among the Operator Nodes. This data flow represents the stream processing topology, where each Operator Node receives data from preceding entities and emits processed data.

The second layer represents the data flow within one Operator Node, as well as among the Operator Node and the associated Processing Nodes. As soon as the Operator Node receives streaming data, the data is buffered in the Incoming Queue of the Operator Node. This streaming data is then fetched from one of the Processing Nodes for processing. When the stream processing operation is finished, the streaming data is transferred back to the Operator Node, i.e., to the Outgoing Queue, to be provided to all subscribed entities.

C. Deployment of Topologies

In order to create a stream processing topology, the user, e.g., the taxi fleet provider discussed in Section II, needs to carry out two steps. First, the user deploys a dedicated Operator Node for each operator of the topology on the desired cloud and defines a specific operator for each Operator Node, i.e., the specific Processing Logic, with the help of the Configuration API. Second, the user needs to configure the topology, by chaining the Operator Nodes together according to the data processing requirements, which is also done by using the Configuration API. As soon as the initial setup and the wiring are finished, PESP automatically spawns the required Processing Nodes and is ready to process the data. Whenever the processing topology needs to be updated, the user can reconfigure the wiring at runtime by invoking the Configuration API.

D. Elastic Resource Provisioning

PESP supports elastic resource provisioning for each operator of the topology based on monitoring information. The aim of our elastic resource provisioning strategy is to minimize the cost for computational resources, i.e., VMs, while processing the data in real-time. This cost optimization is based on leasing Processing Nodes that are required to guarantee real-time processing capabilities and to use the leased resources in the best possible manner according to their Billing Time Unit (BTU). The BTU expresses the minimum leasing duration unit and may differ for each cloud provider. If resources are released before the end of the BTU, this effectively leads to a waste of paid resources. Therefore, it is desirable to only remove those Processing Nodes which have already used most of their BTU.

To realize elastic resource provisioning, we define an optimization problem, which consists of the objective function given in (1) and nine constraints given in (2)–(10).

In our optimization problem, we use the decision variable p_i to denote one particular Processing Node out of the set of all possible Processing Nodes P, which can be obtained by PESP for a single operator. The variable P_U indicates the set of all currently running Processing Nodes. A Processing Node's current CPU load is given by c_{p_i}, its current leasing duration is defined by ld_{p_i}, and we denote the load of the Incoming Queue as q_{in}.

$$\min \sum_{p \in P} p_i + \sum_{p \in P} p_{i_{BTU}} + u \cdot N + d \cdot N \tag{1}$$

The objective function (1), which is subject to minimization, comprises four terms. In the first term we compute the total leasing cost by summing up the assigned values for all Processing Nodes. These values can either take the value 1 when currently leased or the value 0 when not leased, as defined by (8). The second term sums up the remaining leasing time for each Processing Node, which has already been paid according to the BTU, as defined by (6). This term ensures that those Processing Nodes with the smallest remaining usage duration are released first. The remaining two decision variables u (upscaling) and d (downscaling) indicate the required scaling operations. A value of 1 indicates a required scaling operation and the default value of 0 indicates that no scaling operation is required.

$$\sum_{p \in P_U} \frac{c_{p_i}}{P_U} < CPU_{max} + u \tag{2}$$

$$q_{in} < Q_{max} + u \cdot N \tag{3}$$

(2) represents the constraint which triggers an upscaling, whenever the average CPU usage of all running Processing Nodes exceeds the CPU_{max} threshold. The second upscaling constraint, defined by (3), represents the upscaling decisions based on the load of the incoming queue. As soon as the load q_{in} exceeds the threshold of Q_{max}, the variable u becomes 1 and triggers an upscaling operation. The variable N was introduced to decouple the values of u and the values of Q_{max} and q_{in}. Therefore it needs to be an arbitrary large number, which must be larger than any possible value for q_{in}.

$$\sum_{p \in P_U} \frac{c_{p_i}}{P_U} > CPU_{min} - d \tag{4}$$

$$q_{in} > Q_{min} - d \cdot N \tag{5}$$

(4) and (5) represent the constraints for downscaling operations. These constraints work in a similar manner as those for the upscaling operations, but consider the lower thresholds Q_{min} for the Incoming Queue and CPU_{min} for the average CPU usage.

$$p_{i_{BTU}} = \begin{cases} \frac{ld_{p_i} \text{ \% BTU}}{\text{BTU}} & \text{, if } p_i = 1 \\ 0 & \text{, else} \end{cases} \tag{6}$$

(6) defines the remaining usage duration for a specific Processing Node, while respecting the BTU. The result of this constraint is the remaining and already paid usage leasing duration in minutes, while ld_{p_i} represents the time for which the specific Processing Node is already running.

$$\sum_{p \in P} p_i \geq 1 \tag{7}$$

(7) ensures that there is at least one Processing Node running for a specific operator.

$$p_i, u, d \in \{0, 1\} \tag{8}$$

$$0 \leq c_{p_i}, p_{i_{BTU}}, CPU_{max}, CPU_{min} \leq 1 \tag{9}$$

$$P, P_U, ld_{p_i}, Q_{max}, Q_{min} \in \mathbb{N}_0 \tag{10}$$

(8)–(10) define the possible values for the variables.

V. Evaluation

As a proof of concept, we implemented a prototype based on the previously discussed system design and evaluated the prototype in a testbed. The prototype is implemented in Java and reused parts of the software stack of Spring Cloud Data Flow. The Nodes run as web services within Tomcat 7.0.64[7]. The communication infrastructure, i.e., the Incoming Queue and the Outgoing Queue, relies on RabbitMQ 3.2.4[8]. This infrastructure is also used for the communication among the System Monitors of the Processing Nodes and the Usage Monitor. The shared Data Repository is built upon Redis 3.0.4[9].

Each of the Operator Nodes as well as the Processing Nodes are bundled within separate VMs, which eases the deployment on cloud-based computational resources. The deployment functionality of the Processing Nodes is implemented within the Processing Node Management and is capable of deploying Processing Nodes on OpenStack[10] as well as Amazon EC2[11].

[7] http://tomcat.apache.org
[8] https://www.rabbitmq.com
[9] http://redis.io
[10] https://www.openstack.org
[11] https://aws.amazon.com/ec2/

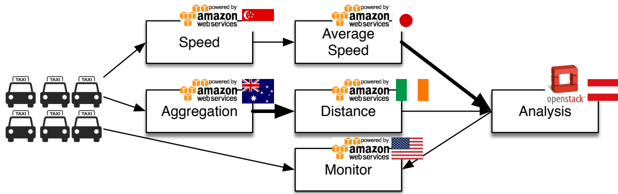

Fig. 3. Evaluation Scenario

TABLE II
RESOURCE SETUP

	Speed	Average Speed	Aggregation	Distance	Analysis	Monitor
Under-provisioning	5	6	2	1	1	1
Over-provisioning	8	10	3	1	1	1

A. Setting

1) Streaming Data: In order to evaluate PESP, we picked up the motivational scenario as described in Section II. Therefore, we selected 75 rides from the T-drive trajectory data sample [27], which provides GPS trajectories within Beijing. Each of these rides consists of multiple locations with a timestamp and a unique ID for each ride. This allows to replay the rides over a given time span in order to obtain a real world data stream. The streaming data exposes changing data rates, since the rides took place across the course of a week and require a reconfiguration of the SPE to consider all rides at runtime. In our evaluation, we replayed the data according to the actual timely sequence of events within 110 minutes for each evaluation run. To evaluate the motivational scenario, we introduced an additional monitoring operator to the topology to observe the data flow of the SPE. The revised topology is depicted in Figure 3. The streaming data is provided to the Speed, Aggregation, and Monitor operators at the same time.

2) Thresholds: For our evaluation, we chose the following concrete values for the elastic resource provisioning strategy: First, we enabled PESP to allocate a maximum amount (P) of 50 VMs for each operator, whereas the BTU for each VM is 60 minutes based on the pricing model of Amazon EC2. The scaling thresholds for the Incoming Queue are set to 5 for scaling down (Q_{min}) and 150 for scaling up (Q_{max}), respectively 90% (CPU_{max}) and 10% (CPU_{min}) for the thresholds for the average CPU usage of the Processing Nodes.

3) Baselines: To evaluate the elasticity aspects of PESP, we selected two baselines. These two baselines represent an under-provisioning as well as an over-provisioning scenario with a fixed amount of processing nodes, as listed in Table II. These fixed resource baselines represent the deployment for current state-of-the-art SPEs. Although there are already some autoscaling approaches available for Amazon IoT or Google Cloud Flow, we opted for a fixed baseline, since these services apply a pricing scheme based on the amount of processed items rather than on the actually used computational resources. Each Operator Node has an individual amount of Processing Nodes, due to different workload within the topology as well as the complexity of the stream processing operator. The baseline configuration has been selected based on the minimal respectively maximal usage within the elastic scenario based on our resource provisioning strategy (see Section IV-D).

4) Testbed: The evaluation was carried out within Amazon EC2 as well as a private OpenStack cloud. The selection of the EC2 regions is based on the motivational scenario, as one can see in Figure 3 indicated by the flags. The *Speed* and *Aggregation* operators are deployed in the Singapore respectively Sydney region of Amazon EC2. Furthermore, the *Average Speed* operator is deployed in the Tokyo region, the *Distance* operator is deployed in the Ireland region, and the *Monitor* operator is hosted in the Oregon region. The *Analysis* operator is deployed on a private OpenStack instance within Europe as already discussed in Section II. In terms of size, we have used *t2.small* instances for the Operator Nodes and *t2.micro* instances for the Processing Nodes on Amazon EC2. VMs on OpenStack implement the same configurations as *t2.small* and *t2.micro* on Amazon EC2.

5) Metrics: To assess the functionality as well as the total cost for our evaluation scenario, we have used different metrics. Since PESP aims not only at providing a distributed SPE, but also at reducing the total execution cost by applying an elastic provisioning strategy, we have assessed the *Cost for Processing Nodes*. This metric aggregates the number of Processing Nodes assigned to all Operator Nodes over the whole evaluation run, where one Processing Node amounts for one cost unit for each minute running. We further assess the *Total Makespan in Seconds*, which represents the time span between the first location recorded by the Monitor operator until the last report is recorded. This allows us to assess the overall stream processing performance.

The *Average Duration for the Report Generation in Seconds* describes the duration which passes between the last location information of a single ride and the issuing of the report. This metric allows us to assess the real-time processing capabilities of PESP. Based on the previous metric we also assess the QoS, i.e., *Total Delays*, by applying a Service Level Agreement (SLA) to the report generation process. We assume that the report needs to be finished within 60 seconds after the last location is recorded, i.e., the *Average Duration for the Report Generation in Seconds* has to be lower than 60. This further results in the *SLA Adherence* that describes how many delays were recorded in relation to the total amount of rides.

B. Results and Discussion

In order to evaluate our prototype, each provisioning scenario was executed three times over two days. This was done to reduce the risk of any corruption of the results, which may occur due to different system loads as well as communication channels among the different regions in Amazon EC2. The raw data of the individual evaluation runs is available in our evaluation repository[12].

[12]http://www.infosys.tuwien.ac.at/staff/hochreiner/evaluation/cloud2016/

TABLE III
EVALUATION RESULTS

	Elastic Provisioning	Over-Provisioning	Under-Provisioning
Total Rides	75	75	75
Location Information Items	50742	50742	50742
Cost for Processing Nodes	2160.66 (σ = 13.61)	2664 (σ = 0.00)	1856 (σ = 0.00)
Total Makespan in Seconds	6653 (σ = 9.60)	6655 (σ = 0.00)	6975 (σ = 1.00)
Average Duration for the Report Generation in Seconds	77 (σ = 10.69)	35 (σ = 0.00)	355 (σ = 0.57)
Total Delays	21 (σ = 5.29)	0 (σ = 0.00)	75 (σ = 0.00)
SLA Adherence in %	28.00 (σ = 7.40)	100.00 (σ = 0.00)	0.00 (σ = 0.00)

Table III lists the average values for all three evaluation runs alongside with the standard deviation σ. Figure 4 presents the number of Processing Nodes across the evaluation alongside with the incoming rate of streaming data. While Figure 4 presents the resource usage of all operators combined, Figure 5 presents the resource usage as well as the load of the Incoming Queue for the Speed Operator Node for the elastic provisioning scenario. In both figures, the horizontal axis represents the time in minutes. The vertical axis represents the total number of Processing Nodes on the left side and the amount of incoming streaming data, as well as the buffered streaming data for Figure 5 on the right side.

The evaluation shows that the system behavior follows a predictable outcome as required by Stonebraker [26]. This outcome is derived from the low standard deviations across the different evaluation runs, despite the fact that the evaluation was carried out in a public cloud environment.

Furthermore, the evaluation also shows the relation between the amount of computational resources, i.e., Processing Nodes, and the total makespan. The under-provisioning scenario exposes the longest makespan, while the total makespan for the over-provisioning and the elastic provisioning scenario are almost the same. This additional required time can be explained due to a shortage of Processing Nodes in the under-provisioning scenario compared to the over-provisioning one. The elastic provisioning scenario only requires 15% more resources than the under-provisioning scenario, while performing as fast as the over-provisioning one. This fact can be attributed to the elastic scaling mechanism, which only allocates additional Processing Nodes when they are required. This can also be seen in Figure 4, which represents the total amount of Processing Nodes over time. Figure 5 provides an even better representation of the relation between the streaming data rates and the amount of Processing Nodes. Each time, the streaming data rates rise, the amount of Processing Nodes also increases to cope with the increased system load.

Since PESP applies a threshold-based scaling approach, it takes some time, i.e., 60-200 seconds, for the SPE to fully adapt to the increased system load. This delay is caused by reasoning the resource provisioning as well as the setup time of the Processing Nodes, i.e., system startup of the VM. Although the resource provisioning represents an optimization problem, its solution amounts only for a small fraction of the overall delay due to the typically small number of Operator

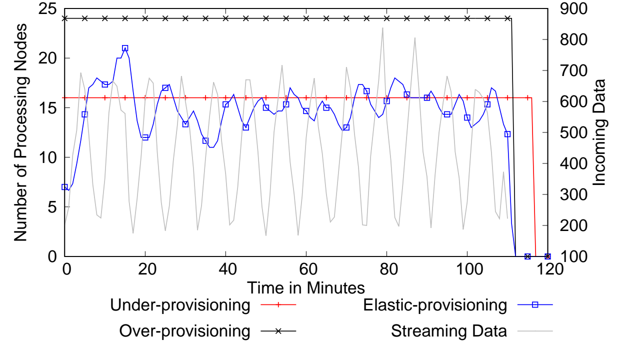

Fig. 4. Resource Usage

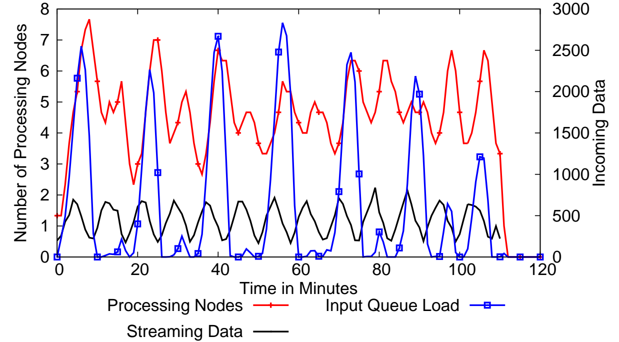

Fig. 5. Ressource Usage of the Speed Operator Node

Nodes. This can be observed in Figure 5, where the Incoming Queue buffers the streaming data, until the Operator Node adapts to the changing streaming data rates. Nevertheless, it takes PESP only twice as long to issue a report compared to the over-provisioning scenario, whereas the under-provisioning scenario requires ten times as much time. This observation is also supported by the total delay as well as the SLA adherence metric shown in Table III, where we can see that the over-provisioning scenario has no issues with complying with the applied SLAs. The elastic provisioning scenario has an SLA adherence rate of 28%, although the report generation took on average only 17 seconds longer than required by the SLA. In the under-provisioning scenario, we can observe an SLA adherence of 0%, i.e., no report was issued in time, which is consistent with the other observations.

Our evaluation shows that it is possible to implement a distributed streaming topology where the stream processing operators are distributed across several regions.

Additionally, it also demonstrates that the elasticity mechanism allows for a cost-efficient realization of a stream processing scenario while being compliant with applied SLAs.

Furthermore, the evaluation shows that PESP provides a solution approach that addresses the major challenges discussed in Section I. Due to the decentralized system design of PESP, it is easy to deploy different nodes in different clouds to support the first challenge of hybrid cloud deployments. Our system design allows for reconfiguration at runtime, which is currently not possible for other SPEs. Finally, our approach also allows for elastic resources and implements a resource allocation algorithm that aims for minimal cost while maintaining a high level of QoS that resolves both challenges of resource elasticity and cost efficiency.

VI. Conclusion

Within this paper, we have shown that it is possible to realize distributed stream processing scenarios, which are typical for the IoT. We further introduced a resource elasticity mechanism to deal with changing rates of streaming data. This allows us to operate a cost-efficient SPE due to a flexible adoption of Processing Nodes.

In our future work, we would like to further investigate resource provisioning approaches. Threshold-based scaling algorithms can only react based on the current system load, and therefore the reaction is often slightly delayed. To cope with these issues, we plan to investigate other scaling approaches, e.g., to predict the future amount of streaming data based on pattern recognition. We further plan to design and implement a control channel alongside the data channel, to propagate the system requirements across the whole SPE in a peer-to-peer manner.

Acknowledgment

This paper is supported by TU Wien research funds. This work is partially supported by the Commission of the European Union within the CREMA H2020-RIA project (Grant agreement no. 637066).

References

[1] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, “SPC: A distributed, scalable platform for data mining,” in *4th Int. Workshop on Data mining standards, services and platforms*, 2006, pp. 27–37.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, and E. Ryvkina, “The Design of the Borealis Stream Processing Engine,” in *Conf. on Innovative Data Systems Research*, 2005, pp. 277–289.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, “STREAM: the stanford stream data manager (demonstration description),” in *2003 ACM SIGMOD Int. Conf. on Management of data*, 2003, pp. 665–665.

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

[5] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, “Internet of things: Vision, applications and research challenges,” *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[6] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, “Millwheel: fault-tolerant stream processing at internet scale,” *Very Large Data Bases (VLDB) Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[7] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, “Elastic scaling of data parallel operators in stream processing,” in *Int. Symp. on Parallel & Distributed Processing (IPDPS)*, 2009, pp. 1–12.

[8] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, “SPADE: The System S Declarative Stream Processing Engine,” in *2008 ACM SIGMOD Int. Conf. on Management of Data*, 2008, pp. 1123–1134.

[9] C. Hochreiner, S. Schulte, S. Dustdar, and F. Lecue, “Elastic Stream Processing for Distributed Environments,” *IEEE Internet Computing*, vol. 19, no. 6, pp. 54–59, 2015.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[11] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, “Network-aware operator placement for stream-processing systems,” in *22nd Int. Conf. on Data Engineering, 2006 (ICDE)*, 2006, pp. 49–49.

[12] F. Lécué, S. Tallevi-Diotallevi, J. Hayes, R. Tucker, V. Bicer, M. L. Sbodio, and P. Tommasi, “Star-city: semantic traffic analytics and reasoning for city,” in *19th Int. Conf. on Intelligent User Interfaces*, 2014, pp. 179–188.

[13] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, “Streamcloud: An elastic and scalable data streaming system,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[14] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, “Distributed QoS-aware scheduling in Storm,” in *9th ACM Int. Conf. on Distributed Event-Based Systems*, 2015, pp. 344–347.

[15] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, “Scalable distributed stream processing.” in *Conf. on Innovative Data Systems Research*, vol. 3, 2003, pp. 257–268.

[16] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik, “Retrospective on aurora,” *The Very Large Data Bases (VLDB) Journal*, vol. 13, no. 4, pp. 370–383, 2004.

[17] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. Zdonik, “Load management and high availability in the borealis distributed stream processing engine,” in *GeoSensor Networks*, 2008, pp. 66–85.

[18] 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.

[19] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, “Esc: Towards an elastic stream computing platform for the cloud,” in *IEEE Int. Conf. on Cloud Computing (CLOUD)*, 2011, pp. 348–355.

[20] T. Heinze, “Elastic complex event processing,” in *8th Middleware Doctoral Symp.*, 2011, pp. 4:1–4:6.

[21] A. Ishii and T. Suzumura, “Elastic stream computing with clouds,” in *IEEE Int. Conf. on Cloud Computing (CLOUD)*, 2011, pp. 195–202.

[22] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *10th European Conf. on Computer Systems*, 2015, p. 18.

[23] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating scale out and fault tolerance in stream processing using operator state management,” in *2013 ACM SIGMOD Int. Conf. on Management of Data*, 2013, pp. 725–736.

[24] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, “Elastic scaling for data stream processing,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[25] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, “Auto-scaling techniques for elastic data stream processing,” in *30th Int. Conf. on Data Engineering Workshops (ICDEW)*, 2014, pp. 296–302.

[26] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[27] Y. Zheng, “T-drive trajectory data sample,” August 2011. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=152883