VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things

Christoph Hochreiner, Michael Vögler, Philipp Waibel and Schahram Dustdar
Distributed Systems Group, TU Wien, Austria
{c.hochreiner, voegler, p.waibel, dustdar}@infosys.tuwien.ac.at

***Abstract*—The Internet of Things is getting more and more traction, nevertheless, state-of-the-art approaches only focus on specific aspects, like the integration of heterogeneous devices or the processing of sensor data emitted by these devices. However, such domain-specific approaches slow the adoption rate of the Internet of Things, because users need to select and integrate different approaches in order to build a solution that fits all their requirements.**

To resolve this shortcoming, we have designed and implemented the VISP ecosystem, which provides a holistic approach for elastic data stream processing in Internet of Things scenarios by supporting the complete lifecycle of designing, deploying, and executing such scenarios. VISP further tackles the challenges of data privacy as well as software reuse, including monetization aspects in today's service landscapes.

This paper analyzes challenges for creating solutions for the Internet of Things, presents the VISP ecosystem, and discusses its applicability for use case specific data stream processing topologies.

***Index Terms*—Distributed Information Systems, Hybrid Clouds, Data Stream Processing, Topology Design**

I. Introduction

In the last couple of years, not only the number of Internet of Things (IoT) devices, but also their application areas increased dramatically. While the first IoT devices were only considered as technological proof of concepts, they are constantly evolving to become suitable for day-to-day application scenarios. Up to now, most IoT devices are only used in point-to-point scenarios, where the sensor data of IoT devices is only processed by one software service, e.g., adaptive lighting based on the current content on a TV screen. Nevertheless, it is necessary to take the communication and data processing capabilities to the next level by integrating heterogeneous IoT devices with multiple software services [1]. Therefore it is required to design appropriate toolkits that allow for easy integration of IoT devices with software services in order to create data stream processing topologies. Such a stream processing topology is a choreography of different data sources and data stream processing operators, like filters, transformations or even complex software services [2].

Currently, there are already several IoT platforms available that support the integration of IoT devices. Nevertheless, most of these platforms have deficiencies such as missing support for heterogeneous IoT devices and untouched challenges concerning the privacy of the processed data [3].

Additionally, state-of-the-art IoT platforms do not sufficiently support complex data processing topologies and do not provide means for reusing processing operators. Traditional providers of most IoT platforms are IoT device manufacturers with the primary goal of processing sensor data of their devices. However, they do not support any complex processing topologies that allow integrating multiple heterogeneous IoT devices.

When it comes to creating complex topologies, these IoT platforms are only suitable for preprocessing data, i.e., transforming raw sensor data into a machine-readable format. For all other processing steps, it is inevitable to fall back to established Stream Processing Engines (SPEs) like Apache Storm[1] or Apache Spark[2] [4]. Although these SPEs excel at processing streaming data, they are inflexible in terms of resource elasticity or topology reconfiguration at run-time and are currently not suitable for a hybrid deployment [5].

Since complex topologies often integrate IoT devices that are situated at different geographical locations, it is often necessary to perform operations, such as filtering data, close to the data source. This hybrid deployment reduces the amount of data that has to be transferred among the different geographic locations and ensures real-time data processing [6]. Nevertheless, a single access point, i.e., user interface, which handles the complexity of the hybrid deployment transparently is vital. It is also essential to provide a graphical user interface that is easy to use for designing topologies. Furthermore, it is required to devise strategies to reuse already existing building blocks, i.e., stream processing operators, for future topologies to reduce the workload of creating use case specific topologies.

To address the deficiencies of state-of-the-art IoT platforms, especially in terms of data stream processing, we propose the VIenna ecosystem for elastic Stream Processing (VISP). In this paper, we present the design of VISP and discuss its capabilities by evaluating a real-world industry scenario.

The remainder of this paper is structured as follows: First, we provide a motivational scenario in Section II. Based on motivational scenario, we then discuss the challenges for realizing use case specific topologies in Section III. In Section IV we present the related work and in Section V we present the system design of VISP. We evaluate the feasibility of the VISP ecosystem based on a use case evaluation in Section VI, discuss the evaluation regarding the identified challenges in Section VII, and conclude the paper in Section VIII.

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

978-1-4673-9885-5/16/$31.00 ©2016 IEEE

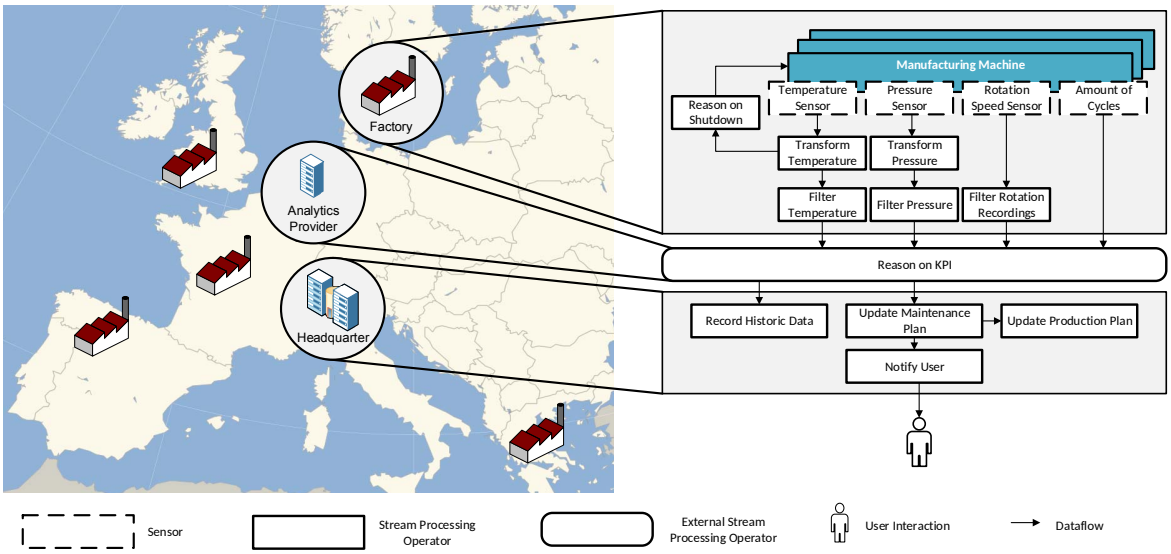

Fig. 1: Motivational Scenario

II. Motivational Scenario

In the following, we are going to describe a representative scenario to motivate and illustrate current challenges for data stream processing. For this, we consider an example from the smart factory domain, where the owner of multiple factories wants to implement a centralized predictive maintenance solution for all manufacturing machines. These machines are situated at different factories throughout Europe, as depicted in Figure 1.

The predictive maintenance solution aims at minimizing the downtime of machines by replacing expendable parts before their estimated operating time to eliminate the down time of manufacturing machines. The failure prediction of expendable parts is a nontrivial task, which builds on the information of several Key Performance Indicators (KPIs) originating from sensor data. These KPIs are then processed by several operators that form a topology. The topology on the right-hand side of Figure 1 depicts the use case specific topology, which is composed of the data sources as well as the required operators for the predictive maintenance scenario.

A. Predictive Maintenance Scenario

First, each manufacturing machine exposes data from three distinct sensors: a temperature sensor, a pressure sensor as well as a rotation speed sensor, and a registry, which counts the amount of all production cycles for this manufacturing machine. Second, the data of some of these sensors, i.e., for the temperature sensor and pressure sensor, is only provided in a proprietary format, and it is required to transform the data into a machine-readable representation. This step is carried out by dedicated transformation operators, as depicted in Figure 1. The machine-readable data, i.e., the KPIs, are then filtered by specific filter operators to discard faulty sensor readings. Besides the filtering, the temperature data is also forwarded to a *Reason on Shutdown* operator, which shuts down the machine as soon as a specific temperature threshold is reached.

The KPIs are then forwarded to a *Reason on KPI* operator for analysis. Since this analysis step is a nontrivial task, it is required to integrate a suitable external operator for this functionality. This operator is typically hosted within the premises of the analytics provider since they do not want to reveal their algorithms.

The result of the reasoning step is processed further in the headquarter by an operator, which stores the reasoning information for future usage, and an operator that updates the maintenance plans for machines in the factories. The result of this updated maintenance plan is also forwarded to dedicated operators to update the future production plan as well as to inform the user about future maintenance operations.

Although each operator within this data stream processing topology has a vital role for the overall result, each operator requires a different level of Quality of Service (QoS) regarding its data processing.

The *Reason on Shutdown* operator, for example, is required to process the data in real-time to mitigate any serious issues, e.g., the machine is starting to overheat, while the *Record Historic Data* operator does not require real-time processing capabilities to store the data. Therefore, each operator is assigned a set of Service Level Agreements (SLAs), which influence the deployment location of the operator, i.e., to minimize the data transfer duration among different operators, as well as the maximal possible delay for processing the data. Figure 1 suggests an optimal deployment for this scenario, where data-intensive operators for each machine are located redundantly in the factories across Europe close to the manufacturing machines and the analytics are located in the headquarter. To minimize the implementation effort for topologies, it is desirable to integrate already existing data stream processing operators as building blocks and only implement those operators, which are unique for the predictive maintenance scenario.

B. Stream Processing Operator Categories

Based on the motivational scenario, we have identified three different data stream processing operator categories:

a) Pre-built operators: Operators of this category are already implemented and available to be integrated into use case specific topologies. These operators are either free of charge, e.g., transformation operators, which are provided by the producers of the IoT devices, or generic operators. Generic operators, like filters, are provided by third party developers and can be obtained by paying a one-time fee to the operator developer. Pre-built operators can be seen as a software library since they are integrated into the data stream processing topology and the user has full control over these operators including their resource allocation and QoS.

b) External operators: The second type of operators is represented by external services that can only be integrated on a Software as a Service (SaaS) basis. For this kind of operators, the user cannot control the QoS. Furthermore, these operators typically imply a pay-per-use policy, where the provider of the operator charges the user for each data item processed. The main reason for providing operators in a SaaS manner is that the owner is not able to give away the code or binary of these operators either due to business related reasons or due to legal aspects. Typical examples for such operators are reasoning algorithms, e.g., on KPIs, where the owner does not want to reveal the intellectual property of the algorithm.

c) Domain specific operators: The last type of operators represents operators that represent business logic that is unique for use case specific topologies, like the *Update Production Plan* operator. Due to their uniqueness, they are not available upfront and need to be implemented by the user for use case specific topologies.

III. Challenges

In addition to the challenges presented by Mineraud et al. [3], e.g., the support of heterogeneous devices, data ownership or data fusion, we have identified further research challenges for the realization of use case specific topologies, like the one described in the motivational scenario.

a) Hybrid deployment: Since some operators require low latency, they need to be deployed close to the data source, e.g., on cloudlets [7]. Although cloudlets allow for a better response time because of the reduced communication channels, cloudlet-based computational resources are more expensive than typical cloud-based resources, due to the economy of scale [6]. Therefore, it is required that an IoT Runtime supports hybrid deployment and migration strategies, to migrate operators among different deployment locations [8].

b) Ease of designing new topologies: In order to establish IoT ecosystems, i.e., holistic toolchains that enable users to integrate heterogeneous IoT devices and process their data, it is essential to minimize the entry barriers for new users. Such entry barriers can be found at designing, deploying, and executing topologies. At design-time, it is required to provide an easy to use toolkit, which enables domain experts to design use case specific topologies. The same also applies at deploy-time and run-time. Here it is required that the runtime environment covers all aspects from obtaining external dependencies, such as operators, to ensuring the required QoS by obtaining enough computational resources for the execution. Furthermore, it is also required to design mechanisms, which enable the update of a topology at run-time to eliminate downtimes.

c) Reuse of operators: To minimize the required effort for implementing use case specific topologies, it is desirable to reuse already existing components similarly to the library usage for generic software development [9]. The data stream processing domain is predestined to decompose topologies into single operators and use them as building blocks. These building blocks serve as the foundation for a data stream processing ecosystem for the IoT, where all participants of the ecosystem can share their operators.

While the shared usage implies several organizational challenges, like monetization aspects, there are also several technical challenges. The most important technical challenge is a unified and self-contained exchange format that allows the execution of these operators without any further configuration. In this regard, the current trend towards lightweight containers suggests a promising solution [10]. Besides the packaging and distribution aspects of operators, each operator also needs to implement a minimal set of common functionality, like subscription mechanisms for data sources or a unified error-handling model.

IV. Related Work

In recent years, the number of IoT platforms not only in research projects, but also in commercial IoT integration solutions increased tremendously [3], [11]. Most of the academic projects target the integration of heterogeneous IoT devices and expose the data for further processing [12], [13], [14]. Nevertheless, there are also several industrial driven approaches, like Apache Quarks[3] or ThingWorx[4], which aim at integrating already existing SPEs as well as IoT devices to make them accessible for data processing operators.

Besides the integration of IoT devices and already existing SPEs, there are also approaches towards building marketplaces to share data and services to establish IoT ecosystems. Munjin and Morin [15] propose a marketplace for software components, which can be used to build applications for the IoT. In contrast to our approach, they only focus on the integration of IoT devices and omit the data processing aspect.

Akpinar et al. [16] propose the ThingStore, which provides a similar IoT platform to VISP. ThingStore aims at providing a marketplace for IoT data and IoT processing applications, which can be either used by developers to design complex IoT applications or end users, which buy processed data. These applications represent queries to the data stream for aggregating or filtering the data, similar to complex event processing systems or SPEs [17].

In addition to application and operator oriented marketplaces, Tien-Dung et al. [18] propose MARSA, a marketplace

[3]http://quarks.incubator.apache.org

[4]http://www.thingworx.com

TABLE I: Comparison of related IoT Platforms

	VISP	Groovestreams	IFTTT	Apache Quarks	ThingStore	ThingWorx	Amazon Web Services IoT	Google Cloud Dataflow	Microsoft Stream Analytics
Runtime	✓	✓		✓	✓	✓	✓	✓	✓
Marketplace	✓		✓		✓	✓			
Topology Builder	✓	✓	✓						
On-Premise Deployment	✓			✓					

for IoT data. MARSA allows data providers to offer their data to users, who either enrich the data by applying algorithms to resell the data or to integrate them into their own applications. Although MARSA focusses on a different aspect of the IoT, the monetization aspects deal with the same challenges. Furthermore, MARSA is so far the only IoT related marketplace that supports different business models.

Last, there are also commercial IoT platforms that aim at establishing an IoT ecosystem. Table I shows the most relevant ones compared to VISP and provides a comparison of them.

Groovestreams[5] offers a platform, which allows users to upload the data to their platform and analyze the data with pre-built operators. This platform also provides preliminary functionality for chaining different operators and realizing topologies that run elastically in the cloud.

The service IFTTT [19] focuses on the creation of topologies consisting of external services. Therefore, this service provides an extensive marketplace of different external services as well as a topology builder, which automatically includes the required data transformations between the external services. Since IFTTT only acts as a broker for external services, it does not provide a runtime.

Apache Quarks focuses on integrating IoT devices and existing SPEs into topologies. Although Quarks provides a runtime for its applications, it represents only an enabler for IoT platforms. Similarly, ThingWorx considers themselves as an enabler for IoT platforms, by integrating existing systems and IoT devices. In addition, ThingWorx also offers a marketplace for operators and provide so-called mashups to monitor data from IoT devices. Nevertheless, they do not support complex topologies.

Besides specialized IoT platforms, also the three major cloud providers Amazon Web Services IoT[6], Google Cloud Dataflow[7] and Microsoft Stream Analytics[8] provide runtimes to process streaming data originating from the IoT. While Amazon Web Services already provides a toolkit dedicated for the IoT, e.g., to integrate IoT devices, the other two service providers only offer generic data stream processing capabilities. For processing IoT related data, they need to integrate IoT platform enablers, like Apache Quarks.

When comparing these different IoT platforms, it can be seen that no platform so far, offers a holistic solution to realize stream processing topologies. While the major cloud resource providers offer reliable data stream processing capabilities, they only provide little support for users to integrate IoT-based data. Some providers, like ThingWorx and IFTTT, picked up the concept of operator reuse and offer a marketplace to ease the development for their platforms.

The same also applies for the user support for designing topologies. Currently, only IFTTT provides an infrastructure to easily design topologies, while all other providers only provide complex configuration interfaces to wire data sources and stream processing operators.

Furthermore, most platforms focus on a centralized cloud-based solution and do not consider the issues of privacy sensitive data. Finally, none of the existing IoT platforms or research approaches provide a holistic solution for creating an IoT ecosystem and realizing topologies.

V. Vienna Ecosystem for Elastic Stream Processing

In order to address the challenges as discussed in Section III, we propose the Vienna ecosystem for elastic stream processing. This ecosystem consists of two major components: the VISP Marketplace and the VISP Runtime, as depicted in Figure 2. The VISP Marketplace aims at creating an ecosystem for IoT-based topologies by minimizing the required effort for the user to design a topology. The VISP Runtime complements the VISP Marketplace, by providing a platform to execute the previously designed topologies. In this section, we are going to present the system design as well as the underlying design rationales.

A. VISP Marketplace

The VISP Marketplace is the first place to go for users, who want to implement use case specific topologies. It is composed of three major components: the Operator Registry, the Topology Builder, and the Billing Infrastructure.

The *Operator Registry* represents the graphical user interface, where a user can browse for data stream processing operators. Furthermore, they can also submit custom operators to the Marketplace to either integrate them into their topologies or to offer them to other users. When submitting a new operator to the Operator Registry, it is required to provide an *Operator Image* (see Section V-B), which can be instantiated by the Runtime. The user also needs to define all possible inputs as well as resulting data types for this operator and its billing model. This Operator Image with its respective metadata is then stored in the *Operator Image Storage*, where it can be accessed by the Runtime to realize the topologies.

[5]https://grovestreams.com
[6]https://aws.amazon.com/iot/
[7]https://cloud.google.com/dataflow/
[8]https://azure.microsoft.com/en-us/services/stream-analytics/

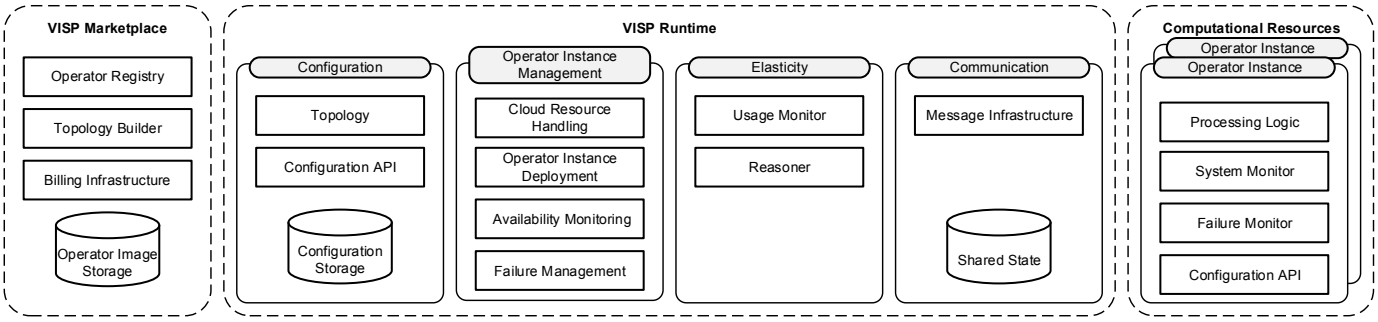

Fig. 2: VISP Ecosystem

Based on the Operator Registry, the Marketplace also offers a *Topology Builder*, which is the central element for designing use case specific topologies. The Topology Builder provides a graphical user interface that allows users to create topologies by dragging operators from the Operator Registry to a canvas, where they can wire the operators to realize the desired topologies. The Operator Registry offers custom and external operators, as well as data sources and data sinks. The characteristics of these operators are described in detail in Section V-C1. For each operator, the Topology Builder further provides a context menu, where the user can specify additional attributes for an operator such as QoS metrics, scaling strategies, and deployment restrictions.

Besides the basic design functionality, the Topology Builder also provides a context-aware search, where the user can search the Operator Registry based on the output type of the currently selected operator on the canvas to find a suitable successor. As soon as the topology is designed, the user can then export the topology based on the VISP Topology Description Language (see Section V-C1) to be deployed on the VISP Runtime.

The goal of the Topology Builder is to allow domain experts to create topologies for different use case scenarios.

The last component of the Marketplace is the *Billing Infrastructure*. The Billing Infrastructure takes care of the monetization aspects of the operators. Currently, the Billing Infrastructure supports three different business models: For the first business model, the Billing Infrastructure charges the user once for the usage of an operator and the user can then use the operator for an unlimited timespan and unlimited amount of operations. This model can only be applied to pre-built operators. The second business model implements a time restricted model, where the user can lease an operator for a specific timespan. As soon as this time span is over, the user is required to lease the operator again or the data processing operation cannot be performed anymore. The last business model implements a pay-per-use model, where the user has to pay for each operation carried out by an operator.

While the first model allows for an on-premise deployment of the topology, the other two models require a regular connection to the Marketplace to either check the validity of the leased operators as well as to report the performed operations for billing purposes.

These three models represent the most common business models for an IoT-based data stream processing topology, but we expect more elaborate business models in the future [20].

B. Operator Images

Operator Images represent building blocks, i.e., concrete implementations of operators, for stream processing topologies. These Operator Images are instantiated by the Runtime to create *Operator Instances* that process data.

Since Operator Images can be contributed by individual developers, we identified several requirements, which need to be met to use operators within VISP.

First of all, we do not restrict the Operator Image implementation to any programming language, as long as the Operator Image can be packaged within a container, i.e., a Docker Container[9]. This common format is required to ease the deployment for the Runtime, since it already provides a pre-configured execution environment.

Additionally, the Operator Image needs to implement several aspects. The most important aspect is the *Processing Logic*. The Processing Logic represents the functionality of the stream processing operator. This functionality ranges from simple filters over SQL-like aggregations to complex business logic [2]. Some operators can also include complex software systems, such as business process management systems or machine learning based decision systems, to conduct the desired operations. Besides the actual processing, the Operator Image also needs to implement a functionality to subscribe to data sources, i.e., preceding operators, as well as to publish the processed functionality.

To ease the implementation for operator developers, VISP already provides several Application Programming Interfaces (APIs) to access the *Shared State* for synchronization among Operator Instances as well as the *Messaging Infrastructure*, which are both part of the VISP Runtime. Nevertheless, it is also possible to access the Message Infrastructure directly, either by the AMQP [21] or the MQTT [22] protocol. For the Shared State, it is required to either use the APIs or send queries directly for the Redis Storage[10].

[9]https://www.docker.com
[10]http://redis.io

To ensure an SLA compliant execution of the data processing, it is required to obtain metrics from each running Operator Instance, like CPU or RAM usage as well as network I/O. These metrics are gathered within the *System Monitor* and the System Monitor reports the metrics to the Usage Monitor in a configurable interval by means of the Message Infrastructure. VISP already provides a System Monitor that can be integrated into the Operator Images without any further implementation required. Nevertheless, it is also possible to implement custom monitoring components.

Besides the QoS monitoring, it is also required to establish a failure propagation mechanism. Since Operator Instances are running autonomously on computational resources, it is necessary to aggregate failure reports at a central location. Therefore, each Operator needs to implement a *Failure Monitor* component that collects runtime exceptions and forwards them to the Failure Management of the VISP Runtime by means of the Message Infrastructure.

In order to configure an Operator Instance at startup, they have to implement a *Configuration API*. This Configuration API allows the *Operator Instance Deployment* to configure the location of the Shared State as well as the information about the data origin and the destination of the processed information in the Message Infrastructure. Additionally, the Configuration API may also be used to configure the functionality of the Operator Instances, e.g., by defining thresholds or filter criteria.

C. VISP Runtime

The VISP Runtime enables the user to run topologies. This Runtime has been designed, to be either deployed on a public cloud or on computational resources within the user's premises. The latter scenario allows to maintain the sovereignty over the data. Therefore, the Runtime is provided as a pre-built package, i.e., a Virtual Machine (VM) image, which needs to be deployed to computational resources. After providing the credentials for the Marketplace and the location for computational resources, the Runtime is ready to go. Nevertheless, it is also possible to deploy single components of the Runtime on different hosts if required.

Furthermore, it is also possible to deploy several instances of the Runtime at different geographic locations, e.g., on cloudlets, and use them to implement a hybrid topology. This is often necessary in order to reduce the latency between the data sources and the data processing operators. The deployment as well as the communication among the individual Runtimes does not require any interaction or configuration by the user, since the hybrid deployment is already considered in the fundamental design of the VISP Runtime.

Besides the overall replication mechanism of the Runtime, it is also possible to replicate individual components on clusters, e.g., the Message Infrastructure or the Shared State, within one instance of the Runtime to cope with the system load.

The Runtime is composed of four blocks: Configuration, Operator Instance Management, Elasticity, and Communication, whose components will be discussed in the remainder of this section.

1) Configuration: The configuration block is in charge of configuring the Runtime by the means of the *Configuration API*. Each update via the Configuration API is stored within the *Configuration Storage* component and accessible for all components of the Runtime. These configurations range from the location and name of other Runtime instances, over credentials for the Marketplace, to the credentials for computational resources.

Besides the Configuration component, this block also features the *Topology* component. This component is responsible for importing the topology description from the Topology Builder, propagating the configuration to other Runtime instances, configuring the Message Infrastructure and obtaining the Operator Images from the Marketplace, as described in detail in Section V-D.

In order to ease the communication between the Topology Builder and the Runtime, we designed the VISP Topology Description Language. This description language builds on the concepts of the SPL description language [23] and was extended to integrate external operators, SLAs, and deployment restrictions. Listing 1 represents the description for the pressure processing aspect of the topology as described in the Motivational Scenario (see Section II).

The topology in our scenario consists of different entities: one data source operator, two data processing operators, one external service, and one data sink operator, which are described in detail in the following.

The source operator, identified by the attribute *type*, is designated to record the raw sensor data and transform it into machine-readable data. These two different data formats are defined by the *inputFormat* respectively *outputFormat* attributes. Furthermore, each source operator is assigned a *source* represented by an IPv6 address, where the source operator can access the sensor data. There are two possibilities on how to access the data, as defined by the attribute *mechanism*. The first approach is the *listen* approach, where the source operator subscribes to the sensor and processes all data based on a push mechanism, while for the second approach, the source operator *queries* the sensor for data on a regular basis. Besides listening or querying sensor data from IoT devices, these source operators can also act as adapters to obtain streaming data from already existing SPEs. The last attribute for the source operator is the *allowedLocations* attribute. This attribute can be used to restrict the deployment to one or arbitrary many Runtimes, nevertheless, the default value for the allowedLocation attribute is "all". In our example, the source operator is restricted to be only deployed within the Runtime with the name "cloudlet1". Other operators, like the filterOp or storageOp in our example, process the output of the source operator. In contrast to the source operator, the processing operator is provided with the information of the preceding operator in the constructor, e.g., the `$filterOp` is processing the data from the `$source`.

Listing 1: Pressure Processing Topology

```
Topology "pressureProcessingTopology" {
 $source = Source() {
  allowedLocations = "cloudlet1"
  inputFormat      = "rawPressureData"
  source           = "2001:db8:0:1::"
  mechanism        = listen
  type             = "PressureSensor"
  outputFormat     = "machineReadablePressureData"
 }
 $filterOp = Operator($source) {
  allowedLocations = "cloudlet1", "cloud"
  inputFormat      = "machineReadablePressureData"
  type             = "FilterDamagedData"
  outputFormat     = "machineReadablePressureData"
  scalingThreshold = maxDelay = "500"
 }
 $reasonService = ExternalService($filterOp, $source) {
  inputFormat      = "machineReadablePressureData"
  type             = "ReasonOnKPI"
  location         = "2001:db8:0:2::"
  outputFormat     = "maintenanceRecommendation"
 }
 $storageOp = Operator($source, $reasonService) {
  allowedLocations = "cloud"
  inputFormat      = "machineReadablePressureData",
                     "maintenanceRecommendation"
  type             = "storeData"
  scalingThreshold = maxQueueSize = "200"
 }
 $sink = Sink($reasonService) {
  allowedLocations = all
  destination      = "2001:db8:0:3::"
  inputFormat      = "maintenanceRecommendation"
  type             = "informUser"
  outputFormat     = "processedMaintenanceRecommendation"
 }
}
```

Besides the common attributes with the source operator, the processing operator features the attribute *scalingThreshold*. This attribute is used by the Reasoner to decide if an additional Operator Instance for the specific operator has to be deployed in order to ensure the desired QoS.

Currently, the Reasoner supports two different scaling thresholds. First, the *maxQueueSize* threshold indicates the maximum amount of buffered messages in the Message Infrastructure for a specific operator until the new Operator Instance is spawned. The second threshold is the *maxDelay* in milliseconds, which represents the maximum time that can be passed between the data processing of the preceding and the current operator. Another notable aspect is that the outputFormat is only an optional attribute for the processing operator since some operators only consume the data, like the storageOp.

Besides the operators that are managed within the Runtime, the topology execution also relies on external services. In contrast to operators, external services do not have any location restrictions or scaling thresholds, since they are not managed within the scope of the Runtime. To access these services, the topology description language contains the attribute *location* of the service for defining the endpoint of the service.

The last entity is the sink operator, which pushes the processed information to the designated receiver, e.g., a user, a monitoring dashboard or an already existing SPE. This entity provides the attribute *destination*, the final location for the data.

2) Operator Instance Management: The Operator Instance Management takes care of deploying Operator Instances on computational resources, monitoring their availability, and aggregating runtime exceptions. Since the Runtime deploys Operator Instances on computational resources, i.e., cloud resources or computational resources on a cloudlet, it is required to obtain sufficient cloud resources, which is carried out by the *Cloud Resource Handling* component. This component is able to obtain computational resources from public clouds, e.g. Amazon EC2[11], or private clouds, e.g., OpenStack[12], and provides a suitable runtime environment for the Operator Instances. As soon as the computational resources are obtained, the Operator Instance Deployment component is able to deploy respectively un-deploy Operator Instances on these computational resources, whereas the deployment decisions are generated in the Reasoner. For each operator, there is, at least, one, but up to arbitrary many Operator Instances running at the same time.

Besides the resource provisioning and deployment mechanism, the Operator Instance Management also takes care of monitoring the availability of all Operator Instances and aggregating failure reports. The *Availability Monitoring* constantly checks whether all Operator Instances are available and capable of processing data. Whenever a downtime or failure of an Operator Instance is recorded, this component triggers a new instantiation by means of the Operator Instance Deployment component.

The *Failure Management* component aggregates all failures, either reported by the Failure Monitors of the Operator Instances or the Instance Availability Monitoring and informs the user about these failures. Optionally, it is also possible to inform the developer of the operator, by generating an error report, including the input data, information about the current state in the Shared State and a complete failure log output.

3) Elasticity: The Elasticity block of the Runtime takes care of the QoS compliance for the topology execution. Therefore, the *Usage Monitor* monitors three performance indicators for the overall execution. The first performance indicator is based on the individual System Monitors of the Operator Instances, which continuously report their performance metrics. On the one hand, a high system usage is an indicator that the current processing capabilities are not sufficient and it is required to add additional Operator Instances to cope with the system load. On the other hand, a low system usage can indicate the possibility to remove one Operator Instance for a specific operator to reduce the overall cost of computational resources.

The second indicator is the system load on the Message Infrastructure. Since the Message Infrastructure not only connects the individual operators, but also acts as a short term buffer, the size of all currently buffered messages can suggest either upscaling or downscaling actions.

[11]https://aws.amazon.com/ec2/

[12]https://www.openstack.org

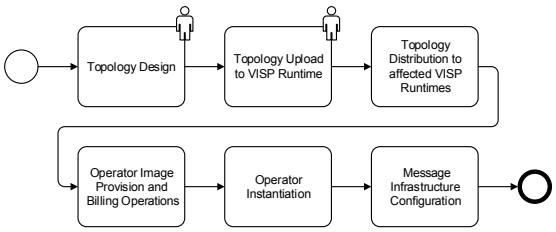

Fig. 3: Deployment Process for a Topology

The third indicator is observed by introspection of the individual messages on the message queue. Here the Usage Monitor selects individual messages in a configurable interval and identifies the processing duration of these messages based on the processing timestamp within the message. Whenever a message waits longer to be processed than defined by the operator, it can also be required to add additional Operator Instances for this specific operator.

Since the Usage Monitor only observes the performance indicators of the Runtime and the individual Operator Instances, the *Reasoner* component analyzes the performance indicators and decides then whether to scale up or down based on the SLAs provided for each operator.

4) Communication: Within VISP, there are two different layers of communication: first, the communication along the data stream processing topology and second, the synchronization among multiple Operator Instances for one operator.

The Message Infrastructure represents a message broker for realizing the communication along the data stream processing topology. While most established SPEs rely on a tight-coupled communication among the different operators, the Message Infrastructure for VISP allows for loose coupling, which can be reconfigured at run-time. These reconfiguration capabilities eliminate downtimes or topology redeployments, whenever an updated topology is deployed, which is not possible for established SPEs [24]. Another advantage of this loose coupling strategy is the possibility to deploy a single topology across different Runtime instances. For the communication within the Message Infrastructure, we require each message to contain a header to define the target operator of the message and a timestamp, stating when the last processing activity was performed, alongside any arbitrary payload in the message body. Finally, the Message Infrastructure also supports a short time buffer for messages that is required to compensate short communication outtakes or store the messages until enough processing capabilities, i.e., Operator Instances, are available.

For the synchronization among multiple Operator Instances for one operator, VISP relies on a shared key-value storage system, as suggested by Gedik et al. [25]. The Shared State allows that each operator to remain stateless. This eases the Operator Instance Management of the topology, by easily adding or removing Operator Instances in order to adapt to the system load.

D. Topology Creation

Each creation of a use case specific or an updated topology follows a specific workflow as depicted in Figure 3. At the beginning, the user, i.e., a domain expert, designs the topology by means of the Topology Builder located in the VISP Marketplace. This topology design only requires domain knowledge for the use case specific topology and can be carried out by dragging suitable operators to the canvas and wiring them. When the topology design is finished, the user exports the topology by storing the topology in a file, based on the VISP Topology Description Language.

The user then uploads the file to one instance of the Runtime. There the Topology component analyzes the topology, i.e., identifies changes compared to previous topologies and whether it is required to deploy any Operator Images on other Runtime instances. After the analysis phase is finished, the Topology component forwards the deployment decision to all affected Runtime instances. Each affected Runtime then obtains the required Operator Images from the Operator Registry and performs the billing operations for the operator. As soon as all Operator Images are obtained, one Operator Instance is deployed for each operator for the initial setup. Finally, the Topology component triggers all required modifications to the Message Infrastructure and the topology is ready to process the data streams or continue the data stream processing with the updated topology.

E. Implementation

The implementation of the VISP ecosystem is built on top of established technologies and libraries in the domain of cloud computing as well as deployment approaches. We have published the source code under the Apache 2.0 license on Github. The current state of the implementation features an ready to use Runtime[13] and exemplary Operator Image implementations[14], which can be used for feasibility evaluations of the VISP ecosystem. This implementation is completed by an exemplary data model[15] for the communication among the operators and a toolkit[16] to replay recorded data streams according to configurable data pattern to evaluate the resources elasticity aspects of the Runtime.

For the basic communication infrastructure, we rely on the Spring Cloud software stack[17]. This software stack provides several tools to implement distributed systems, which feature a reliable communication as well as failure detection and mitigation mechanism. It further supports the software developer to integrate a set of established technologies for different aspects as well as the basic functionality to implement custom capabilities, such as the Topology Management, the Cloud Resource Handling, the Failure Management or the Reasoner.

The communication functionality for the Runtime, is provided by RabbitMQ[18] to realize a reliable communication and Redis[19] for a efficient data storage.

[13]https://github.com/chochreiner/VISP-Runtime
[14]https://github.com/chochreiner/VISP-Processingnodes
[15]https://github.com/chochreiner/VISP-Datamodel
[16]https://github.com/chochreiner/VISP-Dataprovider
[17]https://github.com/spring-cloud
[18]https://www.rabbitmq.com
[19]http://redis.io

We selected these two software implementations because they are often chosen for distributed system due to their efficient data processing design and the possibility to create clusters to deal with high loads.

To ease the entry barrier for third-party developers who contribute custom operators, we decided to rely on the Docker tool stack for packaging Operator Images as well as for the backend for the Marketplace. The Marketplace builds on top of the Docker Registry[20] to provide the Operator Image Storage as well as an efficient and easy to use Operator Image distribution functionality. This technology decision allows us furthermore to deploy the Operator Instances either on a private OpenStack instance, as currently supported by the Runtime, or on commercial Docker Container Hosting services, like Tutum[21]. Besides the deployment related aspects, the Docker tool stack also provides a basic monitoring infrastructure, which is used to monitor the resource consumption of the Operator Instances by the System Monitor.

In contrast to established SPEs, our data processing model does not require a specific programming language or API to realize stream processing topologies. As long as the Operator Images provide the basic set of features as described in Section V-B, the operator developers are able to use any technology stack to implement the data processing functionality. This technology stack can differ based on the stream processing functionality of the operator. For simple functionalities on the one hand, such as filtering data, it may be sufficient to implement a custom solution. More complex ones, like the aggregation of data, on the other hand will be built based on established SPEs, such as Apache Spark or Apache Storm, to leverage the existing stream processing capabilities of provided by these SPEs.

VI. Evaluation

To discuss the feasibility and the usability of the VISP system design, we conduct a use case evaluation based on the motivational scenario described in Section II. For the evaluation, we consider multiple identical manufacturing machines located in three factories, whose data is ultimately aggregated within a cloud as depicted in Figure 4. Here, the configuration steps for a new topology are visualized by the dashed arrows, while the data flow based on the topology is shown by the solid arrows.

In the following we discuss the whole process of realizing a new use case specific topology based on a use case evaluation. The first step for creating a new topology is to analyze the use case, i.e., the predictive maintenance scenario, and identify its required functionality. Based on this requirement analysis, the user then checks the Operator Registry to identify already available operators. For our scenario, we assume that the Operator Registry already offers several operators: The source operators, the transformation operators, and the Reason on Shutdown operator are provided by the manufacturing machine

[20]https://docs.docker.com/registry/

[21]https://www.tutum.co

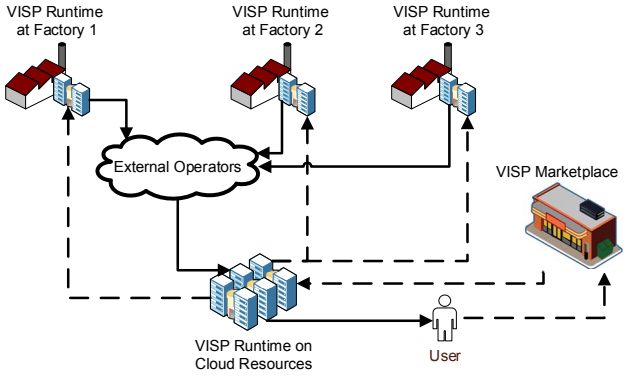

Fig. 4: Evaluation Scenario

producer free of charge because the producer wants to support the integration of their manufacturing machines in Industry 4.0 scenarios. The filter operators, the recording operators, and the Notify User operator are also available on the Marketplace. Since these operators are provided by third party developers, it is required to pay one time fees to integrate them in topologies. Furthermore, the Reason on KPI operator is also available as an external operator. The only missing operators are the domain specific ones: the Update Maintenance Plan operator and the Update Production Plan operator.

The two remaining operators need to be implemented by the user based on the APIs provided by VISP as described in Section V-B. As soon as theses two Operator Images are implemented, there are two possibilities to make them available for the Topology Builder. The first approach is to upload the operator, i.e., the Operator Image, to the Operator Registry and make it either publicly available for other users or flag them as private. Either way, they can be used to design the predictive maintenance topology and are also available for the deployment on the Runtime instances. The second approach is to create an operator stub, which only contains the metadata of the operator, in the Operator Registry, which allows designing the predictive maintenance topology in the Topology Builder, but the Operator Images need to be deployed manually to the Runtime instances. While the first approach is the recommended one, sometimes it is necessary to keep the Operator Images within the companies premises' to ensure the secrecy of data processing algorithms.

When all operators are available in the Operator Registry, the user can start to design the topology by means of the Topology Builder. Figure 5 depicts a screenshot of the Topology Builder showing an intermediate step of the topology design.

Simultaneously to the topology design, the user also deploys the Runtime instances. Therefore, the user instantiates the pre-packaged VM image, containing the Runtime, on cloud resources for the centralized data processing as well as three times on computational resources within the factories premises'. After instantiating these VM images, the user needs to provide the credentials for the Marketplace and the computational resources for the Operator Instances as well as the location of other Runtime instances.

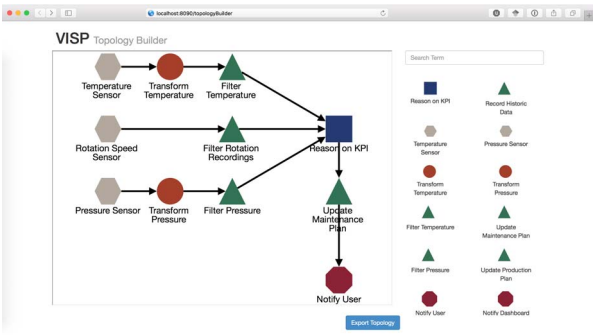

Fig. 5: Topology Builder

After completing the topology design, the user uploads the topology to the Runtime on the cloud resources. There the topology component analyzes the topology and forwards the relevant information to all other affected Runtime instances on the factories premises'. The affected Runtimes then obtain the required Operator Images from the Marketplace, perform billing operations, and wire the operators as described in Section V-D. This concludes the design- and deploy-time aspect of the system and the topology is ready to process data.

The run-time aspects of the predictive maintenance topology do not require any user interactions in order to increase the user acceptance. The Runtime takes care of deploying sufficient Operator Instances to handle the potentially volatile data input and, therefore, enables elastic topologies. Nevertheless, the Runtime also provides several tools, which support the operational aspects for users, like the Failure Management component that represents a single point to obtain all failures from the otherwise hardly accessible Operator Instances.

VII. Discussion

The use case evaluation shows that VISP tackles the challenges identified by us (see Section III) and Mineraud et al. [3].

First of all, the modular system design of the individual source operators and transformation operators allow to integrate heterogenous IoT devices as well as already existing SPEs into the VISP ecosystem. Whenever a user wants to integrate a new IoT device or low level middlewares, like Global Sensor Networks (GSNs), into the VISP ecosystem, it is required to implement a source operator. Depending on the provided data from the device or the GSN, the output either needs to be transformed from a binary format into a textual representation to be further processed by other operators or the source operator subscribes to the raw data stream and forwards them to the communication infrastructure. The implementation of this source operator should be typically done by the sensor manufacturer, since the manufacturer has the required knowledge to transform the often binary data, into a textual representation. The integration of existing SPEs or other middlewares can be realized by integrating Spring Cloud Binders[22] or other projects, e.g., storm-rabbitmq[23], which provides the integration of RabbitMQ, the communication backend for our VISP implementation, with Apache Storm. Although, each new IoT device requires a specific data handling approach, and therefore a custom source operator implementation, our system design only requires one adapter, which can be reused by numerous other topologies.

Second, VISP facilitates a privacy sensitive data processing approach. While other IoT platforms, like Groovestreams, ThingWorx or the cloud computing infrastructure by Amazon, Microsoft or Google, requires the data to be uploaded to a public cloud. VISP on the other hand can be deployed either on a public cloud, like the other IoT platforms, but on the other hand it can be also deployed on a private cloud or fixed computational resources due to its self contained architecture. This private deployment allows the data to be processed within the premises of the company and maintain the control over the data. Due to the flexible deployment possibilities of VISP, this privacy sensitive approach can be applied to the whole topology, or only to parts of the topology, which is currently not supported by established SPEs. The hybrid deployment capabilities not only improve the granularity of privacy control for the data processing, but also enables different levels of QoS, like the real-time processing capabilities on cloudlets close to the IoT devices or more cost efficient, but slower, processing in a centralized cloud.

Next to supporting the hybrid deployment approach, VISP also explicitly addresses the challenges of user support and operator reuse, which are essential for the creation of an ecosystem. In order to tackle the user support, VISP provides the Operator Registry as a graphical user interface and the Topology Builder to enable an easy topology design. This enables domain experts to design use case specific topologies and can therefore help to raise the acceptance rate for the VISP ecosystem. Furthermore, VISP also supports the exchange of operators among the participants of the VISP ecosystem since the Operator Images represent self-contained entities that can be executed on any VISP Runtime. The Operator Images also require only a minimal set of requirements, e.g., subscribing to the Message Infrastructure, when they are implemented. These minimal requirements lower the entry barrier for potential users of the VISP ecosystem, since it allows users to quickly package already existing stream processing implementations and participate in the VISP ecosystem.

Our use case evaluation, as well as further feasibility tests of our prototype, show, that the VISP ecosystem addresses all challenges that we have identified based on our motivational scenario. The modular system design of VISP allows to realize a hybrid deployment of the individual Operator Instances, based on their SLA, but also for the core components of the VISP platform to provide a scalable backend infrastructure. We can further see, that our topology design approach allows a better usability across different topologies in contrast for existing SPEs. Due to the fact that each existing SPE requires the usage of a specific API to implement the topology, each new topology needs to be implemented from scratch, including the operators, which are bound to the dedicated APIs.

[22] http://docs.spring.io/spring-cloud-stream/docs/current/reference/html
[23] https://github.com/ppat/storm-rabbitmq

Our approach tackles this problem by applying the VISP Topology Description Language to design an API agnostic topology, which is then executed by individual operators that can be reused across different topologies. Besides the decreased implementation effort for future topologies, our approach also supports the monetization possibilities for developers. This monetization aspects are vital for attracting operator developers to realize an vivid ecosystem. VISP currently supports several business models for operators, but it also allows the integration of further ones in the future, like bundling strategies for IoT devices or the monetization of the provided data by the sensors.

VIII. Conclusion

In this paper, we presented a holistic approach for realizing elastic data stream processing topologies for the IoT. We proposed an ecosystem that supports the user at design-time by reusing existing components for use case specific topologies and by providing a graphical user interface for creating new topologies. These topologies can then be deployed in a Runtime, which automatically propagates required configuration settings to other Runtime instances, autonomously provisions computational resources and wires the topology. The Runtime also takes care of using sufficient computational resources to comply with given SLAs at any time. The above-mentioned contributions lower the entry barriers for users to participate in the VISP ecosystem dramatically.

However, the design and implementation of VISP raised new research challenges, like the design of suitable business models for the IoT that we will tackle in our ongoing and future work. Since currently VISP only supports basic business models for operators, we are going to investigate towards business models that consider the data perspective and the network effects of the IoT [26]. Furthermore, we plan to extend VISP to address several challenges in terms of data ownership and data privacy [27]. To develop VISP further, we are going to investigate towards efficient cloud resource provisioning algorithms to minimize the operational cost for running data stream processing topologies.

Acknowledgments

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

References

[1] 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.

[2] 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.

[3] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of Internet-of-Things platforms," *arXiv preprint arXiv:1502.01181*, 2015.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[5] 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.

[6] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. of the 1st edition of the MCC workshop on Mobile Cloud Computing*. ACM, 2012, pp. 13–16.

[7] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: bringing the cloud to the mobile user," in *Proc. of the 3rd ACM workshop on Mobile Cloud Computing and Services*. ACM, 2012, pp. 29–36.

[8] 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.

[9] Ø. Hauge, C. Ayala, and R. Conradi, "Adoption of open source software in software-intensive organizations–a systematic literature review," *Information and Software Technology*, vol. 52, no. 11, pp. 1133–1154, 2010.

[10] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[11] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller, "A survey of commercial frameworks for the Internet of Things," in *20th Conf. on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2015, pp. 1–8.

[12] X. Su, R. M. Svendsen, H. N. Castejón, E. Berg, and J. Zoric, "Towards an integrated solution to internet of things-a technical and economical proposal," in *15th Int. Conf. on Intelligence in Next Generation Networks (ICIN)*. IEEE, 2011, pp. 46–51.

[13] A. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi, "Architecture and protocols for the internet of things: A case study," in *8th Int. Conf. on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. IEEE, 2010, pp. 678–683.

[14] S. De, P. Barnaghi, M. Bauer, and S. Meissner, "Service modelling for the Internet of Things," in *2011 Federated Conf. on Computer Science and Information Systems (FedCSIS)*. IEEE, 2011, pp. 949–955.

[15] D. Munjin and J. Morin, "Toward internet of things application markets," in *2012 Int. Conf. on Green Computing and Communications (GreenCom)*. IEEE, 2012, pp. 156–162.

[16] K. Akpınar, K. A. Hua, and K. Li, "ThingStore: A Platform for Internet-of-Things Application Development and Deployment," in *Proc. of the 9th ACM Int. Conf. on Distributed Event-Based Systems*. ACM, 2015, pp. 162–173.

[17] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[18] C. Tien-Dung, T.-V. Pham, V. Quang-Hieu, H.-L. Truong, D.-H. Le, and S. Dustdar, "MARSA: A Marketplace for Realtime Human-Sensing Data," *ACM Trans. on Internet Technology (TOIT) (accepted for publication)*, vol. NN, no. N, p. NN, 2016.

[19] D. Guinard and V. Trifa, "Towards the web of things: Web mashups for embedded devices," in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in Proc. of WWW (Int. World Wide Web Conferences), Madrid, Spain*, 2009, p. 15.

[20] S. Turber, J. vom Brocke, O. Gassmann, and E. Fleisch, "Designing Business Models in the Era of Internet of Things," in *Advancing the Impact of Design Science: Moving from Theory to Practice*. Springer, 2014, pp. 17–31.

[21] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, no. 6, pp. 87–89, 2006.

[22] ISO/IEC, *ISO/IEC 20922. Software engineering – Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1*. ISO/IEC, under development.

[23] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Nasgaard, R. Soule, and K. Wu, "SPL Stream Processing Language Specification," *NewYork: IBM Research Division TJ. Watson Research Center, IBM Research Report: RC24897 (W0911 044)*, 2009.

[24] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things," in *9th Int. Conf. on Cloud Computing (CLOUD)*. IEEE, 2016, pp. NN–NN.

[25] 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.

[26] M. E. Porter and J. E. Heppelmann, "How smart, connected products are transforming competition," *Harvard Business Review*, vol. 92, no. 11, pp. 64–88, 2014.

[27] Z. Yan, P. Zhang, and A. V. Vasilakos, "A survey on trust management for internet of things," *Journal of network and computer applications*, vol. 42, pp. 120–134, 2014.