A Service Based Architecture for Situation-Aware Adaptive Event Stream Processing

Marc Schaaf

University of Applied Sciences Northwestern Switzerland,
Riggenbachstr. 16, 4600 Olten, Switzerland
Email: marc.schaaf@fhnw.ch

***Abstract*—This paper presents the central aspects of a service based architecture for a distributed event stream processing system with an emphasis on its components, as well as related scalability and flexibility considerations. The processing system architecture is designed based on a well-defined situation-aware adaptive event stream processing model and a matching scenario definition language, which allow the definition of such processing scenarios in a processing system independent way.**

Keywords–Event Stream Processing; Microservices; Service-oriented Architecture; Publish-Subscribe

I. INTRODUCTION

Event Stream Processing (ESP) applications play an important role in modern information systems due to their capability to rapidly analyze huge amounts of information and to quickly react based on the results. They follow the approach to produce notifications based on state changes (e.g., stock value changes) represented by events, which actively trigger further processing tasks. They contrast to the typical store and process approaches where data is gathered and processed later in a batch processing fashion, which typically involves a higher latency. Event Stream Processing applications can achieve scalability even for huge amounts of streaming event data by partitioning incoming data streams and assigning them to multiple machines for parallel processing. Due to those properties, Event Stream Processing based analytical systems are likely to have a further increasing relevance in future IT systems. Also, it is likely that future ESP applications will have to handle even larger amounts of data while taking on increasingly complex processing tasks to allow for near real-time analytics to take place.

An example for such a scenario is the detection and tracing of solar energy production drops caused by clouds shading solar panels as they pass by [1]. The scenario requires a processing system to handle large amounts of streaming data to (1) detect a possible cloud (a possible situation), to (2) verify the possible situation and (3) to track the changes of the situation as the cloud moves or changes its size or shape. For the initial detection of a potential situation, a processing system needs to analyze the energy production of all monitored solar panel installations. However, for the second part, the verification of a potential cloud, only a situation specific subset of the monitoring data is needed. In the same way, the later tracking of the situation only requires a situation specific subset, which may change over time.

In order to handle such large numbers of events, a processing system needs to be capable of distributing the processing across several machines. A common mechanism for the distribution is to partition the overall data stream [2]. When a processing system partitions the incoming data streams in order to achieve scalability, such a partitioning will be suitable for the first part of the processing, the detection of a potential situation as the partitioning is *situation independent*. For the later processing part where a *situation specific* subset of the incoming data streams is required, a general stream partitioning scheme based on for example the processing system load, is not suitable as it does not incorporate the needs of currently analyzed situations. Here, a dynamic adaptation mechanism is needed that takes the investigated situations state into account.

This paper presents the central aspects of a microservice based architecture for a distributed event stream processing system that implements the afore mentioned processing model. The discussions put a specific emphasis on the architectures components, as well as related scalability and flexibility due to the realization as microservices based on the OSGi framework.

The remainder of this paper is structured as follows: The next section discusses the related work, followed by a presentation of the processing model in Section II to lay the foundation for Section III which discusses the goals of the here presented processing system architecture. Section IV then presents the architecture and its components. Before concluding the paper in Section VI, Section V discusses several development related aspects towards the presented architecture.

II. RELATED WORK

Various systems for distributing a processing system in order to provide the needed scalability exist like Aurora* and Borealis [3][4]. Aurora* for example starts with a very crude data stream partitioning in the beginning and tries to optimize its processing system over time based on the gathered resource usage statistics [5]. Furthermore, various approaches have been proposed which employ adaptive optimizations to handle load fluctuations by utilizing the dynamic resource availability of cloud computing offerings like [6][7][8] in order to scale on demand. Other approaches introduce new operators which allow for an adaptive partitioning or query plan execution. For example the Flux operator [9] allows for a dynamic partitioning of state-full operators during run-time in order to flexibly scale a stream processing system to handle varying processing loads. An even more flexible version is the Eddy operator which was proposed by Avnur et al. [10]. The Eddy operator allows for a continuous reordering of the operators in a query plan during run-time. The approach considers the query plan as a task where tuples need to be routed through the operators. Within this model, the Eddy operator allows a per-tuple routing decision thus allowing for a fine-grained control of the actual query graph during run-time. An application of the Eddy

 ISBN: 978-1-61208-606-4

operator to continuous queries exists with the Continuous Adaptive Continuous Queries over Streams (CACQ) [11].

In general, the here discussed systems are capable of setting up distributed stream processing based on given queries and to optimize the system to provide the required processing capacity and response times. However, the systems have no mechanisms to adapt deployed stream queries based on detected situations and situation changes as they have no knowledge of the overall analytical task that deployed a given stream query.

III. Processing Model and Language

We approach the outlined problem by defining a *situation-aware adaptive stream processing model* together with a matching *scenario definition language* to allow the definition of such processing scenarios for a scenario independent processing system [12][13][14]. The requirements for the definition of the model and language are the result of an analysis of several scenarios from two application domains, telecommunications network and Smart Grid monitoring.

The designed model defines situation aware adaptive processing in three main phases (Figure 1):

- Phase 1: In the *Possible Situation Indication* phase, possible situations are detected in a large set of streaming data, were the focus lies on the rapid processing of large amounts of data, explicitly accepting the generation of false positives and duplicate notifications over precise calculations.
- Phase 2: The *Focused Situation Processing Initialization* phase determines whether an indicated possible situation needs to be investigated or if it can be ignored, for example because the situation was already under investigation. If a potential situation needs to be investigated, a new situation specific focused processing is started.
- Phase 3: In the *Focused Situation Processing* phase, possible situations are first verified and then an in-depth investigation of the situation including the adaptation of the processing setup based on interim results is possible.

Based on our processing model we defined the Scenario Processing Template Language (SPTL), which allows the specification of processing templates based on the concepts of the processing model in an implementation independent way.

The SPTL allows the specification of the needed processing steps for each phase of the processing model. To allow the specification, it embeds several other languages:

- SPARQL [15] is used as the query language to retrieve background knowledge on the monitored system.
- MVEL [16] is used as an expression and scripting language.
- DROOLS [17] is used to specify the actual stream processing rules.

Furthermore, the SPTL allows the specification of variables which can be embedded and into the sub-languages in order to share processing results. Moreover, simple procedural statements can be made in SPTL.

In order to evaluate the processing model and language, a processing system prototype was developed. The following sections discuss the architecture of this processing system.

IV. Architecture Goals and Resulting Architecture Decisions

The two main aspects considered for the design of the here presented architecture are the scalability and the flexibility of the architecture. With regard to the scalability, the following aspects were considered:

- Handling of a large set of input streams that need to be monitored for potential situations.
- Detection and processing of many parallel situations.

The other main aspect considered is the flexibility with regard to the replacement and adaptation of components during development and while testing:

- *Adaptability of components:* The replacement of components in order to test and evaluate new concepts (interfaces came from shared library ensuring that interface changes can be made at one central place)
- *Local and distributed operation*: The integration with local test environment on developer system, as well as integration with a distributed test environment and the later deployment in a productive environment

While the mentioned aspects regarding the scalability of the processing system were also considered for the design of the processing model, the second aspect was only considered for the design of the processing system architecture. Thus, this paper focuses the discussions on the second aspect, the flexibility of the architecture.

V. Processing System Architecture

The overall processing system was subdivided into several components (Figure 2), each with distinct functionality. For the communication between the components interfaces were defined, which depending on the communication need are implemented as synchronous service based interactions or asynchronous message based interactions.

The following discussion will first outline the functionality of the supporting components. Then the four main components implementing the processing model are presented.

A. Background Knowledge Base (KB)

The processing model defines, that background knowledge on the monitored system can be retrieved for the set-up of the possible indication stream processing, as well as the later focused situation processing. The KB component provides access to this background knowledge. As the processing model only allows read only queries against the KB, the component can easily be scaled out onto multiple machines.

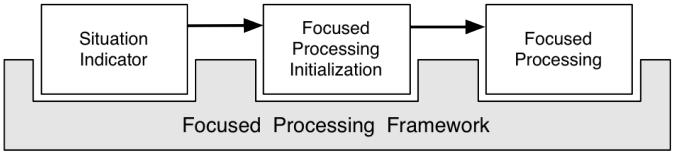

Figure 1. Three phases of the Processing Model

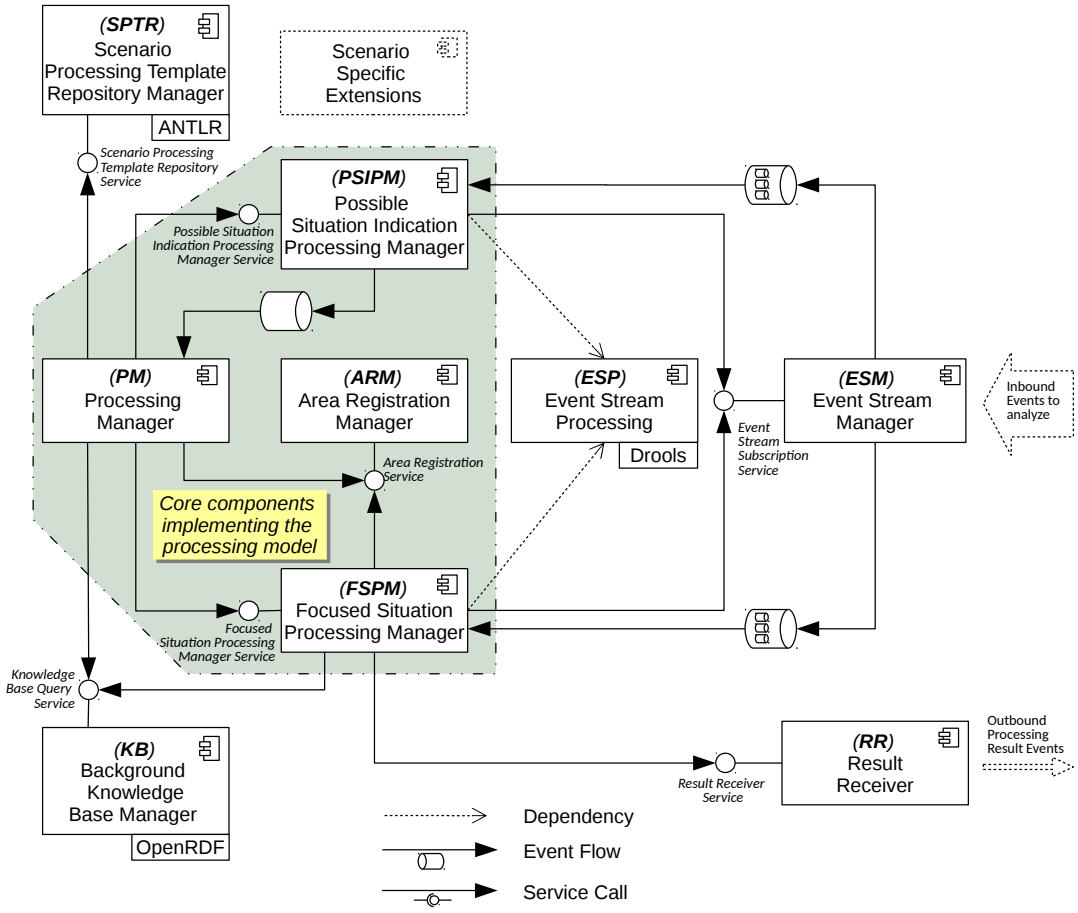

Figure 2. Architecture: Module View [14]

B. Scenario Processing Template Repository Manager (SPTR)

The SPTR is responsible for reading scenario processing templates specified in the SPTL, parsing them and performing an initial validation of the provided templates. The loaded templates can then be retrieved by the provided Scenario Processing Template Repository Service.

1) Event Stream Manager (ESM): The ESM allows other components to request data streams where the data streams can be live data streams or historic data delivered as a steam. When a component requests a certain data stream, the ESM responds with a handle to a messaging channel where the requesting component then subscribes to in order to receive the requested data stream.

2) Result Receiver (RR): The RR receives final or interim processing results in order to deliver them to a foreign system. As such different implementations of this component exist for a simple testing environment, where results are written to file or a live system where the results are for example forwarded to a decision support system.

Aside from the discussed components, there are further supporting components like the Event Stream Processing module or a module providing user defined scenario specific extensions. These components are used as shared libraries, providing an implementation of common functionality for the active components.

Based on these supporting components, the afore mentioned processing model is implemented. In order to allow for a distributed operation, the processing model itself is distributed across the following four components, each with a distinct functionality defined by the processing model:

C. Possible Situation Indication Processing Manager (PSIPM)

The PSIPM implements the functionality to instantiate a static stream processing network based on a provided stream processing topology specification. As such it interacts with the Event Stream Manager (ESM) where it subscribes to the required event streams. As the processing model does not allow any access to background knowledge during the indication stream processing, the PSIPM does not need access to the KB.

D. Focused Situation Processing Manager (FSPM)

The FSPM implements the actual situation specific analysis by providing the means to instantiate multiple situation focused processing tasks upon request. Each processing task investigates a single situation by implementing an iterative processing mechanism where each of these iterations follows a fixed process involving a preparation, a stream processing and a reasoning phase in order to conclude on interim processing results and to define the adaptation of the processing instance for the next iteration. The actual processing within each step is

 ISBN: 978-1-61208-606-4

specified by the user in the corresponding scenario processing template.

As multiple parallel processing tasks are executed in parallel, a synchronization between these instances is realized based on the Area Registration mechanism discussed in Section V-F.

The two afore discussed components, PSIPM and FSPM, both implement the actual stream processing tasks within this processing model. As they are decoupled from the rest of the processing system except from a very limited set of well-defined services, the stream processing can be implemented based on various technologies, based on the actual requirements from a given application domain. While the current implementation of these components utilizes JBoss Drools Fusion [17], an alternative implementation could for example utilize an Apache Spark [18] based cluster to realize the stream processing and thereby greatly enhance the scalability of the processing system.

E. Processing Manager (PM)

The PM acts as a stateful overseer component for the three phases of the processing model. It thus coordinates the initial set-up of the processing system for a given scenario. Further, for Phase 2 it implements the decision process regarding the instantiation of new situation specific processing tasks. The actual stream processing needed for Phase 1 and 3 are delegated to separate components.

For the initial set-up of processing tasks, the PM retrieves the list of available scenario processing templates from the SPTR component, executes the contained definitions for the Phase 1 processing (possible situation indication processing) to generate the stream processing topology definition for the given scenario. Once the topology was generated, the PM hands the topology definition to the PSIPM which in turn instantiates the topology and starts with the stream processing. When handing over the topology description, the PM also provides the PSIPM with a handle for a messaging channel where the PM listens for generated possible situation indication events published by the PSIPM upon the detection of a new possible situation.

When the PM receives a new possible situation indication event, it needs to decide if the event regards a new possible situation or an already investigated situation. The decision is implemented by the PM based on a set of user defined rules from the corresponding scenario processing template. Furthermore, the system needs to keep track of already active situation processing tasks and the situations under investigation. This tracking is implemented through an area registration mechanism which also acts as the main synchronization point for the processing system (discussed in the next section). If the PM decides that a new possible situation was detected, it requests the instantiation of the new Focused Situation Processing instance from the FSPM. If the event was deemed related to an existing situation, the PM delegates the event to the corresponding, already running, Focused Situation Processing Instance.

F. Area Registration Manager (ARM)

The processing model defines the Area Registration mechanism as the central synchronization mechanism between the processing Phase 2 and 3. An Area Registration is defined as a tuple consisting of two sets of nodes, the *Locked Area* LA and the *Focus Area* FA combined with the time frame tf of the registrations validity and a reference to the owner of the registration, a Focused Situation Processing Instance fpi:

$$ar := (LA, FL, tf, fpi)$$

The processing model then requires that at all nodes of any Locked Area LA may only be owned by a single Area Registration within the registration's validity time frame tf. Based on this mechanism, the processing model implements the synchronization between multiple parallel processing tasks. This synchronization is implemented by the ARM which keeps track of all active registration and is the single component with the authority to grant new area registrations. As such, the ARM is a crucial component for the whole processing system making its efficient implementation essential for the scalability of the overall processing system. For the current prototype, a simple Java based implementation of the ARM exists. However, for larger systems, a more optimized version which also considers for example the partitioning of the registrations in order to provide a distributed implementation should be considered. As all access to this component is also well-defined through a limited number of service based interactions, such a replacement would have no impact on the other components.

VI. Development Considerations

We based the processing system on the OSGi module framework where typically each component from the architecture became an OSGi bundle. Aside from the separation of the components as OSGi bundles, the OSGi framework also allowed for service based interactions between these bundles. This allowed the development of the processing system as an at first non distributed version running in a single Java Virtual Machine while for later distributed deployments, the OSGi Remote Services could be used. As distributed message based communication was not supported by OSGi at the time the system was implemented, we utilized the integration mechanism presented in [19].

Moreover, OSGi allowed for the replacement of components during runtime, which eased the development of the processing system and resulted in a more robust system with regard to the handling of service or component availabilities as components and services where often removed and replaced during runtime even while developing the system on a local machine.

Even though this approach allowed the development of a component based system, one of the larger challenges during the early versions of the processing system was the lack of traceability between components. This in particular with regard to the asynchronous message based interactions. We mitigated this problem by enforcing data types for the messaging integration which allowed for a limited traceability based on the data types.

In order to ensure that some basic functionality of the system was given even after larger changes, automatic integration tests where implemented as part of the build process for a small set of test scenarios. These tests turned out to be very fragile during the early development as still concepts of the model and language changed frequently. However, they also

 ISBN: 978-1-61208-606-4

ensured that even after a change in the processing model, the processing system was still capable to provide the processing capabilities needed for some basic scenarios, thus allowing for more confidence in the processing system even as larger changes occurred which regarded multiple components.

VII. CONCLUSIONS

The paper presents a highly modular service based architecture for realizing a processing system for situation-aware adaptive event stream processing. The architecture distributes the main parts of the situation aware processing model into four components, each with distinct functionalities. Further all supporting functionality is provided by separate components. All interactions between the components are defined either as service based or message based interactions which allows the components to be replaced independently.

By using the OSGi framework for the development of the processing system prototype we were able to develop this highly modularized, service based system locally in a single Java Runtime while retaining the possibility to distribute the prototype later based on the OSGi Remote Services.

Due to the design of the architecture together with the usage of the OSGi framework for the implementation, we could achieve the afore mentioned goals, for the development and testing of the system to allow for an easy adaptability of components in order to develop details of processing model and allow for integration into different environments, as well as allowing for a local and distributed mode to ease the development.

ACKNOWLEDGEMENTS

Parts of the here presented work was done as part of the Eurostars Project E!7377.

REFERENCES

[1] G. Wilke et al., “Intelligent dynamic load management based on solar panel monitoring,” in Proceedings of the 3rd Conference on Smart Grids and Green IT Systems, 2014, pp. 76–81.

[2] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, “A Catalog of Stream Processing Optimizations,” ACM Comput. Surv., vol. 46, no. 4, mar 2014, pp. 46–1. [Online]. Available: {http://doi.acm.org/10.1145/2528412}

[3] D. J. Abadi et al., “The Design of the Borealis Stream Processing Engine,” in In CIDR, 2005, pp. 277–289.

[4] Y. Xing, S. Zdonik, and J.-H. Hwang, “Dynamic Load Distribution in the Borealis Stream Processor,” in Proceedings of the 21st International Conference on Data Engineering, ser. ICDE ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 791–802.

[5] M. Cherniack et al., “Scalable distributed stream processing,” in In CIDR, vol. 3, 2003, pp. 257–268.

[6] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, “StreamCloud: A Large Scale Data Streaming System,” in Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on, june 2010, pp. 126–137.

[7] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, “Elastic scaling of data parallel operators in stream processing,” in Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, may 2009, pp. 1–12.

[8] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch, “Balancing load in stream processing with the cloud,” in Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops, ser. ICDEW ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 16–21. [Online]. Available: {http://dx.doi.org/10.1109/ICDEW.2011.5767653}

[9] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, “Flux: an adaptive partitioning operator for continuous query systems,” in Data Engineering, 2003. Proceedings. 19th International Conference on, March 2003, pp. 25–36.

[10] R. Avnur and J. M. Hellerstein, “Eddies: Continuously Adaptive Query Processing,” in Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’00. New York, NY, USA: ACM, 2000, pp. 261–272. [Online]. Available: {http://doi.acm.org/10.1145/342009.335420}

[11] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, “Continuously Adaptive Continuous Queries over Streams,” in Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’02. New York, NY, USA: ACM, 2002, pp. 49–60. [Online]. Available: {http://doi.acm.org/10.1145/564691.564698}

[12] M. Schaaf, “Event processing with dynamically changing focus: Doctoral consortium paper,” in RCIS, ser. IEEE 7th International Conference on Research Challenges in Information Science, RCIS 2013, Paris, France, May 29-31, 2013, R. Wieringa, S. Nurcan, C. Rolland, and J.-L. Cavarero, Eds. IEEE, 2013, pp. 1–6.

[13] M. Schaaf et al., “Towards a timely root cause analysis for complex situations in large scale telecommunications networks,” Procedia Computer Science, vol. 60, 2015, pp. 160–169, knowledge-Based and Intelligent Information & Engineering Systems 19th Annual Conference, KES-2015, Singapore, September 2015 Proceedings.

[14] M. Schaaf, “Situation aware adaptive event stream processing. a processing model and scenario definition language,” Ph.D. dissertation, Technical University Clausthal, 2017, verlag Dr. Hut, ISBN: 978-3-8439-3376-6.

[15] T. W. S. W. Group, “SPARQL 1.1 Overview,” Tech. Rep., march 2013, retrieved: 13.01.18. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

[16] “MVEL Language Guide for 2.0,” Online: http://mvel.documentnode.com/, retrieved: 13.01.18.

[17] “Drools Business Rules Management System,” Online: http://www.drools.org/, retrieved: 13.01.18.

[18] “Apache Spark Streaming,” Online: https://spark.apache.org/streaming/, retrieved: 13.01.18.

[19] M. Schaaf et al., “Integrating asynchronous communication into the osgi service platform,” in WEBIST’11, 2011, pp. 165–168.

 ISBN: 978-1-61208-606-4