Towards a Microservices-based Distribution 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 concepts for a microservices-based distribution of event stream processing pipelines as they are part of our situation-aware event stream processing system. For this, we outline changes to our specification language for a clear separation of the stream processing specification from the actual stream processing engine. Based on this separation, we then discuss our mapping approach for the assignment of the pipelines to stream processing nodes.**

***Keywords*–Event Stream Processing; Microservices; Service-oriented Architecture**

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. ESP 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, ESP based analytical systems are likely to have a further increasing relevance in future business 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.

While we presented the general high level architecture of our processing system in [3], this paper discusses two contributions, first the partial re-design of our specification language to become independent of the Drools Rule Engine and second the the assignment of the actual stream processing as a set of microservices to allow for the scalable distributed deployment of the stream processing pipelines.

The remainder of this paper is structured as follows: The next section discusses the related work followed by a presentation of the processing model. Section III and IV present the basics of our specification language for situation-aware event stream processing and outline the made changes. Section V presents the architecture and the mapping of stream processing pipelines to a microservice-based architecture.

II. Related Work

Various systems for distributing a processing system in order to provide the needed scalability exist like Aurora* and Borealis [4][5]. 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 gathered resource usage statistics [6]. 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 [7][8][9] in order to scale on demand.

In general, the 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.

On the other hand, there are systems aimed specifically at providing the surroundings for distributed processing but without providing processing languages like for example, Apache Storm [10] or Apache Spark Streaming [11]. Such systems could act as a potential basis for implementing our situation-aware adaptive processing model. However, they do not follow the microservice model we aim to explore with our architecture. For the realization of a reactive microservice like architecture as we aim for with our approach, lower level frameworks exist like for example Eclipse Vert.X [12] or Akka [13]. For our processing system architecture, we utilize Vert.X due to its integrated event bus functionality and define a mapping of our processing model to the available functionalities.

III. Processing Model and Language

We approach the initially 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 [3][14][15][16]. 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.

For these three phases, event stream processing takes place during the Phases 1 and 3.

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.

IV. Specification Language

A Scenario Processing Template contains all scenario-specific information to parameterize a processing system for

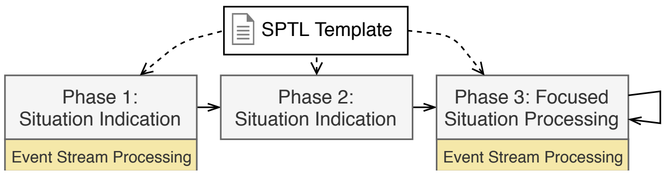
SPTL Template
Phase 1: Situation Indication
Event Stream Processing
Phase 2: Situation Indication
Phase 3: Focused Situation Processing
Event Stream Processing

Figure 1. Three main phases of the Processing Model

```
name "ScenarioName"

PossibleSituationIndication {
  // [...] Specifications for Processing Phase 1
}

FocusedSituationProcessingInitialization {
  // [...] Specifications for Processing Phase 2
}

FocusedSituationProcessing {
  // [...] Specifications for Processing Phase 3
}
```

Figure 2. Structure of a processing template in the SPTL

a scenario (e.g., How to detect a train delay and how to determine its impact). The template is divided into a preamble and three blocks which resemble the three major phases defined in the processing model as outlined in the listing in Figure 2.

Each block contains the specifications required for the setup and execution of the corresponding phase. Within the here discussed processing model, scenario specific event stream processing takes place during the Phases 1 and 3. Thus, the definitions of these two phases each contain the definition on how the scenario specific event stream processing has to be done. In the old version of the SPTL, the specification needed to be given as a Stream Processing Builder statement. This statement contained a mixture of several languages (Drools [17], MVEL [18], SPARQL [19]) in order to build/generate the actual event stream processing rules in the Drools Language.

A. Language Changes

One of the main limitations of this first version of the SPTL lies in its tight link to the Drools Rule engine, as well as in the complexity resulting from the combination of several languages. The tight link originates in the definition of the the actual stream processing statements which needed to be given based on the Drools Rule Language as shown in the listing in Figure 4.

In order to decouple the SPTL from the Drools Rule language and to further ease the stream processing specifications, the SPTL was extended with its own stream processing specification language. The new language is build around the concept of a stream processing pipeline as shown in the listing inf Figure 5.

The first statement $$focusArea.delay specifies the source of the events together with the type of the event ”DelayEvent”, the second statement ”filter(...)” defines a filtering condition followed by the last statement, which specifies a small function that shall be called for every event to allow a modification of the stream processing context ”context(...)”. The functions are separated from each other by ”=>” which indicates the forwarding of the event stream to the next processing step.

As the new stream processing specifications are independent of the actual stream processing engine used to execute them, alternative mappings of the language to a processing system implementation can now be defined aside from the Drools based mapping. As such we are investigating a mapping the processing pipelines to a microservice-based architecture based on the Vert.x tool-kit.

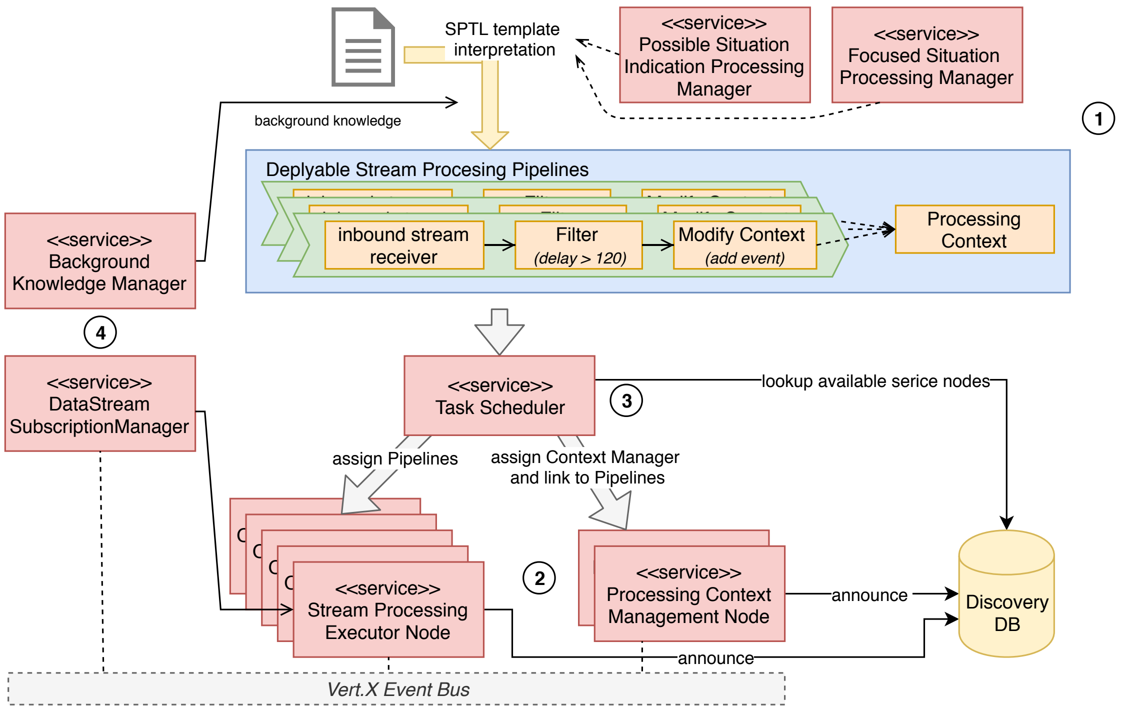
SPTL template interpretation
<<service>> Possible Situation Indication Processing Manager
<<service>> Focused Situation Processing Manager
1
background knowledge
Deplyable Stream Procesing Pipelines
inbound stream receiver
Filter (delay > 120)
Modify Context (add event)
Processing Context
<<service>> Background Knowledge Manager
4
<<service>> DataStream SubscriptionManager
<<service>> Task Scheduler
3
lookup available serice nodes
assign Pipelines
assign Context Manager and link to Pipelines
<<service>> Stream Processing Executor Node
2
<<service>> Processing Context Management Node
announce
Discovery DB
announce
Vert.X Event Bus

Figure 3. Overview of the stream processing pipeline assignment

```
IterationStreamProcessingBuilder {
  foreach $$focusArea as $$train {
    rule [DROOLS_TEMPLATE]
      when
        $a : DelayEvent( ) from entry-point "$${{train?
             delay}}"
      then
        CONTEXT.addToSet("$$trainEvents", $a  );
      end
      [/DROOLS_TEMPLATE] publishes no stream manipulates
           context;
  }
};
```

Figure 4. Stream Processing Builder definition in the old SPTL version

```
IterationStreamProcessing  {
  $$focusArea.delay : DelayEvent
    => filter( e.delay > 120 )
    => context( [M] $$trainEvents.add(e) [/M] );
}
```

Figure 5. Stream Processing definition in the extended version of the SPTL

V. Processing System Architecture

The overall processing system was subdivided into several components each with distinct functionality as discussed in [3]. For the communication between the components interfaces were defined, which, depending on the needed communication, are implemented as synchronous service-based interactions or asynchronous message based interactions.

However, in our initial architecture the stream processing itself was defined as one opaque component for the Phase 1 and as a second similar component for the Phase 3 processing with no further subdivision into smaller services. This initial design decision was caused by the use of one instance of the Drools Rule Engine for each of those components and the tight coupling of the SPTL to Drools. With the changes of the language as discussed in Section IV-A, the stream processing can now be subdivided into smaller sub-components based on the notion of defining a potentially distributed processing pipeline.

A. *Mapping of Stream Processing Pipelines to a Microservice-oriented Architecture*

For the processing system, such a processing pipeline consists of several separate stream processing statements where each statement takes a stream as input and potentially generates a stream as a result. Alternatively, a stream processing statement can also modify a so called processing context which is a shared data store in the context of the current situation's stream processing. Such pipeline definitions are the result of the interpretation of an SPTL template (Figure 3 Part 1).

In our microservices-oriented architecture, we map such pipelines to multiple small services where in the most fine grained form any stream processing operation could be provided by a separate service (Figure 3 Part 2). The services itself can then be distributed across several processing nodes in order to implement a distributed stream processing.

For the communication between the different microservices that form a processing pipeline the event bus mechanism provided by Vert.x will be used as it can act as a distributed peer-to-peer messaging system.

In order to deploy such a pipeline, we define a scheduler service. This scheduler has an overview over all available worker nodes which can execute stream processing tasks

(Figure 3 Part 3). This scheduler service is used by the two processing managers to assign their processing pipelines which they generated based on the given SPTL template. To allow the scheduler to find the available worker nodes, each node publishes itself as a service to a service registry, where the scheduler can thus find all available processing nodes.

Further services are provided that offer supporting facilities like an Event Stream Subscription Manager Service, allowing the pipeline nodes to request the needed event streams (Figure 3 Part 4). Moreover for the Phase 3 Stream processing, every situation specific processing requires a processing context that is shared between all stream processing pipelines associated with this situation. This processing context is again provided by a separate service assigned by the scheduler service (Figure 3 Part 2).

VI. Conclusion and Outlook

The paper outlines an extension of our service-based architecture towards the use of microservices for the distribution of the actual stream processing. The distribution is based on processing pipelines which we introduced by adapting the scenario template specification language. In our approach, the stream processing is realized by multiple microservices which together form a concrete stream processing pipeline potentially distributed across multiple machines. The actual distribution decision will be made by a scheduler service which oversees the available resources through their service registrations.

Currently, our model and architecture does not define any mechanisms for handling component failures during processing. We plan to add such a functionality in the form of an overseer service which monitors deployed pipelines, detects service failures and re-deploys the failed services. This however also requires an extension of the processing model itself so that a partial rollback of inconsistent processing state becomes possible, thus allowing the processing to resume in a defined state after a failure.

While the discussed language changes are already implemented, future work is the realization of the proposed microservice-based distribution as part of our prototype, thus, allowing for a detailed evaluation of the approach. In particular an evaluation of the performance of the Vert.x event bus in a distributed setup needs to be conducted as the later processing system will use this as its communication backbone and will thus rely on its performance.

Acknowledgments

Parts of the here presented work were done as part of the Eurostars Project E!7377 as well as Project 18014 funded by the Hasler Stiftung.

References

[1] G. Wilke, M. Schaaf, E. Bunn, T. Mikkola, R. Ryter, H. Wache, and S. G. Grivas, “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] M. Schaaf, “A service based architecture for situation-aware adaptive eventstream processing,” in The Tenth International Conference on Advanced Service Computing, Barcelona, Spain, February, 2018, pp. 40–44.

[4] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, “The Design of the Borealis Stream Processing Engine,” in In CIDR, 2005, pp. 277–289.

[5] 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.

[6] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik, “Scalable distributed stream processing,” in In CIDR, vol. 3, 2003, pp. 257–268.

[7] 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.

[8] 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.

[9] 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}

[10] “Apache Storm,” Online: https://storm.apache.org/, retrieved: March/04/19.

[11] “Apache Spark Streaming,” Online: https://spark.apache.org/streaming/, retrieved: March/13/19.

[12] “Vert.X Homepage,” Online: https://vertx.io/, retrieved: March/13/19.

[13] “Akka Homepage,” Online: https://akka.io/, retrieved: March/13/19.

[14] 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.

[15] M. Schaaf, G. Wilke, T. Mikkola, E. Bunn, I. Hela, H. Wache, and S. G. Grivas, “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.

[16] 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.

[17] “Drools Business Rules Management System,” Online: http://www.drools.org/, retrieved: March/13/19.

[18] “MVEL Language Guide for 2.0,” Online: http://mvel.documentnode.com/, retrieved: March/13/19.

[19] 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/