Quality of Service Based Event Stream Processing Systems in Smart Grids

Epal Njamen Orleant
Grenoble INP, LIG
Saint Martin d'Hères, France
Email: orleant.epal-njamen@imag.fr

Lourdes Martinez
Grenoble INP, LIG
Saint Martin d'Hères, France
Email: martinez@imag.fr

Christine Collet
Grenoble INP, LIG
Saint Martin d'Hères, France
Email: christine.collet@grenoble-inp.fr

Genoveva Vargas-Solar
CNRS, LIG-LAFMIA
Saint Martin d'Hères, France
Email: genoveva.vargas@imag.fr

Christophe Bobineau
Grenoble INP, LIG
Saint Martin d'Hères, France
Email: christophe.bobineau@grenoble-inp.fr

Abstract—**This paper presents an approach for composing event streams based on quality of service requirements (QoS) of smart grids. The approach consists of an event stream model, composition strategies guided by QoS such as memory consumption, event priority and notification latency. Model and strategies are implemented by a distributed event stream processing system consisting of execution units that can be deployed across a smart grid. The paper describes implementation issues and experimental results.**

***Keywords*–Complex event processing; Quality of service; Smart Grids.**

I. Introduction

Smart grids are complex networks vastly instrumented with intelligent electronic devices (sensors, smart meters, actuators, etc.), network communication and information technologies. Devices emanate huge amounts of data that can be exploited for a wide range of applications like network traffic analysis, automation of operational control, prevention or detection of dysfunctions, etc. Strategies to handle asynchronous data collection, data transfer, and real-time data notification and processing are critical for achieving smart grid monitoring.

Those data can be considered as events that refer to happenings of interest produced within the system environment. The capacity to monitor and supervise a smart grid relies on processing low level events in order to infer higher level events semantically richer and more useful for end user applications [1]. This process includes events filtering, aggregation, correlation, windowing, etc. Infrastructures able to achieve these computations on events are referred to as complex event processing systems like [2–6].

For example, let us consider that a smart meter produces an event of type CoverOpenAlert when its cover is opened, and a sensor produces an event of type BadVoltage when it detects an abnormal voltage on the electrical line. An application may be interested in the sequence of CoverOpenAlert and a BadVoltage occurring at the same place, within a two minutes time window. This pattern detects suspicious activities (MeterSuspected event type) on smart meters. The detection of such a high level event includes event filtering (type and attribute based filtering), windowing and temporal correlation.

In these situations, devices may notify events to a remote Information System (IS) able to perform complex events processing. The IS sends commands (with or without human intervention) to certain devices for reacting to the reported events. The dialog IS - devices may take considerable time, thus hindering real-time requirements. An intuitive manner for alleviating this problem is to inject certain intelligence into devices, such that they can react to situations without some external intervention. Thus, (total or partial) event processing should be distributed among the smart grid devices. An inherent consequence is the necessity to deploy event processing systems in distributed architectures. The latter must efficiently achieve event processing while adapting to their environment in terms of the multiplicity of data sources (sensors, smart meters, existing databases, etc.) and smart grid QoS requirements.

a) Multiplicity of data sources: Distributed systems like smart grids consist of different types of components that can act as event producers or consumers, with different interaction modes (synchronous or asynchronous, push or pull based style), as illustrated by sensors, smart meters, existing databases. The diversity of interaction modes, coupled with the difference in data formats make it difficult to integrate events from different producers for event processing purposes.

b) Quality of service (QoS): The need to detect and notify complex events from basic events is sometimes correlated with some quality of service requirements like memory consumption, network occupancy, event priority, notification latency, etc. The extension of event models towards more flexible and QoS oriented event models requires an analysis and the semantics that should be given to the events, and of their associated processing strategies. This requires dissociating the modeling of event and the application design and, the proposal of methods that allow to define event types independently of the management issues (detection, production, notification). Therefore, it is required to adapt the event models to smart grid characteristics. On the other hand, those QoS requirements generally constrain the way the event processing must be achieved. More precisely, event processing must be achieved on each device considering its memory availability.

Existing systems are limited in the sense that they do not fully satisfy QoS requirements for event processing. The problem we address in this paper can be summarized as follows: given smart grid needs in terms of event composition and QoS, how to provide the complex event processing system that best fulfills expected QoS requirements?

Our approach considers an event based abstraction of smart grids functions and services. This abstraction allows to reason on the smart grid in terms of event streams that are generated by smart grid components. In order to identify relevant or critical situations (complex events) among those event streams, we propose a distributed complex event processing architecture. The event processing logic is implemented as a network of operators executed by distributed event processing units. We also propose strategies applicable to event processing units in order to address the following QoS dimensions: event priority, memory occupation and notification latency.

The remainder of the paper is organized as follows: Section II presents the related work. Section III presents the overview of our approach for QoS based complex event processing in smart grids. Section IV describes the proposed model and system architecture for QoS based event processing. Section V discusses how to specify QoS requirements and introduces the QoS adoption strategies. These strategies are presented in Section VI and Section VII. Section VIII discusses the experimental results. Finally, Section IX concludes the paper.

II. Related Work

Many works have been achieved on event streams analysis and composition, and many event processing systems have been proposed so far [2–6], either for centralized or distributed architectures.

In centralized architectures, produced events are processed by a single node acting as an event processing server [3][5][6][7][8]. In this approach, event streams must be routed to the server node. This potentially increases the latency of the event processing, and overloads the network and server, which risks to become a point of failure. Therefore, this approach is not suited for distributed contexts.

In distributed architectures, the event processing logic is performed by a set of distributed communicating nodes, each one achieving a part of the work. This offers a better scalability and availability than centralized approaches. Some distributed event processing systems are [2][4][9][10]. In this category, we distinguish between clustered and in-network architectures. In clustered architectures, the event processing is realized in a clustered environment [4][11], whereas in-network architectures allows to distribute the event processing over a large number of nodes within a network topology [9][2]. This work aims to propose an in-network event stream processing systems for smart grid that deals with QoS.

Behnel et al. [12] and Appel et al. [13] identify some QoS dimensions (latency, priority, etc.) relevant for distributed event processing, but they do not propose mechanisms for their adoption. However, some other systems provide QoS support. They optimize the query processing according to a particular objective, and differs from each other by the adopted QoS dimensions. For example, [2] focuses on reducing the network traffic whereas [9] studies energy consumption. In wide networking environments, it is not reasonable to expect that all applications share the same objective. In our approach, we identify a set of QoS properties relevant for event processing in smart grids, and we study their adoption by the event processing system.

A survey on the QoS requirements of smart grid communication systems is presented in [14]. It focuses on the functionalities that have to be provided by smart grid communication infrastructures in order to address application requirements. Sun et al. [15] propose to add QoS by providing differentiated service for data traffic with different priority at the MAC (Media Access Control) layer. GridStat [16] is a publish-subscribe middleware framework designed to meet the QoS requirements for the electric power grid. It manages network resources to provide low-latency, reliable delivery of information produced anywhere on the network and sent to multiple other points. In our work, we assume the existence of QoS support at the networking layer (e.g, message priority) on which a complex event processing system dealing with event priority, memory occupation and notification latency can be proposed for smart grids.

III. Approach overview

Figure 1 summarizes our approach to integrate complex event processing technologies into smart grids. It consists in three layers of abstraction, namely smart grid, event streams, and event processing network layers.

- The smart grid layer consists in the real physical smart grid architecture, which includes telecommunication based devices such as smart meters, sensors, data concentrators, etc. Those devices are connected by communication networks technologies including power line, wireline or wireless communications [17]. The smart grid is described in terms of information being used and exchange between functions, services and components. This layer of abstraction is referred to as the *Information layer* in the smart grid reference architecture model [18]. In our approach, information is seen as events that happen within the smart grid.
- The event streams layer considers that data generated by smart grids components are event streams. In this layer, smart grid components act as sources, which can generate different types of events in a continuous manner. The event model considered in this work is presented in Section IV.
- The event processing network layer consists in a set of distributed event processing units that are connected by event channels. This network is created according to complex event subscriptions. It may be deployed across multiple distributed computers, software artifacts and physical networks. The complex event subscriptions are tagged with applications QoS requirements such as event priority and notification latency. Those QoS requirements have to be translated into constraints applicable to event processing units at execution time. In addition to constraints derived from applications requirements, inherent constraints to the smart grid infrastructure also must be taken into consideration, such as limitations on computational

resources (i.e., memory and CPU) and / or communication networks (i.e., network occupation).

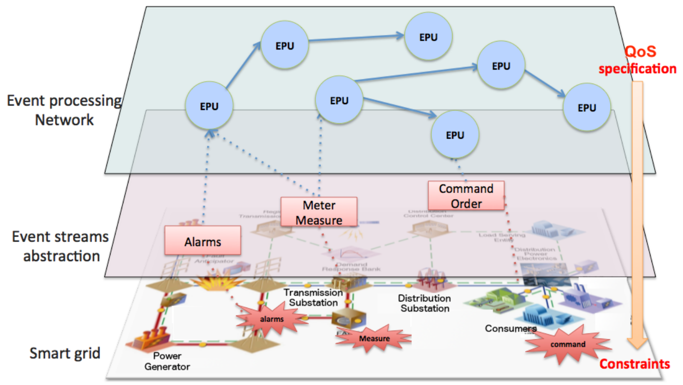

Figure 1. Approach overview

IV. Model and Architecture

This section presents the event model (event type and event stream), and the runtime architecture of our approach (event processing network).

A. Event model

1) Event type and event instance: An event type represents a class of significant facts (events) and the context under which they occur. The definition of an event type includes the attributes presented in Table I.

TABLE I. Event type attributes

Name	Type
typeName	String
producerID	String
detectionTime	Number
productionTime	Number
notificationTime	Number
receptionTime	Number
priority	Number
context	Set<Attribute >

The *typeName* attribute refers to the name of the event type. The *producerID* attribute refers to the id of the entity who produced the event instance. The *detectionTime* attribute refers to the time at which the event instance has been detected by a source. The *productionTime* attribute refers to the time at which the event has been produced (as a result of a processing on others events) by an event processing unit. The *notificationTime* attribute refers to the time at which the event is notified to interested consumers. The *receptionTime* refers to the time at which the event is received by an interested consumer. The *priority* attribute represents the priority value associate to the event instance. The context (*context* attribute) of an event type defines all the attributes that are particular to this event type. They represent the others data manipulated by the producer, which are relevant to this event type. For example, the context of a *MeterMeasure* event type generated by a smart meter includes the *voltage* and *current* attribute.

An event type can be simple or composite. Simple event types are event types for which instances are generated by producers (sensors, smart meters, etc.). They are not generated from the processing of other events. In the example considered in Section I, $BadVoltage$ and $CoverOpenAlert$ are simple event types. More generally in a smart grid, the event types include $Alarms$, $MeterMeasure$ and $SensorMeasure$ generated by electric devices and such as smart meters and sensors, and $Command$, $ControlOrder$, $ControlAction$ generated by utility applications.

Complex (or composite) event types are event types for which instances are generated as a result of event processing. Reference [19] includes a set of operators applicable to events. They capture particular situations (relevant or critical) that can be inferred from occurrences of others events. Those situations have to be notified to utility applications, such that the system can be automatically or manually controlled. In the same example, $MeterSuspected$ is a complex event type. Complex event types can also capture aggregated values, like the daily electricity consumption of a household. This can be product of the aggregation of the $MeterMeasure$ event instances included on a one-day window.

An event instance (or simply event) is an occurrence of an event type. The event instance defines the value associated to each attribute of the event type. For example, the event occurrence e with attributes presented in Table II denotes an event instance of type $MeterMeasure$, which has been produced by producer *meter1* at time 1, notified at time 2, received at time 3, which has a priority value 3, and for which the voltage and current values are 220 and 3, respectively.

TABLE II. Event instance

Name	Type
typeName	'MeterMeasure'
producerID	'meter1'
detectionTime	1
productionTime	1
notificationTime	2
receptionTime	3
priority	3
voltage	220
current	3

2) Event stream: An event stream is a continuous, append-only sequence of events. We note $Stream(s,T)$ the stream of events of type T generated by the source s. If S is a set of sources, then $\{\bigcup stream(s,T), s \in S\}$ defines a stream of events of type T, denoted $Stream(T)$.

B. Event processing network

As introduced in Section III, the event processing logic is implemented by the event processing units. The runtime deployment of event processing units with associated event channels is called the event processing network [20][21]. This is illustrated in Figure 2.

The general vision of our QoS based complex event processing system can be briefly described as follows: applications subscribe to composite events by issuing complex event patterns to the system, this must also include the specification of the associated QoS requirements. The system then deploys a set of distributed event processing units, which apply different strategies to meet QoS requirements during event processing. Complex events produced by the event processing units are notified to consumers. In a smart grid, such an infrastructure can act as a middleware on which utility applications rely for detecting interesting or critical situations (sensors errors, alarms, etc.) over the electrical grid, and at the same time, rely on some QoS guarantees (e.g., priority, notification latency, etc.).

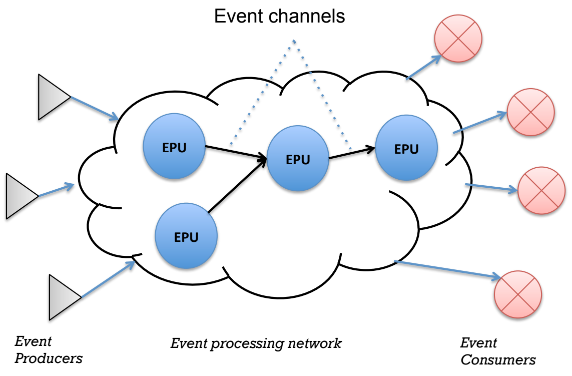

Figure 2. Event processing network

1) Event processing unit: An event processing unit can be defined by three types of components (see Figure 3):

- a set of input queues, on which parts of input event streams are maintained.
- an operator, which implements a three step event processing logic: *fetch-produce-notify*. In the first step (fetch), some events are selected from the input queues and marked as ready to be used to produce new composite events. In the second step (produce), the events selected at the first step are used to produce new composite events according to the operator semantic. The produced complex events are stored in the output queue. In the third step (notify), events in the output queue are notified either, to another event processing units or to the interested consumers.
- an output queue, which contains events to be notified.

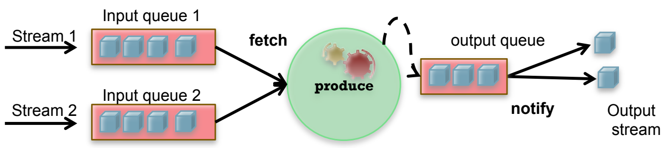

Figure 3. Event processing unit

2) Event channel: Event processing units communicate through event channels. Event channels are means of conveying events [22]. This can be done via standard tcp or udp connections, or higher level communication mechanisms like publish/subscribe [23] or group communication [24] provided by a middleware layer.

C. Architecture

The architecture of the proposed QoS based event processing system is depicted at Figure 4. It consists of four layers described as follows.

- the application layer consists of two types of components: event producers (sensors, smart meters, etc.), and event consumers that subscribe to complex event patterns having specific QoS requirements.
- the event processing network layer consists of a set of distributed event processing units that communicate among them via event channels.
- the middleware layer provides a high level communication mechanism to event processing units. This can be publish/subscribe [23][16] or group communication services [24]. It relies on the underlying network layer.
- the network layer ensures messages delivery from one destination to another.

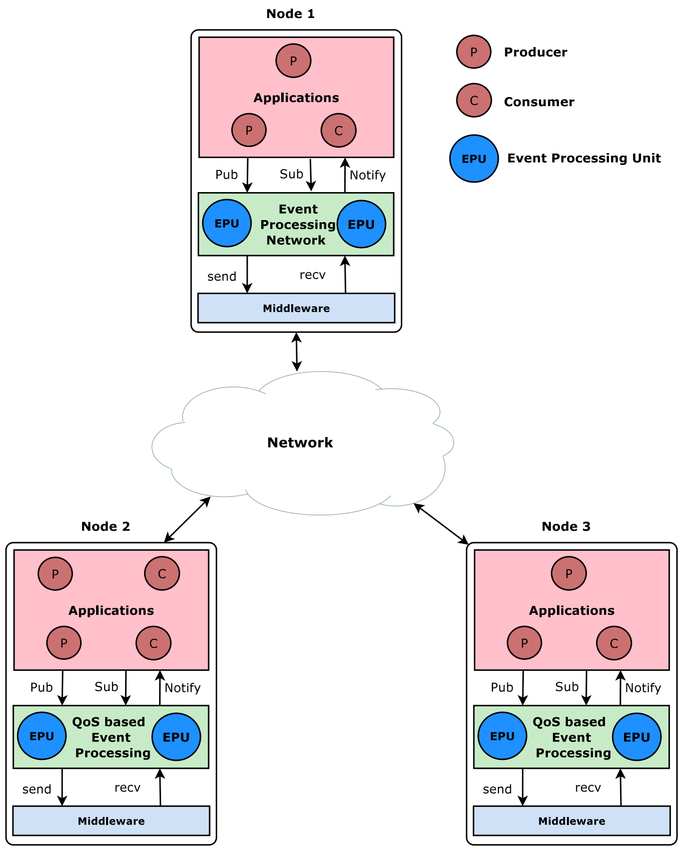

Figure 4. System architecture

V. QoS SUPPORT IN EVENT PROCESSING

The need to detect and notify complex events from basic events is sometimes correlated with some QoS requirements. The QoS dimensions we address in this paper are event priority, notification latency and memory occupation. Those QoS requirements are either imposed by smart grid applications (event priority, notification latency), or by the execution environment (memory occupation).

1) Event priority: Event priority defines a priority order between events. In some contexts, events may have different priorities that have to be captured at event processing runtime. For example, in a smart grid, a BadVoltage event can be higher priority than a CoverOpenAlert event. Events that have a higher priority have to be processed and notified earlier than less priority events.

2) Memory occupation: Smart grid devices may have different memory capacity. To adapt event processing to the memory capacity of devices, it must be a way to specify the maximum memory occupation incurred by an event processing unit at execution time. The memory occupation constraint gives an upper bound of the number of events that an event processing unit can maintain at execution time.

3) Notification latency: In the common practice for power device protection, the circuit breaker must be opened immediately if the voltage or current on a power device exceeds the normal values. The notification latency of an event is the time elapsed between its production and its notification to interested consumers (end users or event processing units). The notification latency constraint imposed on an event processing unit defines an upper bound on the notification latency of events produced by that event processing unit.

A. QoS expressions

Each QoS requirement is associated to a specific value domain. Below we specify the value domains corresponding to the introduced QoS requirements:

- $D_{latency}$ denotes the notification latency domain. Latency is a measure of time that adopts a numeric value expressed as either, a positive integer or a positive fraction. Therefore, a latency value belongs to the domain of real numbers $\mathbb{R}^+$. Thus, we can say that $D_{latency} \subseteq \mathbb{R}^+$. We also assume that the arithmetic and comparison operations that can be applied on real numbers also can be applied among values belonging to $D_{latency}$. For instance, intuitively a low latency is preferable than a high latency, thus it can be desirable to compare latency values using the comparison operators less than($<$), and less than or equal to ($\leq$).
- $D_{priority}$ denotes the event priority domain. An event instance is associated to a priority level that varies according to the event type. A priority level is represented with a positive integer value. Therefore, we consider that a priority value belongs to the domain of natural numbers $\mathbb{N}$; thus, $D_{priority} \subseteq \mathbb{N} \leq$ n. We assume that we can use comparison or arithmetic operators on latency values. The priority is a heavily restricted bounded QoS requirement, priority = 1 denotes the highest priority and priority = n denotes the lower priority. The equal to (=) operator is required to associate an event to a priority level.
- D_{memocc} denotes the memory occupation domain. We express the memory occupation in terms of number of events, for this reason, such a requirement adopts an integer positive value thus belonging to the domain of natural numbers $\mathbb{N}$. We state that $D_{memocc} \subseteq \mathbb{N} \leq$ m. Where the comparison operator less than or equal to ($\leq$) specifies an upper bound and m specifies the maximum memory capacity of the current device.

Let us assume that D is the set of the considered QoS domains, thus $D = D_{latency} \bigcup D_{priority} \bigcup D_{memocc}$. Given a domain D_Q, we assume a function *name(D_Q)* that returns the domain name, a function *operator(D_Q)* that returns the set of related operators, and a function *value(D_Q)* that returns the set of possible values.

For instance, let us consider the domain $D_{latency}$, thus:

- *name(D_Q)* = notification latency
- *operator(D_Q)* = greater than ($>$), greater than or equal to ($\geq$), less than ($<$), less than or equal to ($\leq$), equal to (=), not equal to ($\neq$)
- *value(D_Q)* = $\mathbb{R}^+$, this is the set of all positive real numbers

1) Atomic QoS expression: An atomic QoS expression α specifies a QoS requirement. It is of the form *(n, Θ, v)*, where

- n denotes a domain D_Q , where $D_Q \in D$
- $\Theta \in$ *operator(D_Q)* and,
- $v \in$ *value(D_Q)*

For instance, the atomic QoS expression (notification latency, $\leq$, 2000 ms) specifies that the latency for notifying an event must be equal than or less to 2000 milliseconds.

2) Complex QoS expression: A complex QoS expression ε specifies multiple QoS requirements. Assuming that an atomic QoS expression specifies a QoS requirement, thus a complex QoS expression results from the conjunction of two or more atomic QoS expressions. The definition of a complex QoS expression is as follows:

- If α_1 and α_2 are atomic QoS expressions then $\alpha_1 \bigcup \alpha_2$ is a complex QoS expression ε_1.
- Let us suppose that the complex QoS expression ε_2 results from the conjunction $\alpha_1 \bigcup \alpha_2 \bigcup \alpha_3$ of atomic expressions.
- Thus, ε_2 results from the conjunction $\varepsilon_1 \bigcup \alpha_3$.
- A complex QoS expression results from the conjunction of two or more QoS atomic expressions, or other complex expressions.

The QoS expression (notification latency, $\leq$, 2000 ms) $\bigcup$ (event priority, =, 1) specifies that the notification latency must be less than or equal to 2000 milliseconds, and in addition, the highest priority level (i.e., priority = 1) is required.

B. QoS adoption

QoS expressions are translated into constraints that have to be satisfied by the runtime environment. In order to address those QoS requirements, we propose:

- an event processing units placement algorithm that ensures load balance between the available processing devices while minimizing the end to end latency. For simplicity, we will refer to this problem as the operators placement problem, since the placement of an event processing unit is the same as the placement of the operator it implements.
- a strategy applicable to event processing units allowing to ensure that high priority events will be processed and notified earlier than less priority events.

VI. Operator Placement

Event stream processing operators can be deployed in several ways on smart grid devices. Operators placement may considerably impact the quality of service. For instance, deploying operators on a single node may potentially minimize the latency of events processing, since it avoids the time spent to communicate among several nodes. However, concentrating the process on one node may overflow its memory capacity, thus resulting in the violation of a QoS requirement (i.e., memory occupation). This section presents a QoS adoption strategy that addresses the operator placement problem.

A. Problem definition

Operators placement refers to the (close to) optimal selection of the physical nodes hosting the operators in an event processing network in order to satisfy a predefined global cost function. Operators placement is an instance of a more general task-assignment problem that addresses the (close to) optimal assignment of m tasks to n processors in a network, which has an $O(n^m)$ complexity. The operator placement problem is NP-complete.

The operator placement algorithm takes as input a specification of a physical network topology $T = \{N, E\}$, which consists in a set of computing nodes N and their links E. The operator placement also requires a specification of the resources (i.e., memory and CPU) available on each node, and the latency of communication links. Figure 5 shows an example of network topology that comprises 9 computing nodes, each communication link being labeled with its corresponding latency. Table III shows the resources availability on each computing node. In order to specify the exact value of CPU rate available on each node, we specify a coefficient that indicates how fast is that node compared to a reference node for which the CPU coefficient is 1. For example, node n_1 is two times slower than node the reference node n_7, and node n_8 is three times faster than node n_7

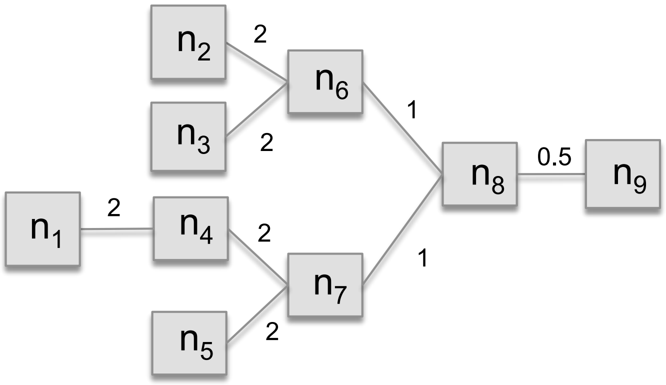

Figure 5. A physical network topology

The operator placement algorithm also takes as input an event processing graph $EPG = < \theta, A >$, which consists in a set of event streams producers $P \in \theta$, a set of stream processing operators $O \in \theta$ and a set of event stream consumers $C \in \theta$. A represents the set of edges that connect the operators in O. Figure 6 shows an example of an event processing graph, where P_1 and P_2 are two producers, C_1 is a consumer, o_1, o_2, o_3 and o_4 are stream processing operators. The operators are associated with measures or estimates of demand, such as the memory and CPU time that each operator expects for processing a single input event. Table IV shows the estimates associated to operators o_1, o_2, o_3 and o_4.

TABLE III. Resources availability on network nodes

Node	Memory	CPU coefficient
n_1	10	1/2
n_2	10	1/2
n_3	15	1/2
n_4	12	1/2
n_5	10	1/2
n_6	10	1/2
n_7	50	1
n_8	60	3
n_9	30	2

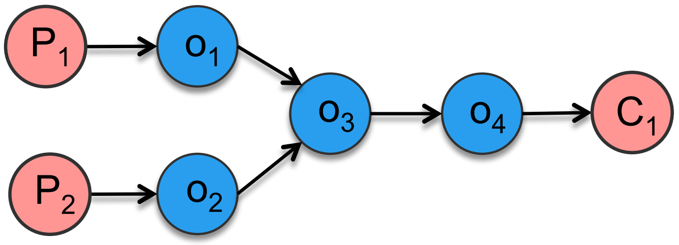

Figure 6. An event processing graph

TABLE IV. Estimates of the operators resources requirements

Operator	Memory	Execution time
o_1	8	4
o_2	12	5
o_3	10	6
o_4	20	9

The output of the operator placement algorithm is a mapping function λ that associates to each operator the node on the network topology in which it should be hosted. Figure 7 shows a possible operator placement, where operators o_1, o_2, o_3 and o_4 are mapped to nodes n_6, n_4, n_8 and n_8, respectively.

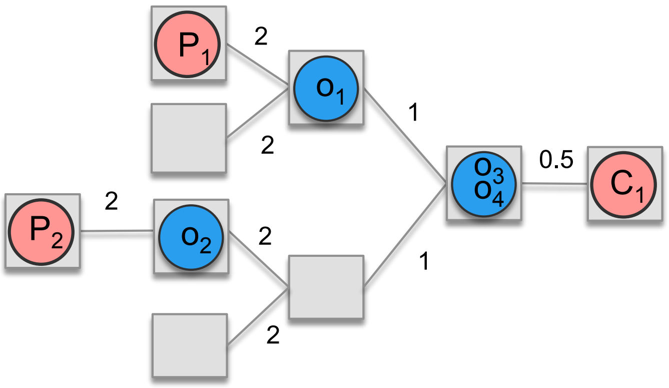

Figure 7. Example of operators placement.

We assume that each producer and each consumer is restricted to a permanent physical network node. Operators can be placed on arbitrary nodes having enough available resources for their execution. In general, a placement algorithm assigns operators to processing nodes in a way that satisfies a set of specified constraints and optimize a given objective function. In our setting, the constraint is to ensure that no processing node is overloaded beyond its memory capacity. The objective function is the expected end-to-end latency between producers and consumers.

B. Problem formalization

In order to formally define the issue of operators placement, let us consider the notations presented in Table V.

TABLE V. NOTATIONS

Operator	Execution time
o	event processing operator
n	network node
$init$	initial mapping of producers and consumers
$time(o)$	execution time of o on a reference node
$mem(o)$	memory required by operator o
$p(o,n)$	execution time of o on node n
$cpuCoef(n)$	CPU coefficient of node n
$amem(n)$	memory available on a node n
$lat(e)$	latency of the network link e
$netPath(n_i,n_j)$	the network path between nodes n_i and n_j
$c(a)$	latency of the communication between operators connected by the edge a
$\lambda(o)$	the mapped location of operator o

We formalize the operator placement problem as follows:

$$\underset{\lambda}{\text{minimize}} \quad cost(\lambda) = \sum_{o\in\theta} p(o,\lambda(o) + \sum_{a\in A} c(a)) \tag{1}$$

subject to:

$$\lambda(o) = init(o),\ if\ o \in P \cup C \tag{2}$$

$$\forall n \in N \sum_{o:\lambda(o)=n} mem(o) \le amem(n) \tag{3}$$

where

$$p(o,n) = \frac{time(o)}{cpuCoef(n)} \tag{4}$$

$$c(a) = \begin{cases} 0 & \text{if for } a=(o_i,o_j),\ \lambda(o_i)=\lambda(o_j) \\ \beta(a) & \text{otherwise} \end{cases} \tag{5}$$

$$a=(o_i,o_j),\ \beta(a) = \sum_{e_i\in netPath(\lambda(o_i),\lambda(o_j))} lat(e_i) \tag{6}$$

Equation (1) states the cost of an operator mapping λ, which is the estimated end-to-end latency incurred by λ. It is calculated as the sum of the latency due to event processing (first part) and the latency due to the network communication (second part). Equation (2) states that the mapping should be consistent with respect to the initial mapping of producers and consumers. Equation (3) states that the mapping should be defined such that no processing node is overloaded beyond its memory capacity. Equation (4) shows the formula that allows to compute the processing time of a mapped operator. Equations (5) and (6) show how to compute the network latency incurred by inter operator communications. By using these formulas, we can compute the cost of the operators mapping presented in Figure 7.

First, note that this mapping is valid, since it does not violate (2) and (3). Following (4), we compute $p(o,\lambda(o))$ for operators o_1 to o_4 as 8, 10, 2 and 3, respectively. The latency of event processing is then 23.

Now let us compute the latency due to inter-operator communications. For the edge (P_1, o_1), it equals 2. For the edge (P_2,o_2), it also equals 2. For the edge (o_1,o_3), it equals 1. For the edge (o_2,o_3), it equals 2+1, so 3. For the edge (o_3,o_4) it equals 0. For the edge (o_4,C_1), it equals 0.5. The latency incurred by inter operator communication is then 7.5. Thus, the total cost of the operator mapping is $cost(\lambda)$ = 23+7.5 = 30.5.

C. Brute force approach

The operator placement problem can be modelled as a constraint satisfaction problem (CSP). CSPs are mathematical problems defined as a set of objects whose state must satisfy a number of constraints. The constraints that we consider are defined by (2) and (3), and are similar to the constraint defined by the bin packing problem, where items of different volumes must be packed into a finite number of bins, each with a given volume. For the purpose of operator placement, the bins represent the processing nodes, and their size represents their memory capacity. The items represent the operators, and their volume represents their memory occupation. We can now rely on a CSP solver to find the set of valid mappings according to the bin packing constraint. The optimal mapping is the one with the minimum cost among the set of valid mappings, as shown in the following algorithm.

```
OpPlacement(EventProcessingGraph epg, NetworkTopology topo, InitialMapping init)
$\lambda_{opt} \leftarrow null$;
$solver \leftarrow BinPackingSolver()$;
solver.constructBinPackingConstraint(epg, topo, init);
if solver.hasSolution() then
    $\lambda \leftarrow solver.nextSolution()$;
    $c \leftarrow cost(\lambda)$;
    $\lambda_{opt} \leftarrow \lambda$;
    while solver.hasSolution() do
        $\lambda \leftarrow solver.nextSolution()$;
        $c_2 \leftarrow cost(\lambda)$;
        if $c_2 < c$ then
            $c \leftarrow c_2$;
            $\lambda_{opt} \leftarrow \lambda$;
        end if
    end while
end if
return $\lambda_{opt}$;
```

D. Greedy approach

The *OpPlacement* algorithm browses the whole space of correct solutions (with respect to the bin packing constraint) in order to find the optimal one. Then, it follows a brute force approach. Because of its exponential complexity, the *OpPlacement* algorithm fails to produce a result in an acceptable period of time for large event processing graphs and network topologies. In order to deal with such large inputs, we propose a greedy approach for operator placement. The idea of this approach is to incrementally map parts of the event processing graph on specific parts of the network topologies, combining found solutions, till all operators are mapped. There are two main aspects that have to be considered here in order to apply this approach. First, it should be specified how to compute parts of the event processing graph. Then, it should be specified how to compute the part of the network topology where a computed part of the event processing graph should be mapped. In order to do that, we rely on the following hypothesis on event processing graph and network topology respectively:

- *Hypothesis 1*: there is one consumer for each input event processing graph. This reduces the complexity of the problem, since considering many consumers in the event processing graph will lead to multi optimization with respect to each consumer, especially when some consumers share the same part of the event processing graph.
- *Hypothesis 2*: the network topology has a tree structure. This is consistent with electrical grid topologies, which are generally designed under a tree structure.

1) Computing subgraphs of the event processing graph: Given the original event processing graph, a subgraph will consist of intermediates operators that are reachable from a given producer to the consumer c. Therefore, they will be the same number of subgraph than the number of producers in the original graph. In the following, we assume the existence of a function $subgraph(EPN, P_i)$ that computes the subgraph associated to the producer P_i. For example, considering the event processing graph in Figure 8, the result of the function $subgraph(EPN, P_2)$ is the subgraph that consists of the set of nodes $\theta' = \{P_2, o_2, o_3, o_4, c\}$ and the set of edges $A' = \{(P_2, o_2), (o_2, o_3), (o_2, o_4), (o_3, c), (o_4, c)\}$.

2) Computing a subgraph of the network topology: Once we compute a subgraph $subgrap(EPN, P_i)$ of an event processing graph for a given producer P_i, we need to compute the subgraph of the network topology where it should be mapped. In order to do that, we consider the mapped location of the producer P_i and the one of consumer c as defined by the initial mapping *init*. The resulting subgraph is the one that includes the nodes in the path between $init(P_i)$ and $init(c)$. Since the network topology is a tree, there is only one path between $init(P_i)$ and $init(c)$. Then, the size of the subgraph is of the order of $O(log(n))$, where n corresponds to the number of nodes in the original network topology. We assume that this subgraph is computed by the function $subgraphTopo(T, n_i, n_j)$.

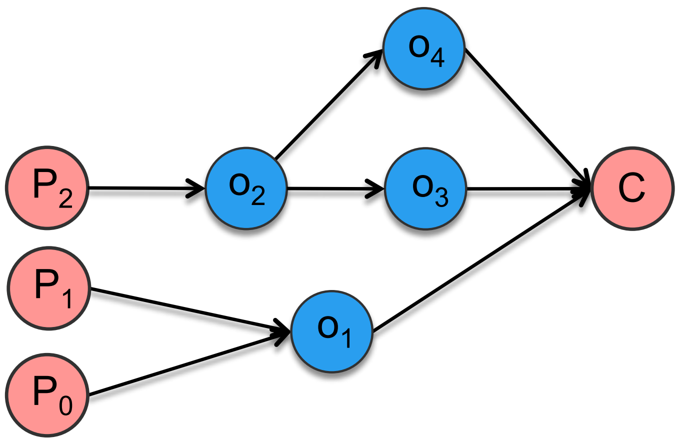

Figure 8. Event processing graph

For example, considering the network topology in Figure 9, and assuming that the producer P_2 and the consumer c are initially mapped at nodes n_6 and n_{10}, respectively, the result of the function $subgraphTopo(T, n_6, n_{10})$ is the subgraph that consists in the set of nodes $N' = \{n_6, n_8, n_9, n_{10}\}$ and the set of edges $E' = \{(n_6, n_8), (n_8, n_9), (n_9, n_{10})\}$.

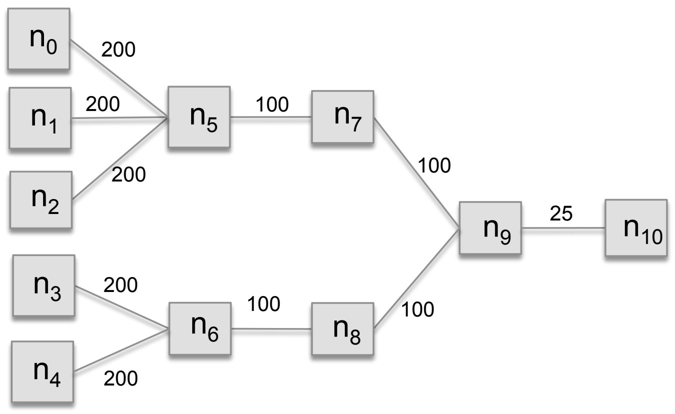

Figure 9. Network topology

3) Greedy algorithm: The greedy version of the algorithm is presented as follows.

```
***OpPlacementGreedy***(EventProcessingGraph epg, NetworkTopology topo, InitialMapping init)
$\lambda \leftarrow init$;
**for** each producer $P_i$ in $epg$ **do**
    $epg' \leftarrow subgraph(epg, P_i)$;
    $topo' \leftarrow subgraphTopo(topo, init(P_i), init(c))$;
    $\lambda' \leftarrow OpPlacement(epg', topo', \lambda)$;
    **if** $\lambda'$ != null **then**
        $\lambda \leftarrow \lambda \bigcup \lambda'$;
        **for** each operator $o$ in $epg'$ **do**
            **if** $o$ is not mapped **then**
                mark $o$ as mappped;
                update the availble memory in $\lambda(o)$;
            **end if**
        **end for**
    **else**
        **return** null;
    **end if**
**end for**
**return** $\lambda$;
```

The *OpPlacementGreedy* algorithm achieves local optimization for each computed subgraph of the original event processing graph. At each step, the solution is combined with the previously found solutions and the result is used like the initial mapping for others iterations. As it finds solutions during subgraph mappings, it marks all non-mapped operators as mapped, and continues till all subgraphs are mapped. If the mapping of a subgraph of the original event processing graph fails, the algorithm stops and the mapping is considered as failed.

VII. Dealing with Event Priority

The QoS requirements concerning memory occupation and latency have already been addressed by the operators placement algorithm presented in the previous section. However, the event priority still remains for being attended, this section presents a proposal to deal with this requirement.

As stated in Section IV, any instance of an event type has a priority attribute to which an integer value must be assigned (See Table I). The priority value of a simple event is defined by its producer, whereas the priority value of a composite event is computed as the maximum priority of its operand events. The higher is the priority value associated to an event instance, the higher is the event priority. Events are inserted into input and output queues according to their priority. Input and output queues are priority-based FIFO structures with limited capacity. The priority relation $\prec$ is defined as follows: $e_i \prec e_j \rightarrow e_i.priority < e_j.priority \vee e_i.priority = e_j.priority \wedge e_j.detectionTime \leq e_i.detectionTime$

The $\prec$ relation ensures that high priority events will be notified early compared to less priority events. As consequence, the notification of a less priority event can be postponed for a significant amount of time. This issue, that we refer to as the starvation problem, is stated more precisely as follows: an event in the output queue may suffer the starvation problem with respect to the notification step, if after a significant number of notifications k, the event is still in the output queue, due to its priority that is less compared to that of inserted events.

To solve the starvation problem, we associated to each event in the output queue a time to live value *ttl* that is initialized to an integer k. At each notification step, the *ttl* value of each event decreases and the events for which the *ttl* value equals zero are notified. For the event priority defined by applications to be really effective, there are also some assumptions that have to be made on the underlying layers of the event processing runtime. More precisely, the middleware layer must provide a FIFO delivery mechanism, allowing to convey events while preserving their notification order such as in [24] [25].

VIII. Experimental evaluation

We focused our experiments to the evaluation of the operator placement algorithm. We developed our algorithm using Java programming language. We rely on the Jacop CSP solver [26] to implement the bin packing constraint. For our experiments, we defined different types of devices (smart meters, data concentrators, sensors, etc.) with different resource profiles. A resource profile is defined by a memory capacity and a CPU coefficient. Based on this, we generated network topologies with various sizes, and containing devices with different defined profiles. The latency of the communication links among the different devices was fixed too. We followed the same idea with operators. We defined different kind of operators with memory and CPU time requirements. We generated event processing graphs of different sizes, and comprising the specified operator types.

We conducted a first experiment to compare the results of the greedy algorithm with those of the brute force algorithm. More precisely, we focused on the algorithm execution time, and the quality of resulting operator placement, which is captured by its cost. We generate 20 different inputs for the algorithm, each consisting in a network topology and an event processing graph. Each network topology consisted in 15 nodes, and the number of operators in each event processing graphs ranged from 7 to 10. For each input, we executed the *OpPlacement* algorithm (brute force) and the *OpPlacementGreedy*. We choose to run this experiment over a small network topology and small event processing graphs in order to make sure that the optimal solution can be calculated.

We compared first the execution time of the algorithms. The result is depicted at Figure 10, which presents for each of the 20 executions (x axis), the time duration (y axis) of the brute force algorithm, and the time duration of the greedy algorithm. Clearly, the greedy algorithm performs faster than the brute force algorithm, being in average one order of magnitude faster.

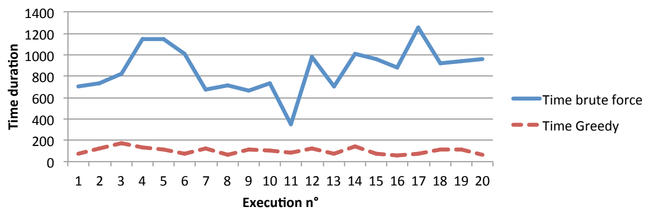

Figure 10. Operator placement execution time: comparison between brute force and greedy algorithm

Figure 11 compares the cost of operator placement computed by the greedy algorithm with the optimal one, computed by the brute force algorithm. We can observe that the cost of the operator mapping computed by the greedy algorithm is generally close to the optimal one. Even more interesting, for this experiment, the accuracy of the greedy approach (computed as the percentage of optimal solutions that were found) was 55%.

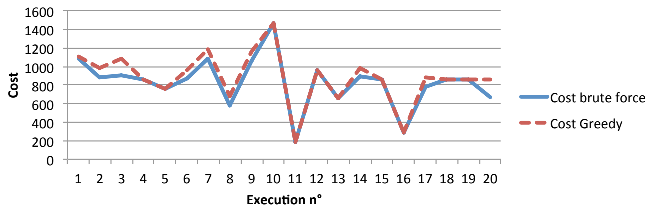

Figure 11. Cost of operator placement: comparison between brute force and greedy algorithm

We conducted another experiment in order to test how the greedy algorithm behaves on large event processing graphs. We execute the algorithm over a network topology consisting in 50 nodes. The size of event processing graphs ranged from 15 to 110 nodes. It is worth to mention that the brute force approach was not able to compute the optimal result here, due to its time complexity. Figure 12 presents the result of this experiment. We notice that the time duration of the greedy algorithm does not necessarily increases when the size of the event processing graph growths. In fact, the structure of the event processing graph is another factor that impacts the performance of the algorithm. In event processing graphs for which operators are highly connected, the subgraph associated to a producer can have a high number of operators. Therefore, the time to compute the mapping of that subgraph can be longer. For example, in the experiment, the event processing graphs having size 45 and 60 were dense, this explains the peaks we observed in the duration time.

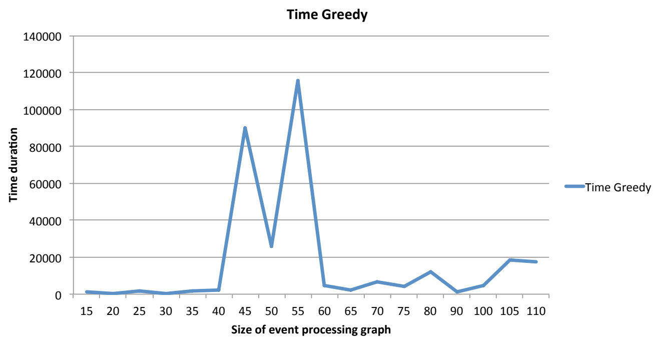

Figure 12. Scale up of the greedy algorithm

IX. Conclusion

This paper shows that monitoring of smart grids can be done using an event based approach where event streams generated by distributed sources are processed by distributed event processing units. Such units may produce complex events indicating situations of interest that are notified to consumers. Since the invocation of business, critical processes is now triggered by events. The QoS of the event processing infrastructure becomes a key issue. We have identified key QoS dimensions relevant to smart grids, namely event priority, memory occupation and notification latency. We proposed a brute force algorithm for deploying event stream operators in a network topology, considering memory occupation and latency. To overcome the time complexity of the brute force approach, we propose a greedy algorithm for operator placement. The experiments shown that, while performing faster than the brute force approach, the greedy algorithm provides good quality solutions. We also proposed a strategy to deal with event priority.

We are currently developping a simulation platform to demonstrate our approach. The simulation platform allows to represent a smart grid topology and an event processing network. In addition, it implements the proposed strategies for QoS adoption. On the other hand, we are working on the specification of a real smart grid use case. The simulation platform will leverage the implementation and validation of the proposed use case.

Network occupation is another QoS dimension relevant to smart grids that was not addressed in this work. As future work, it would be interesting to integrate network occupation as another constraint in our model. The proposed QoS based event stream processing approach can be associated with a language for describing complex event composition with related QoS. This will also be studied in future works.

Acknowledgment

This work was carried out as part of the SOGRID project (www.so-grid.com), co-funded by the French agency for Environment and Energy Management (ADEME) and developed in collaboration between participating academic and industrial partners.

References

[1] O. Epal, C. Collet, and G. Vargas, "Towards a Quality of Service Based Complex Event Processing in Smart Grids," in Proceedings of the 5th International Conference on Smart Grid, Green Communication and IT Energy-aware Technologies (ENERGY). Internatonal Academy, Research and Industry Association, May 2015, pp. 1–4.

[2] G. Cugola and A. Margara, "Raced: An adaptive middleware for complex event detection," in Proceedings of the 8th International Workshop on Adaptive and Reflective Middleware, ser. ARM '09. New York, NY, USA: ACM, 2009, pp. 5:1–5:6.

[3] "Homepage of Esper," 2015, URL: http://esper.codehaus.org/ [accessed: 2015-03-27].

[4] "Homepage of TIBCO StreamBase," 2015, URL: http://www.streambase.com/ [accessed: 2015-03-27].

[5] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: complex event processing over streams," in Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR), 2007, pp. 407–411.

[6] "Homepage of Oracle CEP," 2015, URL: http://www.oracle.com/ [accessed: 2015-03-26].

[7] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system." in Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR). www.cidrdb.org, January 2007, pp. 412–422.

[8] D. Luckham, "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," Stanford University, Tech. Rep., 1996.

[9] O. Saleh and K.-U. Sattler, "Distributed complex event processing in sensor networks," in Proceedings of the 2013 IEEE 14th International Conference on Mobile Data Management - Volume 02, ser. MDM '13. IEEE Computer Society, June 2013, pp. 23–26.

[10] P. R. Pietzuch, B. Shand, and J. Bacon, "A framework for event composition in distributed systems," in Proceedings of the ACM/IFIP/USENIX International Conference on Middleware, ser. Middleware '03, vol. 2672. Springer-Verlag New York, Inc., 2003, pp. 62–82.

[11] "Storm: Distributed and fault-tolerant real-time computation," 2013, URL: http://storm.incubator.apache.org/ [Accessed: 2015-11-10].

[12] S. Behnel, L. Fiege, and G. Mühl, "On quality-of-service and publish-subscribe," in Proceedings of the International Conference on Distributed Computing Systems, 2006, pp. 1–6.

[13] S. Appel, K. Sachs, and A. Buchmann, "Quality of service in event-based systems," in Proceedings of the CEntral EURop Workshop (CEUR), vol. 581, 2010, pp. 1–5.

[14] Y.-h. Jeon, "QoS Requirements for the Smart Grid Communications System," Journal of Computer Science and Network Security, vol. 11, no. 3, 2011, pp. 86–94.

[15] W. Sun, X. Yuan, J. Wang, D. Han, and C. Zhang, "Quality of Service Networking for Smart Grid Distribution Monitoring," in Proceedings of the 1st IEEE International Conference on Smart Grid Communications, 2010, pp. 373–378.

[16] H. Gjermundrø d, D. E. Bakken, C. H. Hauser, and A. Bose, "GridStat: A flexible QoS-managed data dissemination framework for the power grid," IEEE Transactions on Power Delivery, vol. 24, no. 1, 2009, pp. 136–143.

[17] W. Wang, Y. Xu, and M. Khanna, "A survey on the communication architectures in smart grid," Computer Networks, vol. 55, no. 15, Oct. 2011, pp. 3604–3629.

[18] Smart Grid Coordination Group, "Smart grid reference architecture," 2012, URL: http://ec.europa.eu/ [accessed: 2015-03-27].

[19] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," ACM Computing Surveys, vol. 44, no. 3, 2012, pp. 15:1–15:62.

[20] L. Perrochon, W. Mann, S. Kasriel, and D. C. Luckham, "Event Mining with Event Processing Networks," pp. 474–478, 1999.

[21] G. Sharon and O. Etzion, "Event-processing network model and implementation," IBM Systems Journal, vol. 47, no. 2, 2008, pp. 321–334.

[22] "Event processing glossary version 2.0," 2011, URL: http://www.complexevents.com [accessed: 2015-03-27].

[23] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," ACM Computing Surveys, vol. 35, no. 2, 2003, pp. 114–131.

[24] G. V. Chockler and R. Vitenberg, "Group Communication Specifications : A Comprehensive Study," ACM Computing Surveys, vol. 33, no. 4, 2001, pp. 427–469.

[25] A. Malekpour, A. Carzaniga, G. T. Carughi, and F. Pedone, "Probabilistic FIFO Ordering in Publish/Subscribe Networks," in Proceedings of the 10^{th} International Symposium on Network Computing and Applications. Ieee, augst 2011, pp. 33–40.

[26] "Homepage of JaCoP," 2015, URL: http://jacop.osolpro.com [accessed: 2015-03-26].