HAL
open science

NETAH a framework for composing distributed event flows

Orleant Njamen Epal

To cite this version:

Orleant Njamen Epal. NETAH a framework for composing distributed event flows. Databases [cs.DB]. Université Grenoble Alpes, 2016. English. NNT : . tel-01586506

HAL Id: tel-01586506

https://hal.science/tel-01586506

Submitted on 12 Sep 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Orléant EPAL NJAMEN

Thèse dirigée par **Christine Collet** et
Co-encadrée par **Genoveva Vargas-Solar**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale de Mathématiques, Sciences et Technologies de l'Information et de l'Informatique**

NETAH, un framework pour la composition distribuée de flux d'événements

Thèse soutenue publiquement le **11 Octobre 2016**,
devant le jury composé de :

Dr Béatrice, FINANCE
MCF HDR Université de Versailles Saint Quentin, Rapporteur
Prof. Jean-Marc, PETIT
Professeur INSA-Lyon, Rapporteur
Prof. Michel,RIVEILL
Professeur Polytech'Nice Sophia, Examinateur
Dr Fabienne, BOYER
MCF HDR Université Grenoble Alpes, Examinateur
Prof. Christine, COLLET
Professeur Grenoble INP, Directrice de thèse
Dr Genoveva VARGAS SOLAR
CR HDR CNRS, Co-Encadrante de thèse

Rien n'est plus insondable
que le système de motivations
derrière nos actions...
— Georg Christoph Lichtenberg

À mes chers parents que j'aime si fort...
Mr Njamen Daniel & Mme Yateu Bénédicte

Remerciements

C'est rempli d'émotion que je souhaite par ces quelques mots exprimer ma gratitude aux personnes et entités qui par leur soutien ont contribué de près ou de loin à l'aboutissement de ce travail:

Mes directrices de thèse Christine Collet et Genoveva Vargas-Solar. Vos questions et vos conseils n'ont cessé de me recadrer et m'encourager dans mon travail. Je ne saurais assez vous remercier de l'opportunité que vous m'avez donné de travailler à vos côtés depuis mon stage de master jusqu'à la soutenance de cette thèse.

Je tiens aussi à remercier les rapporteurs qui ont pris le temps d'évaluer ce travail, pour leur remarques et suggestions, et les membres du jury pour leur disponibilité.

Mes sincères remerciements aux membres des équipes HADAS et SLIDE avec qui j'ai collaboré et échangé durant mes années de doctorat.

À toute la communauté camerounaise de Grenoble (ACI), qui a contribué à rendre mon séjour à Grenoble très agréable. Merci pour le soutien permanent.

À vous, Christiane, Juliette, Léonie, Alain, Léon, Serge, Fred, ainsi qu'à tous les amis rencontrés sur mon chemin. Merci!

J'ai une pensée toute particulière pour les membres de ma famille: Boris, Hervé, Aristide, Christelle et Madrid; Je ne vous remercierai jamais assez. Plus qu'un soutien, vous avez été pour moi une motivation. Vous m'avez toujours accordé votre faveur, chacun à sa manière, sans jamais perdre patience.

Je ne pourrai finir sans mentionnner ma fiancée, Priscille qui n'a cessé de me motiver et m'encourager depuis notre rencontre. Merci!

Grenoble, 25 Juillet 2016

Orléant

Abstract

This thesis proposes NETAH, a framework for the construction of event stream composition services in highly distributed contexts with limited resources such as smart grid, the Internet of Things, etc. An event stream composition service instanciated using NETAH is a network of event processing units, which can be deployed on a target architecture composed by a set of connected processing nodes having different capacities. NETAH implements a model for the representation and composition of different types of event streams, with appropriate composition operators. Each event processing unit corresponds to a event stream composition operator. NETAH provides an algorithm for the deployment of event processing units. The deployment of event processing unit considers the resources available on network nodes, as well as the latency of communication links, in order to satisfy the quality of service. Our approach exploits the computing resources of heterogeneous devices distributed over the network.
The NETAH framework has been instantiated to implement event stream composition networks in the context of smart grid monitoring. This experiment demonstrates the validity of our proposal for event stream composition in the distributed and constrained environment that is a smart grid.

Keywords: event stream processing, complex event processing, distributed systems, smart grids, Quality of service .

Résumé

Cette thèse propose NETAH, un framework pour la construction de services de composition distribuée de flux d'événements dans des contextes hautement distribués et contraints en ressources comme les smartgrids, l'internet des objets, etc. Un service instancié en utilisant NETAH est un réseau d'unités de traitement qui peut se déployer sur une architecture cible composée de noeuds de calcul avec des capacités différentes, connectés entre eux. NETAH implémente un modèle pour la représentation et la composition de différents types de flux d'événements, avec des opérateurs de composition adaptés. Chaque unité de traitement correspond à un opérateur de composition de flux d'événements. NETAH fournit un algorithme de déploiement d'unités de traitement. Le déploiement des unités de traitement prend en compte les ressources disponibles sur l'environnement d'exécution cible (dispositifs de calcul, réseau) afin de satisfaire la qualité de service (QoS) des applications. Notre approche exploite les ressources de calcul des dispositifs hétérogènes et hautement distribués sur le réseau.
Le framework NETAH a été instancié pour implanter un réseau de composition de flux d'événements dans le contexte du monitoring d'un smartgrid. Cette expérimentation démontre la validité de notre proposition pour la composition de flux d'événements dans les contextes distribués et contraints que sont les smartgrids.

Mots clefs : Composition de flux d'événements, traitement d'événements complexes, systèmes distribués, qualité de service, smartgrid.

Contents

Remerciements iii

Abstract (English/Français) v

List of figures xi

List of tables xv

1 Introduction 1
- 1.1 Context and Problem . . . 1
- 1.2 Objective and approach . . . 2
- 1.3 Main contributions . . . 3
- 1.4 Document organization . . . 4

2 Event Stream Composition Systems 7
- 2.1 Stream processing . . . 8
  - 2.1.1 Event/Stream processing model . . . 10
  - 2.1.2 Deployment model . . . 12
  - 2.1.3 QoS requirements . . . 16
- 2.2 Existing projects and systems . . . 17
  - 2.2.1 Event/Data stream processing systems . . . 18
  - 2.2.2 Complex event processing systems . . . 26
- 2.3 Discussion . . . 31
- 2.4 Conclusion . . . 32

3 The NETAH Event Stream Composition Model 35
- 3.1 Preamble . . . 36
- 3.2 Event, event type, event streams . . . 37
  - 3.2.1 Event . . . 37
  - 3.2.2 Event type . . . 38
  - 3.2.3 Event stream . . . 41
- 3.3 Event stream composition operators . . . 41
  - 3.3.1 Filter . . . 42
  - 3.3.2 Disjunction . . . 42
  - 3.3.3 Conjunction . . . 43
  - 3.3.4 Sequence . . . 43
  - 3.3.5 Window operators . . . 44

3.3.6 Aggregation operators . . . 46
3.3.7 Selection operators . . . 47
3.3.8 Flatten . . . 48
3.3.9 Computing the meta-attributes of events in output streams . . . 48
3.4 Event stream composition . . . 49
3.4.1 Complex event stream, simple event stream . . . 49
3.4.2 Event stream composition expression . . . 49
3.4.3 Labelled event stream composition expression . . . 50
3.4.4 Well formed event stream composition expression . . . 50
3.4.5 Representing an event stream composition expression as a directed graph 50
3.4.6 Developed form of an event stream . . . 52
3.5 QoS requirements . . . 52
3.5.1 QoS expression . . . 53
3.5.2 QoS tagged event stream composition expression . . . 53
3.6 Conclusion . . . 54

4 The NETAH Framework 55
4.1 Overview . . . 56
4.2 Subscription . . . 58
4.3 Event stream composition network . . . 59
4.3.1 Producer . . . 59
4.3.2 Consumer . . . 60
4.3.3 Event processing unit . . . 61
4.4 Runtime environment . . . 64
4.5 Event stream composition network creation . . . 65
4.6 Event processing unit mapping . . . 68
4.7 Event processing unit deployment . . . 73
4.8 Conclusion . . . 75

5 NETAH in Smart Grids 77
5.1 Smart grid and NETAH . . . 78
5.1.1 Overview of a smart grid . . . 78
5.1.2 NETAH framework in a smart grid . . . 80
5.2 Operator mapping algorithm . . . 80
5.3 Simulating the smart grid network . . . 85
5.3.1 Routing data over the network . . . 86
5.3.2 Implementing a publish/subscribe communication style . . . 87
5.4 NETAH in the simulated smart grid network . . . 89
5.4.1 Defining an event stream composition scenario . . . 90
5.4.2 The graphical user interface (GUI) . . . 94
5.5 Conclusion . . . 95

6 NETAH for Location of Resistive Failures **97**
6.1 Overview of the energy supply chain 98
6.1.1 Source station 98
6.1.2 HTA/BT substation 98
6.2 Detection and location of resistive faults 100
6.2.1 The general practice of fault detection and location 100
6.2.2 Resistive fault 101
6.3 Implementation 101
6.3.1 Event stream compositions 102
6.3.2 Simulating a resistive fault detection and location scenario 105
6.3.3 Defining the smart grid topology 110
6.3.4 Running the scenario 111
6.4 Experimental results 115
6.5 Conclusion 115

7 Conclusions and perspectives **117**
7.1 Summary 117
7.2 Perspectives 118

A Experimental evaluation of the operator mapping algorithms **121**

Bibliography **130**

List of Figures

1.1 Approach overview 3

2.1 A microgrid. 8
2.2 Generic architecture of a stream processing system 9
2.3 Processing models 10
2.4 Implementation of an event stream processing as a DAG of processing elements 11
2.5 Implementation of complex event detection with finite state automata 11
2.6 Implementing an event stream processing with Petri nets 12
2.7 Implementing the query **Q1** using a dataflow model 12
2.8 Deployment models 13
2.9 A centralized stream processing model 14
2.10 Distributed deployment models 14
2.11 A clustered stream processing model 15
2.12 A networked stream processing model 16
2.13 Example of a dataflow graph in Aurora 21
2.14 Example of a *Storm* topology 23
2.15 Example of a dataflow graph in *Samza*. 25

3.1 Example of an event type 39
3.2 Example of an event instance 40
3.3 Example of prioritized event instances 41
3.4 Operator synopsis 42
3.5 Example situation 1. The time associated to events represents the time at which the events are processed by the operator. 43
3.6 Example situation 2. The time associated to events represents the time at which the events are processed by the operator. 44
3.7 Example situation 3. The time associated to events represents the time at which the events are received by the operator. 45
3.8 Example situation 4. The time associated to events represents the time at which the events are processed by the operator. 46
3.9 Example of a graph reresentation of an event stream composition expression. . 52

4.1 Overview of NETAH 56
4.2 Architecture of NETAH 57
4.3 Class diagram of a NETAH node 58
4.4 Class diagram of a subscription 59

4.5 An event stream composition network 60
4.6 A producer 60
4.7 Class diagram of a producer 60
4.8 A consumer 61
4.9 Class diagram of a consumer 61
4.10 Event processing unit 62
4.11 Class diagram of an event processing unit 62
4.12 Example situation where a selection policy should be applied 63
4.13 The runtime environment 64
4.14 Creation of EPUs from a subscription. 66
4.15 Graph associated to the event stream composition expression *expr*. 67
4.16 The event stream composition network associated to the subscription *s*. 68
4.17 Operator mapping 69
4.18 A physical network topology 70
4.19 Example of an event stream composition network 70
4.20 Initial mapping of producers and consumers. 71
4.21 Example of EPU mapping. 71
4.22 Deployment of event processing units in the runtime environment 73
4.23 The deployment sequence 74

5.1 A smart grid 79
5.2 The NETAH approach in smart grid 81
5.3 Overview of the greedy approach 82
5.4 Example of an event stream composition network 83
5.5 Network topology 84
5.6 Class diagram: the smart grid environment 85
5.7 Class diagram: producer 87
5.8 Topic based publish/subcribe broker 88
5.9 Sequence diagram: subscribing to a topic 89
5.10 Sequence diagram: publishing an event 89
5.11 NETAH for event stream composition in a simulated smart grid network 90
5.12 Base class for defining a producer 91
5.13 Base class for defining a consumer 91
5.14 Base class for defining a Subscription 92
5.15 Directed acyclic graph associated to the subcription created by Listing 5.3 . . . 93
5.16 The class *Simulation* 94
5.17 The simulator graphical user interface 95

6.1 Simplified view of a power grid 98
6.2 A source station 99
6.3 A HTA/BT substation 99
6.4 Fault detection on a HTA network 100
6.5 Processing of a DC subscription 104
6.6 Processing of the SIT-R subscription 106

6.7 Example of an input file for the slave station 107
6.8 Example of an input file for the HTA coordinator 108
6.9 The smart grid topology of the simulation 112
6.10 Event stream composition networks for resistive faults detection and location . 114
6.11 Detection and location of a resistive fault 116

A.1 Operator deployment execution time: comparison between brute force and greedy algorithm 122
A.2 Cost of operator mapping: comparison between brute force and greedy algorithm 122
A.3 Scale up of the greedy algorithm 122

List of Tables

2.1 Comparison of existing systems 32

3.1 Event type meta-attributes 39
3.2 Meta-attributes of prioritized event types 40

4.1 Parameters associated to stream operators defined in *expr*. 67
4.2 Resources availability on computing nodes 70
4.3 Estimates of the EPU resource requirements 71
4.4 Notations 72

6.1 QoS parameters of the subscription 105
6.2 Resources availability on smart grid devices 111

1 Introduction

1.1 Context and Problem

Many applications need to process event flows generated from a large number of geographically distributed sources, to obtain timely responses to complex queries. Examples of such applications come from the most fields, including environmental monitoring, finance, smart grids, smart cities, the Internet of Things, etc.
In environmental monitoring, users need to process data coming from sensors deployed in the field to acquire information about the observed world, detect anomalies, or predict disasters as soon as possible [BCMR09]. Similarly, several financial applications require a continuous analysis of stocks to identify trends [DGH+06]. In new application domains such as the Internet of Things (IoT) and smart grids, monitoring applications need to observe and process event streams generated by a large variety of devices to detect and notify critical situations such as faults, alerts, etc.

Event stream processing systems are well adapted to the programmation of such applications. The components of an event stream processing system communicate by producing and consuming event streams, where an event is the notification that a happening of interest has occurred [VSC02]. An event stream composition service mediates producers and consumers enabling loosely coupled communication among them. Producers publish event streams to the service, and consumers express their interest in receiving certain types of event streams by issuing subscriptions. A subscription is seen as a continuous query that allows consumers to obtain event stream notifications. The service is then responsible for matching received event streams with subscriptions and conveying event stream notifications to all interested consumers.

New application domains such as the IoT and smart grids consist in a large number of computing devices (sensors, actuators, smart meters, data concentrators, etc.) with limited computing resources (memory, CPU), connected by low capacity networks connections.
For example in a smart grid, the computing devices include smart meters, sensors, actuators, data concentrators, servers and so on. These computing devices have different resource profiles, ranging from low memory and CPU devices like sensors, actuators and smart meters, to high memory and CPU devices like data concentrators and computers. Similarly, the communication support connecting those devices includes low capacity network technologies like

power line communication or radio, and high capacity network like ethernet or WIFI.

A vast amount of event streams are produced from the large variety of devices deployed within such environments. The monitoring of these environment relies on the capacity to observe, filter, aggregate, and compose those event streams in order to detect interesting situations and notify them to interested applications. This requires the definition of event stream composition systems that can be deployed in large scale distributed environments. These systems must efficienly achieve event stream filtering, correlation, aggregation and composition while adapting to the environment in terms of the multiplicity of data sources, the heterogeneity of computing resources (memory, CPU, network) and the application quality of services (e.g, notification latency, event priority).

In this thesis, we address the problem of efficiently achieving event stream composition in highly distributed environments with limited computing resources. This implies accessing and processing distributed event streams produced by different sources and notify the results to interested parties, considering both the limitations of the runtime environment and the application QoS requirements.

Our research is motivated by the observation that current event stream composition systems are not appropriate in highly distributed contexts having limited computing devices. Centralized systems like [ABB+03, ACc+03, WDR06, Esp15] present scalability and single point of failure issues in these contexts. Clustered systems like [Aba05, SMMP09, STO13, ZCF+10, Apa16c, Apa16a] resolve the issues of centralized systems by achieving the event stream processing within a cluster of machines, ensuring high availability and a better scalability. Nevertheless, clustered systems require the event streams to be routed to the remote cluster for processing. This is not efficient from a network point of view as it induces latency, especially on limited network connections. In addition, Clustered solutions consider that there is enough computing resources on the processing nodes, an assumption which is not feasible in contexts like the IoT and smart grids, where limited capacity devices (sensors, smart meters, etc.) are part of the runtime environment.

1.2 Objective and approach

In this thesis, we aim to leverage the computing resources offered by devices deployed in the environment to enable a large scale distributed event stream composition that deals with QoS. Our objective is to provide methods and tools for the programmation of applications that need to process event streams from distributed sources, considering both the QoS requirements of applications (e.g, event priority, latency, etc.) and the physical constraints of the heterogeneous environment (e.g, resources limitation of compute nodes, network limitations).

Our approach is summarized in Figure 1.1. It consists in three layers of abstraction, namely the runtime environment, event streams, and event stream composition network layers.

- The environment layer represents the distributed runtime environment, which consists of a set of heterogeneous devices connected by communication networks technologies.

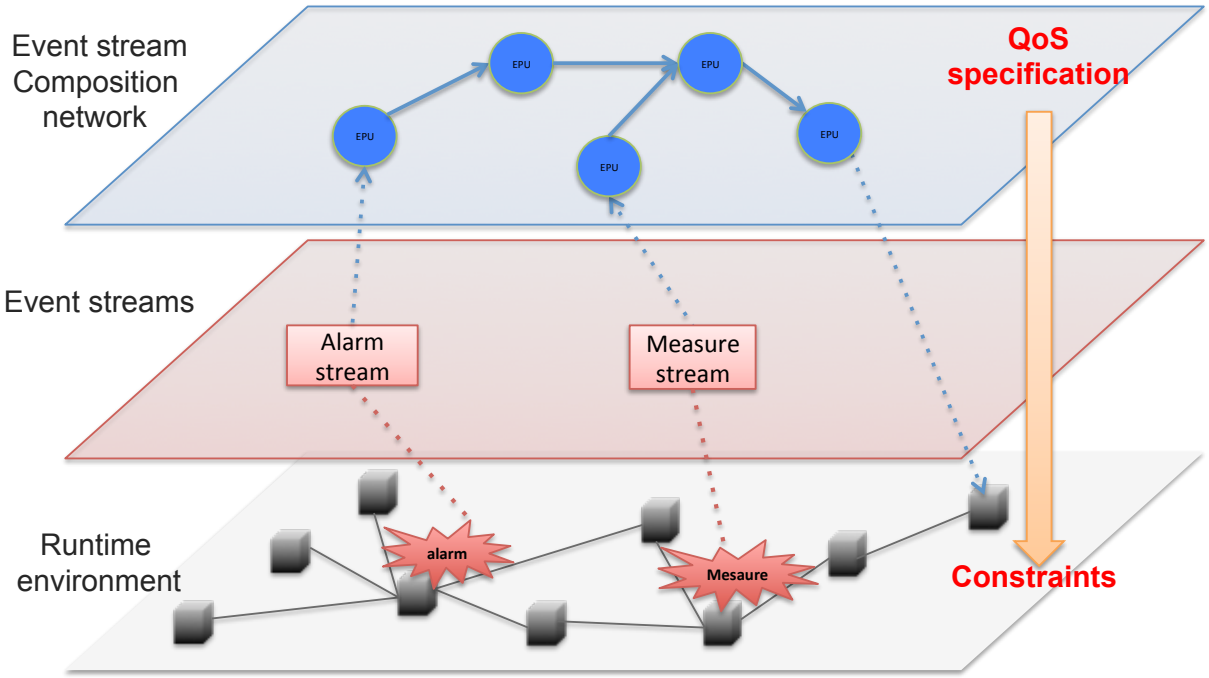

Figure 1.1 – Approach overview

The runtime environment is described in terms of information being used and exchange between functions, services and components. Those information are seen as events that happen within the runtime environment.

- The event streams layer considers that data generated within the environment are event streams. In this layer, some devices can act as producers that generate different types of events in a continuous manner.

- The event stream composition network layer consists in a set of connected event processing units. This network is created according to complex event stream subscriptions. It may be deployed across multiple distributed computers and physical networks. The complex event stream subscriptions are tagged with applications QoS requirements such as event priority and notification latency. Those QoS requirements are translated into constraints applicable to event processing units at execution time. In addition to these constraints, inherent constraints of the runtime environment are also considered, such as limitations on computational resources (i.e., memory and CPU) and / or communication networks (i.e., network latencies).

1.3 Main contributions

The main contributions of this thesis are:

- *An event stream composition model.* We propose an event stream composition model

that proposes a set of concepts for the representation of event streams, and a set of operators that operate on such event streams. We also propose a way of defining event stream composition expressions representing complex event stream processing queries, and the associated QoS requirements.

- *An event stream composition framework.* We propose an event stream composition framework for the generation and deployment of event stream composition networks in distributed and constrained runtime environments, considering QoS requirements of applications. Our framework derives an event stream composition network from an event stream composition expression. The event stream composition network consists in a set of connected event processing units that implement stream operators. We rely on a mapping algorithm for efficiently deploying event processing units on the distributed runtime environment, considering QoS requirements such as latency and memory occupation. We model the QoS requirements as a set of constraints that have to be satified by such an algorithm. The proposed constraints are extensible, allowing to take into account new QoS requirements.

- *Event stream composition in a smart grid.* We specialize the framework in the context of a smart grid. We propose an algorithm for efficiently mapping event processing units on smart grid devices, considering QoS dimensions memory, CPU and latency. The specialized framework allows to build event stream composition networks in a smart grid. We simulate the smart grid runtime environment, on top of which we implement a business use case which validates our proposition.

1.4 Document organization

The remainder of this document is organized as follows.

Chapter 2 presents the background and existing works related to event stream processing. The stream processing domain is presented as well as existing projects and systems, with an emphasis on the adopted processing model, deployment architecture and QoS support.The chapter concludes with a discussion on the inadequacy of existing solutions in distributed execution contexts with constrained resources.

Chapter 3 presents our event stream composition model. It starts with our model for the representation of event streams in distributed contexts. Then, event stream composition operators are presented, followed by our model for expressing event stream composition expressions and the associated QoS requirements.

Chapter 4 presents NETAH, our framework for building and deploying event stream composition networks in distributed contexts that are constrained in resources. First the chapter presents the architecture of the framework, with the strategies to deal with event priority. Then, it presents how to generate an event stream composition network from a user subscription, as well as the deployment process of event stream composition networks in the constrained runtime environment.

Chapter 5 presents the specialization of NETAH in smart grids. It starts with an overview of a smart grid, and then presents how to adopt NETAH in such a context. Then, it proposes an algorithm for mapping event processing units to smart grid devices considering QoS constraints in term of memory, CPU, latency. Finally, it presents a simulation of the smart grid environment, which includes an API allowing to implement event stream composition scenario in that environment.

Chapter 6 validates our proposal with the implementation of a use case related to the location of resistive faults on the medium voltage network of a smart grid. It presents the event stream composition expressions that models the detection of a resiste fault and its location, with the associated QoS requirements. Then, it presents the use of NETAH for generating and deploying the corresponding event processing network in a smart grid topology. Finally, it discusses the experimental results.

Chapter 7 presents our conclusions and perspectives providing a discussion on the challenges that remain and perspectives.

2 Event Stream Composition Systems

This chapter presents existing works related to event stream processing. Section 2.1, introduces the general stream processing domain,comprised of data stream processing systems and complex event processing systems. Section 2.2, compares existing systems from the two categories, considering both their processing and deployment models, and the QoS support. Section 2.3, discusses the limitation of current systems to fulfill our objectives, and finally, Section 2.4, concludes this chapter.

Contents

2.1 Stream processing 8
2.2 Existing projects and systems 17
2.3 Discussion 31
2.4 Conclusion 32

2.1 Stream processing

The stream processing domain consists in a class of systems that process continuous event (data) streams. They arise in telecommunications, health care, financial trading, and transportation, among other domains. Timely analysis of such streams can be profitable (in finance) and can even save lives (in health care). In the streaming model, events arrive at high rate, and algorithms must process them under very strict constraints of space and time. Furthermore, often the events volume is so high that it cannot be stored on disk or sent over low capacity networks before being processed. Instead, a stream processing system can analyse continuous event streams immediately, reducing large-volume input streams to low-volume output streams for further storage, communication, or action.

For example, let us consider a microgrid[1] which provides electricity to households using an intelligent energy distribution system (see Figure 2.1). Households are equipped with smart meters that measure each minute the energy consumption. The event streams generated by smart meters are sent to the energy monitoring via the power line communication network. Each event indicates the ID of the smart meter[2], the energy consumed and the timestamp. Let us assume this query from the network monitoring:
Q1: Notify me whenever the energy consumed either at house *h1* or *h2* on the last hour is greater than 10 kWh.

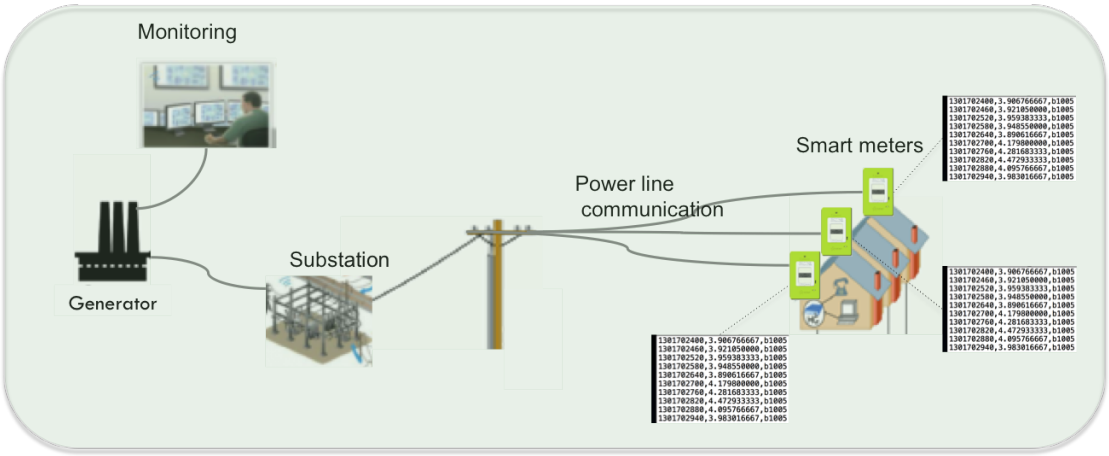

Figure 2.1 – A microgrid.

In order to answer that query, the event stream processing system should continuously apply filtering, windowing, aggregation and event stream composition in order to produce the desired outputs and notify them to the network monitor.

Figure 2.2 presents a generic architecture of a stream processing system.

The system provides an interface where users can express continuous queries or rules using

[1] A small independent smart grid.
[2] We assume that this also identifies the house

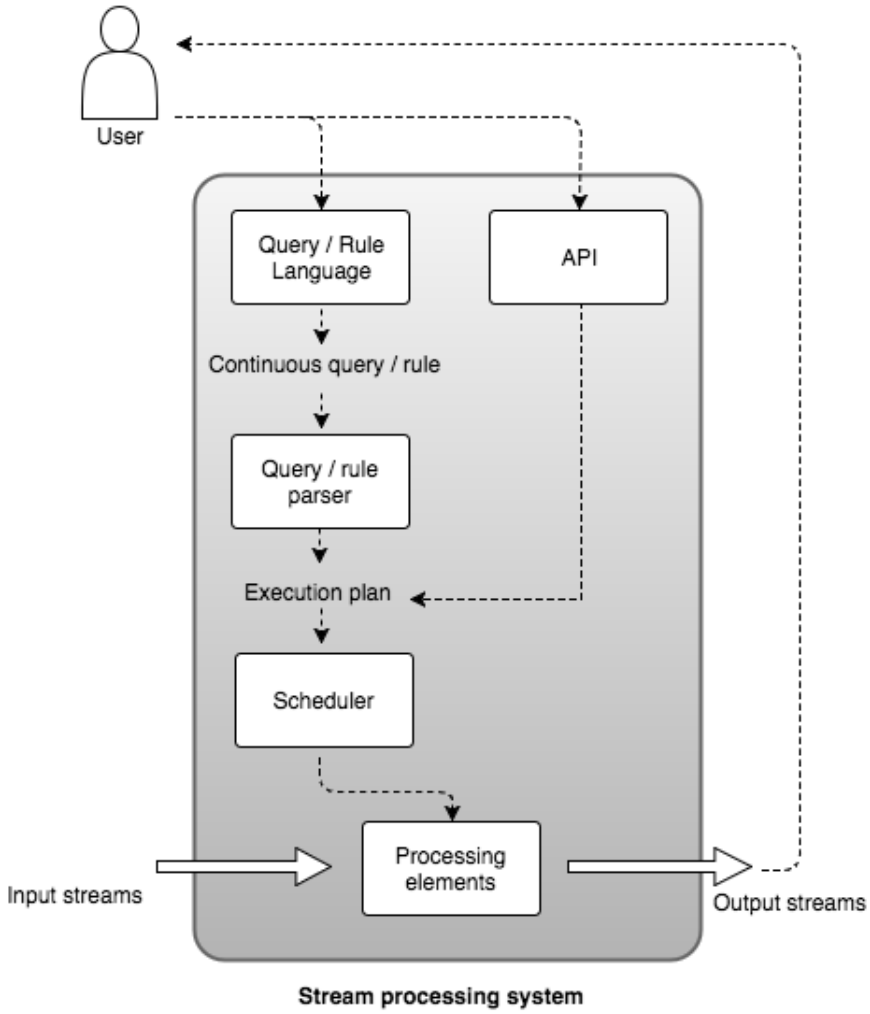

Figure 2.2 – Generic architecture of a stream processing system

a declarative adhoc language [CM94, LS03, ABW06, JMS+08, CM10]. Such queries are then processed by a *query parser* that generates an execution plan[3]. The stream processing system may also expose an API allowing users to directly program execution plans and submit them to the system [ZCF+10, STO13, Apa16a]. The execution plan is then processed by a *scheduler*, which creates and deploys the *processing elements* which actually process the input streams according to the received execution plan. The result of such processing is an output stream, which is notified to the user.

We distinguish between two categories of stream processing systems. The first one, *data stream processing systems*(DSP), from the database community, focus on extending existing database technologies and systems to deal with the characteristics of streams (continuous, unpredictable rates, real time requirements, etc) [ABB+03, CDTW00, ACc+03]. The goal of such systems is to execute SQL-like queries on continuous data streams. The second one,

[3]For sake of simplicity, we suppose the execution plan is optimal, such that we don't need a query optimizer.

complex event processing systems (CEP), from the middleware community, focus on extending publish/subscribe middleware, allowing applications to subscribe to complex event patterns within the event streams. The goal of such systems is to efficiently extract complex event patterns within the event streams and notify them to interested applications.
In order to take benefits of both system categories, they have been integrated in some recent systems [Apa16a, Esp15].

The remainder of this chapter focusses on three dimensions that characterize existing stream processing systems: their processing model, deployment model and QoS support.

2.1.1 Event/Stream processing model

The *processing model* of a stream processing system refers to the way it implements stream processing. We distinguish between three kinds of processing models (see Figure 2.3): *dataflow*, *finite state automata*, and *petri nets*.

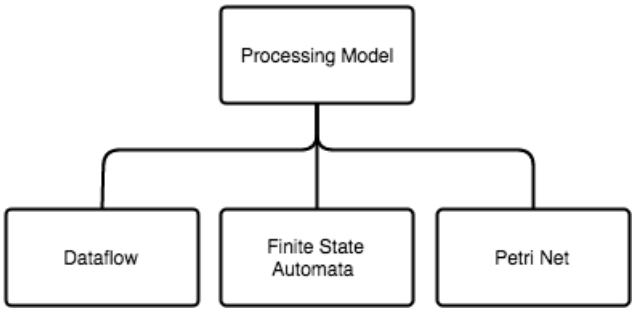

Figure 2.3 – Processing models

Dataflow

A dataflow model considers that the stream processing system is implemented by a set of concurrent computational actors that transform and exhange data from the input sources to the final destination. Such a model is well adapted for systems which need to apply continuous transformations on data streams in order to produce results [ZCF+10, STO13, ABB+03, CDTW00, ACc+03]. The event stream processing is implemented by a set of processing elements (PEs) organized as a directed acyclic graph(DAG), as illustrated in Figure 2.4. Upstream to the DAG are the sources from which emanate event streams, and downstream is the application which collects final results. Each processing element in the DAG implements an operation, and forwards its results either to another processing element, or to an application. Basically, a processing element can implement any operation. Examples of operations are filters, windows, joins, aggregations, pattern matching, or user defined operations.

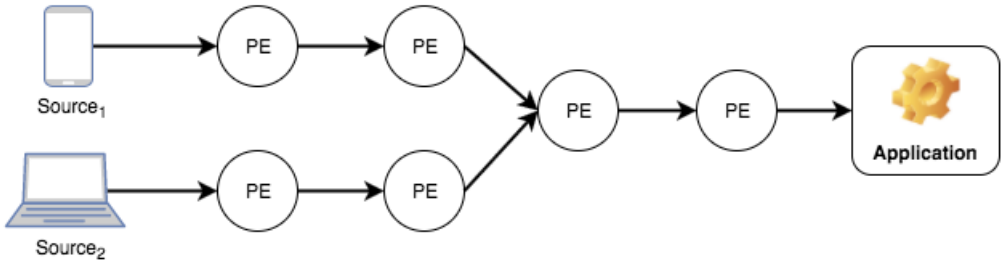

Figure 2.4 – Implementation of an event stream processing as a DAG of processing elements

Finite state automata

Finite state automata (FSA) are well adapted for systems that need to detect complex event patterns over event streams. The event detection is implemented using finite state automata [ADGI08, WDR06].

Complex event patterns are expressed using operators like conjunction, disjunction, negation, sequence, windows, iterations, etc. A complex event pattern represents a complex event type, and is associated to an automaton. A state of such automaton represents a partial match of the complex event pattern. The transition from one state to another represents a condition that should be satistied on the input event stream in order to apply the transition. The incoming event streams provides the sequence of input events to the automaton. If the automaton reaches an accepting state, then the complex event implemented by the automaton occurs. Figure 2.5 illustrates the finite state automata for the complex event type $(A \wedge B;\ C)$ that occurs whenever instances of A and B event types occur, followed by an instance of the C event type.

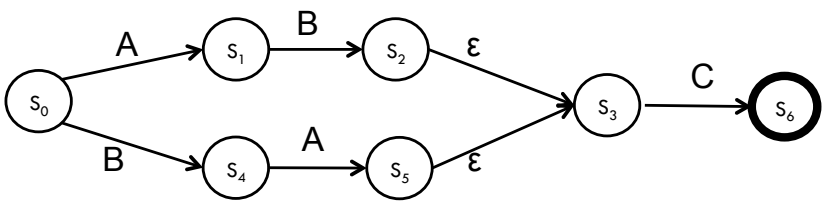

Figure 2.5 – Implementation of complex event detection with finite state automata

Petri nets

Some systems adopted Petri nets for detecting complex event patterns within event streams [CKAK94, Hin03, GD94, JLKY08]. A Petri net consists of places, transitions and arcs. Arcs connect places with transitions and transitions with places. The places of a Petri net correspond to the potential states of the net, and such states may be changed by the transitions. Transitions correspond to the possible events which may occur (perhaps concurrently). In Coloured Petri nets, tokens are of specific token types and may carry complex information.
When an event occurs, a corresponding token is inserted into all places representing its event

type. The flow of tokens through the net is then determined; a transition can fire if all its input places contain at least one token. Firing a transition means removing one token from each input place and inserting one token into each output place. The parameters corresponding to the token type of the output place are derived at that time. Certain output places are marked as end places, representing complex events. Inserting a token into an end place corresponds to the detection of a complex event. Figure 2.6 presents the Petri net for the sequence $E_1; E_2$.

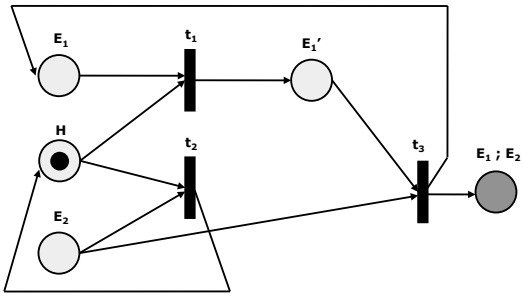

Figure 2.6 – Implementing an event stream processing with Petri nets

The choice between each of these processing models (dataflow, finite state automata and Petri nets) depends on the target application. For example, the query **Q1** defined in Section 2.1 requires continuous transformation of event streams (windows, aggregations and filters) rather than event pattern recognition. Therefore, the dataflow model is the natural way to implement such a query, as illustrated in Figure 2.7. Each smart meter produces an input

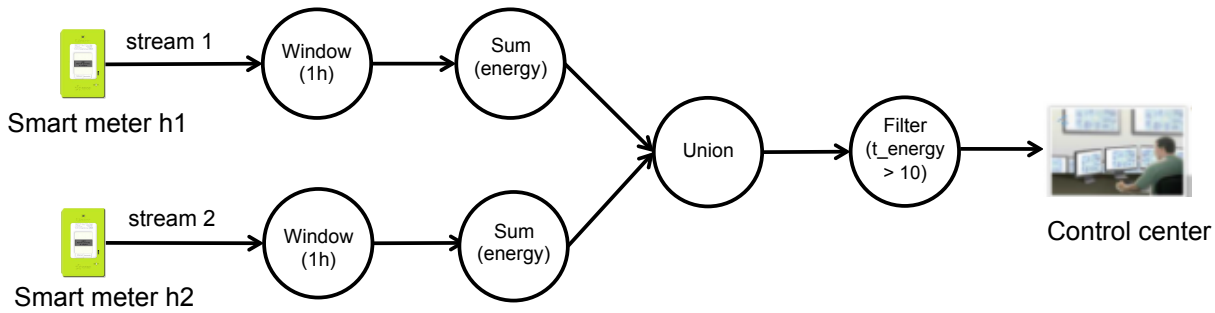

Figure 2.7 – Implementing the query **Q1** using a dataflow model

stream, on which we compute one hour sliding windows. Then, we compute the sum of the energy on each window. Then, we merge the resulting output streams into one using an "union" operation. Then, we apply a filter on the merged stream to consider only energy sum that are greather than 10 kWh. The resulting output stream is notified to the control center.

2.1.2 Deployment model

Several stream processing applications include a large number of sources and sinks, possibly distributed over a wide geographical area, producing and consuming a large amount of event

streams that have to be processed by the stream processing engine in a timely manner. Hence, an important aspect to consider is the deployment architecture of the engine, that is, how the processing elements that implement the event stream processing are deployed on target physical architectures. We distinguish between *centralized* and *distributed* models (see Figure 2.8).

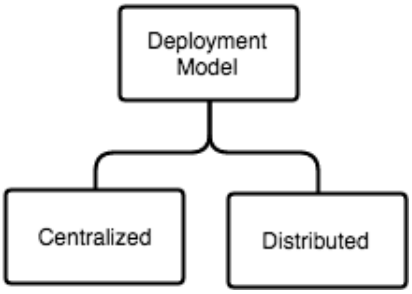

Figure 2.8 – Deployment models

Centralized model

In a centralized architecture, the stream processing is realized by a single node in the network, in a pure client-server architecture. The stream processing engine acts as the server, while event producers and consumers act as clients [ABB+03, ACc+03, WDR06, Esp15]. For example, Figure 2.9 presents the adoption of a centralized architecture for the implementation of stream processing within the microgrid (see Section 2.1). The event streams are conveyed to the stream processing server, which implements the stream processing query *Q1* and notifies the results to the control center.

The main issue with a centralized model is the fact that event streams have to be acheminated to the server in order to be processed, which is network consuming. In addition the server, which can face scalability issues in large scale settings, is a single point of failure.

Distributed model

There is an increasing interest in extracting useful information from large scale, distributed data streams, in realtime (smart grids, internet of things, etc.). Motivated by the fact that centralized stream processing systems failed to scale in presence of large stream sources, distributed stream processing systems have been investigated.
A distributed event stream processing system organizes the event stream processing in a set of tasks which are executed accross different nodes of a computer network and collaborate to produce the desired output [Aba05, SMMP09, STO13, ZCF+10, Apa16c, Apa16a]. The nature of this network of tasks allows to further classify distributed event stream processing systems into *clustered* and *networked* systems (see Figure 2.10).

In a clustered engine, scalability is pursued by sharing the effort of processing incoming event

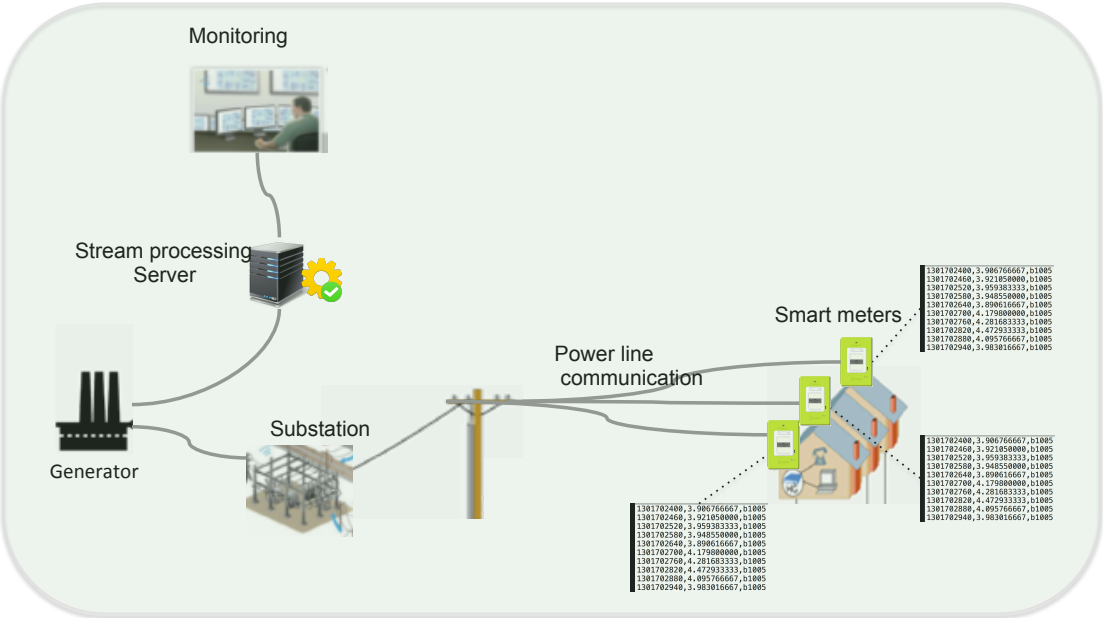

Figure 2.9 – A centralized stream processing model

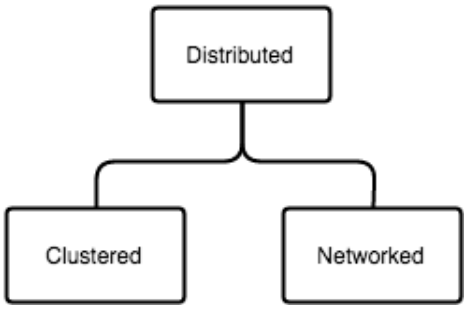

Figure 2.10 – Distributed deployment models

streams among a cluster of strongly connected machines, usually part of the same local area network. In a clustered event stream processing engine, the links connecting processing nodes among themselves perform much better than the links connecting sources and sinks with the cluster itself. Furthermore, the processing nodes are in the same administrative domain.
For example, Figure 2.11 presents the adoption of a clustered deployment architecture for the implementation of stream processing within the microgrid (see Section 2.1). The stream processing engine is deployed within a cluster at the control center. The cluster consists in a set of computing nodes that provide computing resources to the stream processing engine. Similarly to centralized architectures, a clustered architecture requires the event streams to be conveyed to the cluster location for processing, which is network consuming. This is undesirable when using low capacity networks such as power line communication in the microgrid.

Conversely, a networked event stream processing engine focuses on minimizing network

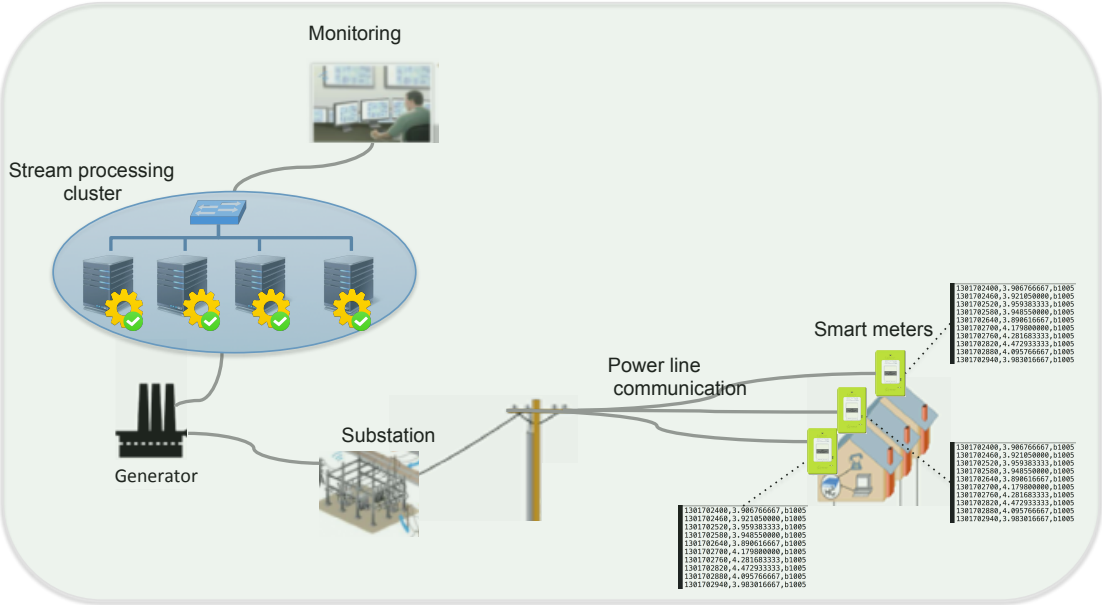

Figure 2.11 – A clustered stream processing model

usage by dispersing tasks over a wide area network, with the goal of processing information as close as possible to the sources. In such a model, each network node can participate in the stream processing. As a result, in a networked event stream processing engine, the links among processing nodes are similar to the links connecting sources and sinks to the engine itself. The processing nodes are widely distributed and potentially run in different administrative domains.

For example, Figure 2.12 presents the adoption of a networked deployment architecture for the implementation of stream processing within the microgrid (see Section 2.1). All the computing node deployed within the microgrid can participate in the stream processing. This includes smart meters at households, sensors over the electrical lines, data concentrators at substations, etc. Processing event streams as close as possible to the sources can reduces the network occupation. For example, executing the query **Q1** within the substation would be a better alternative[4] than executing it at the control center, as this would reduce the data flow (and thus, the network occupation) upstream to the substation.

In summary, seeking for better scalability, clustered and networked engines focus on different aspects: the former on increasing the available processing power by sharing the workload among a set of well connected machines, the latter on minimizing network usage by processing event streams as close as possible to the sources.

The runtime environment is the environment on which stream processing tasks are executed.

[4] from a network occupation point of view

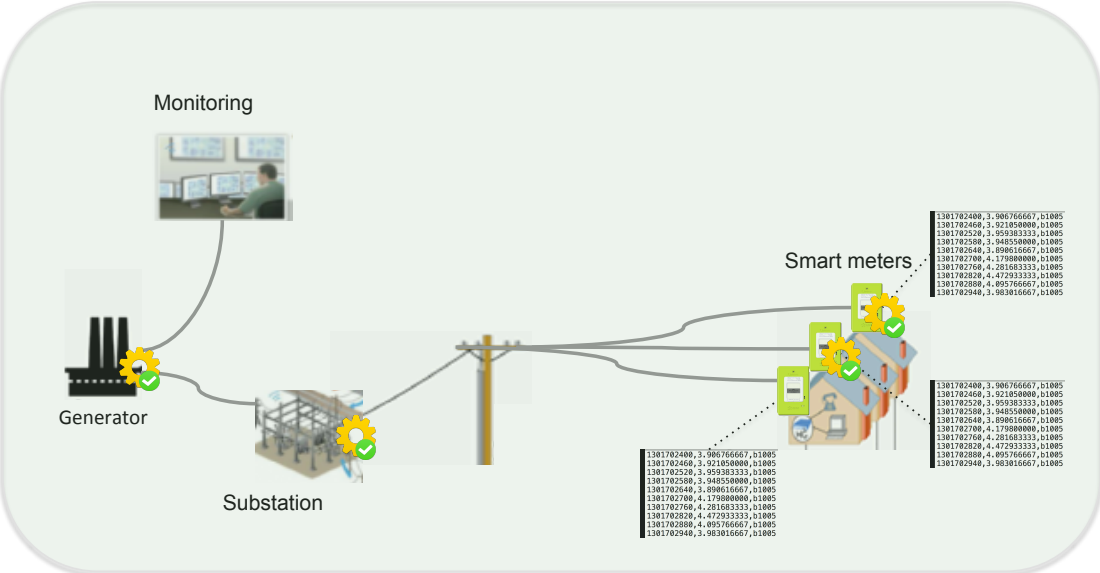

Figure 2.12 – A networked stream processing model

2.1.3 QoS requirements

Stream processing systems must often meet particular QoS goals (as opposed to systems working with batch style workloads), otherwise the quality of the output degrades or the output becomes worthless at all. A number of QoS dimensions have been adopted in existing event stream processing systems, including throughput, latency, memory, accuracy and high availability [ASB10].

Latency It is the amount of time or the average amount of time it takes for an event or a sequence of events to go through processing of a query and be notified to a consumer. The latency considers the processing time of the operators, the waiting time on buffers, and the network latency in case of distributed processing. For example in [LWK13], latency is addressed using a task chaining strategy, which consists in chaining lightweight operators together and execute them in a single thread or process. This eliminates the communication between the concerned operators, and thus decreases the latency.

Memory It is the maximum amount of memory used by the stream processing system. This can be defined for each operator or for the entire stream processing system. As stream processing systems are generally classified as *in-memory processing*, the memory footprint of stream processing processes is a crucial concern, in particular in resource constrained environments. For example, in order to reduce the memory requirements during the processing, [DGP+07], proposed a memory management scheme which allows objects to be shared as much as possible between operators, reducing the space and time overhead for such objects. *Storm*[STO13] implements a resource aware scheduler, which assigns operators to computing nodes considering their memory and CPU time requirements.

Throughput It is the number of events that are output per unit of time. High throughput generally goes with low latency. Therefore, strategies to achieve low latency stream processing lead to a better (higher) throughput. Conversely, strategies to achieve high throughput generally improve the latency. For example, the strategy used by [LWK13] to improve the throughput is to dynamically adjust the size of operator buffers in order to avoid long waits before event notification. This allows a fast event delivery, thus decreasing the latency.

Accuracy It is the accuracy of results in terms of error tolerance. Stream processing systems are often subject to data loss. Data loss can be triggered by the system itself, in the context of load shedding in case of bursty event stream arrival. Data loss can also be caused by the saturation of event buffers, or by the use of a non reliable transport mechanism like UDP for event dissemination. Such data loss can affect the accuracy of the stream processing results. For example in Aurora [CBB+03], the user indicates the importance of tuples as part of its QoS specifications. This information is used by the system to determine which tuples can be dropped, such that the accuracy of the results is not altered.

High availability It is the capacity of the stream processing system to be operational even in abnormal situations. In a centralized stream processing system for example, an abnormal situation can be the failure of the server. In distributed stream processing systems, this can be the failure of a processing node. For example, in Esper[Esp15], high availability is ensured by adding redundancy to the system so that failure of a server can be masked by an automatic switch to another server.

Some or all of the above can be specified with each continuous query. Given the QoS requirements for a continuous query or a set of continuous queries, it is the responsibility of the stream processing system to satisfy them or indicate that they cannot be satisfied with the available resources and the current load on the system.

It is important to stress that the QoS dimensions are not independent of each other. For example, increase in average event latency is likely to increase the total memory requirement in a stream processing system. As another example, decrease in memory usage due to dropping of events will reduce the accuracy of results. Because of this tradeoff among QoS dimensions, it is important to have a suite of techniques (e.g., scheduling algorithms, load shedding) for optimizing each metric individually and for balancing them as needed.

2.2 Existing projects and systems

This section presents a survey of some existing stream processing projects and systems. We first present some data (event) stream processing systems, which focus on applying continuous transformations on data streams. Then, we present some complex event processing systems, which focus on detecting complex event patterns within the event streams.

2.2.1 Event/Data stream processing systems

Stream

Stream [ABB+03] is a general-purpose data stream processing system. It supports a declarative, SQL-like query language called CQL [ABW06], which stands for continuous query language. *Stream* considers streams of structured data records (tuples) that conform to a specific schema. For example, if we denote by *MeterStream*, the event stream generated by smart meters in the example introduced in Section 2.1, then we can define its schema as: *MeterStream(meterID, energy, ts)*.
CQL provides a unified syntax to query both streams and stored relations. To do so, CQL distinguishes between three categories of operators: relation-to-relation operators, stream-to-relation operators, and relation-to-stream operators.
Relation-to-relation operators derive from SQL: they are the core of the language, which actually defines the processing. This has the advantage that a large part of the query definition is realized by using the standard notation of a widely used language. In order to add support for stream processing, CQL introduces the notion of windows, as a way to capture and store portion of each input stream inside relation tables for processing; for this reason, CQL denotes windows as stream-to-relation operators. Windows can be based both on time and on the number of contained elements. The last kind of operators that CQL provides is that of relation-to-stream operators, which define how processed tuples can become part of the output stream. Three relation-to-stream operators exist: *IStream* put a tuple into the output stream when it is added to the table, *DStream* considers only removed tuples, while *RStream* considers every tuple contained in the table. For example, using CQL, we can answer the query **Q1** (see Section 2.1) as follows:

```
select meterID, sum(energy) as t_energy
from MeterStream [Range 60 Minutes]
Where (meterID ='h1' or meterID = 'h2')
group by (meterID)
Having (sum(energy) > 10)
```

Notice the [Range 60 Minutes] CQL construct, which defines a 1 hour window on the *MeterStream* event stream. This results is a relation containing the set of events received during the last hour. The remaining of the query uses standard SQL constructs.
Starting from a CQL query, *Stream* computes a query plan. Query plans are composed of operators, which perform the actual processing, connected by queues, which buffer tuples as they move between operators, and synopses, which store operator state. Therefore, *Stream* follows the dataflow processing model.
Stream is a centralized stream processing system. The query parser, the scheduler, and the operators are colocated on a single server. The operator scheduling mechanism of *Stream* focuses on memory minimization under user-specified latency constraints[BBD+04]. In order to deal with resource overload problems on the *Stream* server, two major shedding techniques have been proposed: the first addresses the problem of limited computational resources by applying load shedding on a collection of sliding window aggregation queries [BDM04]; the

second addresses the problem of limited memory, by discarding operator state for a collection of windowed joins [SW04].

Stream is a centralized stream processing system. In consequence, all the event streams should be routed to the *Stream* server node before actually being processed. This is not desirable in resource constrained environments like smart grids as it increases the usage of network resources, which are limited. Moreover, the centralized architecture suffers scalability issues in presence of a large number of producers and consumers, and the *Stream* server is a single point of failure.

NiagaraCQ

NiagaraCQ [CDTW00, NDM+01] is a data stream processing system for Internet databases. The goal of the system is to provide a high-level abstraction to retrieve information, in the form of XML data sets, from a frequently changing environment like Internet sites. *NiagaraCQ* continuous query language is based on XML-QL [DFF+99], a declarative, SQL-like language for specifying XML queries. The command to create an continuous query has the following form:
CREATE *CQ_name*
XML-QL query
DO *action*
{**START** *start_time*} {**EVERY** *time_interval*} {**EXPIRE** *expiration_time*}

A query in *NiagaraCQ* combines an ordinary XML-QL query with additional time informations. Each query has an associated time interval that defines its period of validity: the query starts at *start_time* and ends at *expiration_time*. A query can be either timer-based or change-based; the former is evaluated periodically, after each *time_interval* during its validity interval, while processing of the latter[5] is driven by notifications of changes received from information sources. The defined action is performed upon the XML-QL query results.

As example, let us consider the query **Q1** defined in Section 2.1. This query cannot be expressed on *NiagaraCQ*, because XML-QL does not allow filters on aggregated values. Instead, we could partially answer query **Q1** by computing the sum of energy consumed by each house during the last hour. The filter, which checks for energy sum greater than 10 kWh should be implemented at the application level. The partial query is written as follows:

```
CREATE partialQ1
 Where <MeterEvent>
          <meterID>$id</meterID>
          <energy>$e</energy>
       </MeterEvent> IN eventstreams.xml
 Construct
       <result>
```

[5]The *time_interval* value is set to zero.

```
            <t_energy ID = meterID($id)> sum($e) </t_energy>
        </result>
DO notifyControlCenter()
START 0 EVERY 1h EXPIRE 0
```

This query assumes that the event streams from smart meters are appended to a file named *eventstreams.xml*. The *Where* clause of the XML-QL query matches each event in the *eventstreams.xml* file. The *Construct* clause contruct results from the matched events, which is the sum of the energy consumed for each group of events related to the same home (value of the *meterID* property). This query is executed every hour as specified by the *EVERY 1h* clause. The continuous query execution mechanism of *NiagaraCQ* ensures that only the delta file for the source file *eventstreams.xml* will be processed each time.

NiagaraCQ processes continous queries in a central server. It associates to each query a query plan, which is similar to a query execution plan in relational databases. To increase scalability, *NiagaraCQ* uses an efficient caching algorithm, which reduces access time to distributed sources and an incremental group optimization, which splits query plans into groups; members of the same group share the same query plan, thus increasing performance. Similarly to *Stream*, the main disadvantage of *NiagaraCQ* is its centralized architecture. In addition, *NiagaraCQ* does not provide support for QoS.

Aurora/Borealis

Aurora [ACc+03] is a general-purpose data stream processing system. It allows the continuous execution of queries over push-based input streams. Each input conforms to a particular schema, similarly to a row in a relational database table. In *Aurora*, a developer creates transforming rules with an imperative language called SQuAl, which stands for stream query algebra. SQuAl defines rules in a graphical way, by adopting the boxes and arrows paradigm, which makes connections between different operators explicit. The result is a dataflow graph which corresponds to the execution plan, which Aurora executes in a centralized way. For example, the dataflow graph associated to the query **Q1** (see Section 2.1) is depicted at Figure 2.13. For each house, it computes the sum of energy over one hour sliding windows of the smart meter event stream. Then, using a filter operator, it tests whether the energy sum is greater than 10. It merges the results of both houses into one stream that represents the final output stream of the query.

Interestingly, SQuAl allows users to associate a QoS specification to each output stream. This specification indicates how much latency the connected application can tolerate, the tolerance of the application regarding tuple drops, and the importance of the output tuples. In case adequate responsiveness cannot be assured due to overload situations, *Aurora* attempts to reduce the volume of input tuples via load shedding. The load shedding can be achieved by randomly dropping tuples or by choosing a semantic dropping strategy where it does some filtering before dropping tuples. The choice of the dropping technique depends on the user

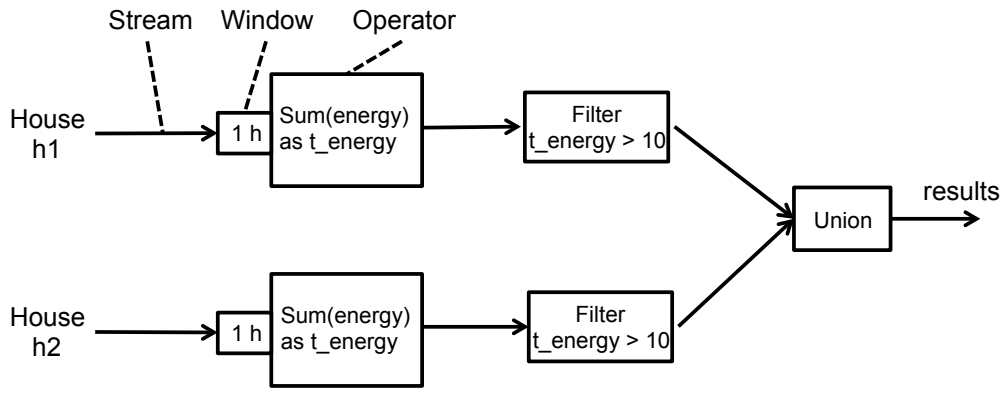

Figure 2.13 – Example of a dataflow graph in Aurora

QoS specification. For example, random drops are used to increase responsiveness, while semantic drops are used to avoid loosing important tuples.

Like *Stream* and *NiagaraCQ*, the centralized architecture of *Aurora* is inappropriate in large scale, resource constrained contexts like smart grids.

The *Aurora* project has been extended to investigate distributed processing both inside a single administrative domain and over multiple domains [CBB+03]. In both cases, the goal is that of efficiently distributing load between available resources. In these implementations, called *Aurora** (distribution within the same administrative domain) and *Medusa* (inter administrative domain distribution), communication between operators takes place using an overlay network, with dynamic bindings between operators and data streams. The two project have merged their features into the *Borealis* stream processor [Aba05].

Gigascope

Gigascope [CGJ+02, CJSS03, JM05] is a data stream processing system specifically designed for network applications, including traffic analysis, intrusion detection, performance monitoring, etc. The main concern of Gigascope is to provide high performance for the specific application field it has been designed for.
Gigascope defines a declarative, SQL-like language, called GSQL. All inputs to a GSQL are streams, and the output is a data stream. A data stream is bound to a schema, which defines the set of attributes exposed by its data items (tuples). The GSQL language includes only filters, joins, group by, and aggregates. Interestingly, it uses processing techniques that are very different from those of other data stream systems. In fact, to deal with the blocking nature of some of its operators, it does not introduce the concept of windows. Instead, it assumes that each tuple of a stream contains at least an *ordered attribute*, i.e. an attribute that monotonically increases or decreases as items are produced by the source of the stream. For example, a timestamp defined w.r.t. an absolute time, or a sequence number assigned at source. Users can specify which attributes are ordered, as part of the data definition, and this

information is used during processing.
For example, using GSQL, we can answer the query **Q1** (see Section 2.1) as follows:

```
SELECT meterID, sum(energy) as t_energy
FROM MeterStream
WHERE (meterID = 'h1' or meterID = 'h2')
GROUP BY (ts/3600, meterID)
HAVING (sum(energy) > 10)
```

The attribute ts (timestamp) is a 1-second granularity timer, so ts/3600 defines groups of 1 hour, in other words, one hour windows. When a group (window) with a new value of *ts* is produced, all of the pre-existing groups are closed, and therefore are aggregated and filtered. These mechanisms make the semantics of processing easier to understand, and more similar to that of traditional SQL queries. However, they can be applied only on a limited set of application domains, in which strong assumptions on the nature of data and on arrival order can be done. *Gigascope* translates an GSQL rule into basic operators, and composes them into a query plan, which conforms to the dataflow model. Query plans are executed in *Gigascope* in a centralized way, which is not appropriate in highly distributed, resource constrained contexts like smart grids. In addition, there is not a support for QoS in *Gigascope*.

Spark Streaming

Spark [ZCF+10] is a general framework for large-scale data processing. The main abstraction in *Spark* is that of a *resilient distributed dataset* (RDD) [ZCDD12], which represents a read-only collections of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Every data are represented as RDD, which represents the generic data type in Spark. To use Spark, developers write a driver program that implements the high-level control flow of their application and launches various operations in parallel. RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset.
Spark includes the *Spark Streaming* library, which is an extension of the core *Spark* API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. *Spark Streaming* provides a high-level abstraction called *discretized stream* or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs. Each RDD in a DStream contains data from a certain interval. Any operation applied on a DStream translates to operations on the underlying RDDs. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches. This approach has the benefit that it allows programmers to write streaming jobs the same way they write batch jobs.
For example, let us consider again the query *Q1* (see Section 2.1). Assume a DStream named *meterStream*, which contains smart meter events from a one hour interval. Then the query **Q1** can be answered by the following *Spark Streaming* job:

```
val ds1 = meterStream.map(e => (e.meterID, e))
val ds2 = ds1.reduceByKey((e1, e2) => e1.energy + e2.energy)
val res = ds2.filter((id, v) => { (id == "h1" || id == "h2") && v > 10})
res.print()
```

The spark execution engine is built around an acyclic dataflow model, and it runs on dedicated servers in a cluster. Therefore, Spark achieves a localized distribution of stream processing tasks over a set of homogeneous, resource-rich nodes connected with high-speed network links. Our focus is on achieving large scale distribution over a network of heterogeneous devices, some of them being potentially connected by low-speed network links. In consequence, *Spark Streaming* is not appropriate in large scale, resource constrained contexts like smart grids.

Storm

Storm [STO13] is a distributed framework for reliable and realtime stream processing. In *Storm*, streams are defined with a schema that names the fields in the stream's tuples. The logic for a realtime application is packaged into a *Storm* topology. A topology is a directed acyclic graph of producers called *spouts*, and operators called *bolts* that are connected with stream groupings. A stream grouping is a policy which defines how an output stream should be partitioned among the bolt tasks. For example, Figure 2.14 presents a topology which answers the query **Q1** (see Section 2.1).

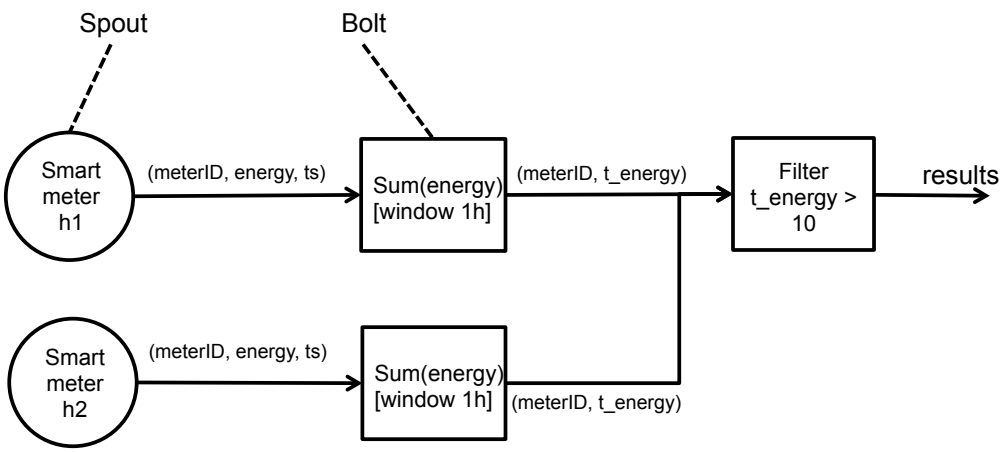

Figure 2.14 – Example of a *Storm* topology

Storm topologies run on a cluster. Clients submit topologies to a master node, which is called the *Nimbus*. *Nimbus* is responsible for distributing and coordinating the execution of the topology accross the worker nodes in the cluster. Each worker node runs one or more worker processes. At any point in time a single machine may have more than one worker processes, but each worker process is mapped to a single topology.
Storm offers several different levels of guaranteed message processing, including best effort (at most once), at least once, and exactly once. To do that, spouts keep the messages in

their output queues until they are being acknowledged. The acknowledgement happens after the successful processing of an event by the topology. If an acknowledgement comes for a message within a reasonable amount of time spouts clear the message from output queue. If an acknowledgement didn't come within a predefined period (30 second default) the spouts replay the message again through the topology. This mechanism guarantees that the messages are processed at least once inside *Storm*. In order to address the lack of resource awareness in *Storm*, [XCTS14, PCH+15] propose a task scheduling algorithm which increases overall throughput, by maximizing resource utilization (CPU, memory) while minimizing network latency.
Similarly to *Spark*, the clustered deployment model of *Storm* is inappropriate in large scale, resource constrained contexts like smart grids.

Samza

Samza [Apa16c] is a Java based, general purpose stream processing framework. A *Samza* data stream is an immutable unbounded collection of messages of same type. *Samza* data model is pluggable, the structure of message types being defined by users. A stream can have any number of consumers, and reading from a stream doesn't delete the message . *Samza* always persists streams in its brokering layer. An application in *Samza* is a logical collection of processing units that act on a message streams and produce output streams. The application creates a *Samza job*, which applies transformations on the input streams. It is possible to compose multiple jobs to create a dataflow graph where the nodes are streams containing data, and the edges are jobs performing transformations. This composition is done purely through the streams the jobs take as input and output. The jobs are otherwise totally decoupled: they need not be implemented in the same code base, and adding, removing, or restarting a downstream job will not impact an upstream job.
For example, Figure 2.15 presents a dataflow graph corresponding to the query **Q1** defined in Section 2.1. It contains two aggregate jobs which compute the sum of energy on one hour sliding windows of each input stream *h1* and *h2*. Their results are appended in the stream *h12*, which is processed by the filter job that detects events that report energy values that are greater than 10. Those events are appended to the final stream, which can be consumed by interested consumers.

Messages of a stream can optionally have an associated key which is used for partitioning. Streams are partitioned into sub streams and distributed across the processing tasks running in parallel.
A job is scaled by breaking it into multiple tasks. Each task consumes data from one partition for each of the job's input streams.
Samza supports a clustered deployment. It uses Apache Kafka [Apa16b] for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management. Message streams are handled by kafka, which provides a distributed publish/-subscribe system with persistence for message streams.

Similarly to *Spark*, the clustered deployment model of *Samza* is inappropriate in large scale,

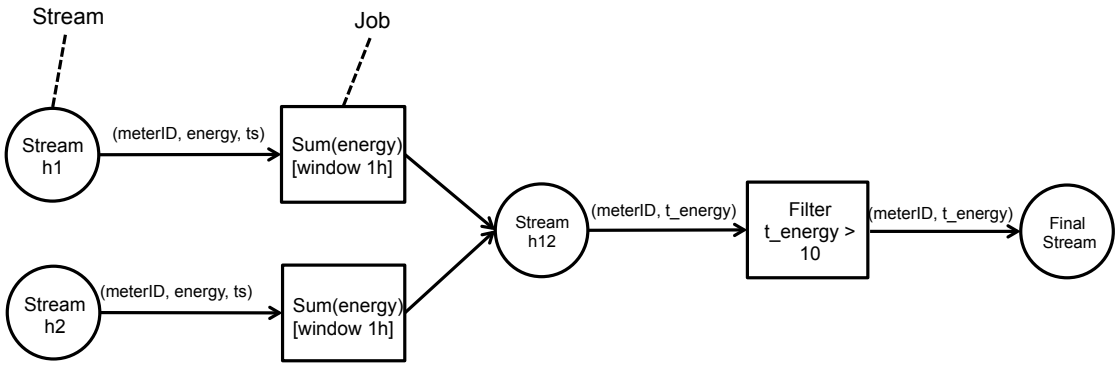

Figure 2.15 – Example of a dataflow graph in *Samza*.

resource constrained contexts like smart grids.

Flink

Flink [Apa16a] is an open source platform for distributed stream and batch data processing, based on the Stratosphere research project [ABE+14]. *Flink*'s core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. The *Flink* data model supports all primitive types, and makes it possible to define event types using Java or Scala classes.
The *Flink's DataStream* API makes it easy to write programs that implement transformations on data streams (e.g., filtering, updating state, defining windows, aggregating) coming from a variety of sources (sockets, files, Hadoop file system, Twitter, RabbitMQ, Kafka, ElasticSearch, etc...).
For example, using the *Flink's DataStream* API, the query **Q1** defined in Section 2.1 can be written as follows:

```
val evtStr : DataStream[MeterEvent] = getSmartMeterEventStreams()
val evtStr2 = evtStr.filter(e => e.meterID=="h1" || e.meterID == "h2")
val evtStr3 = evtStr2.map(e => (e.meterID, e.energy))
val evtStr4 = evtStr3.keyBy(0).timeWindow(Time.hours(1)).sum(1)
val res = evtStr4.filter((id, v)=> v > 10)
res.print
```

Flink also proposes *FlinkCEP*, a complex event processing library which is defined on top of the *Datastream* API. *FlinkCEP* allows to detect complex event patterns within an event stream. The pattern language allows only to specify sequences, type and content based filters, and windows. There is no support for logical operators like conjuction and disjunction.
Flink supports both centralized and clustered deployments. Its runtime consists of two types of processes: the master process also called *JobManager*, which coordinates the distributed execution (tasks scheduling and recovery, etc.), and the worker processes also called *TaskManagers*, which execute the tasks (or more specifically, the subtasks) of a dataflow. There is one

worker process on each node of the cluster, and each worker process can have many processing slots (threads). *Flink* executes a program in parallel by splitting it into subtasks and scheduling these subtasks to processing slots. The task scheduling algorithm allows subtasks from the same job to be executed within the same processing slot. The result is that one slot may hold an entire pipeline of the job. This reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency. The cluster deployment model of *Flink*, associated to the lack of support for event priority, make it inappropriate in large scale, resource constrained contexts like smart grids.

2.2.2 Complex event processing systems

Rapide

Rapide [Luc96, LV95] is considered as one of the first steps toward the definition of a complex event processing system. It consists of a set of languages and a simulator that allows users to define and execute models of system architectures.
Rapide enables users to capture the timing and causal relationships between events: in fact, the execution of a simulation produces a causal history, where relationships between events are made explicit.
Rapide models an architecture using a set of components, and the communication between components using events. Event types are defined as tuples whose data represent such communications. *Rapide* embeds a complex event detection system, which is used both to describe how the detection of a certain pattern of events (so called complex events) by a component brings to the generation of other events, and to specify properties of interest for the overall architecture. *Rapide* defines a pattern language which includes conjunctions, disjunctions, negations, sequences, and iterations, with timing constraints. There is no support for windowing and aggregations. Because of these limitations, the query **Q1** cannot be answered using *Rapide*. Instead, we can answer the following alternative query :
"*Notify me whenever the smart meter at home h1 or h2 reports an energy consumption higher than 2kWh.*"
This is written using the Rapide pattern language as follows:

```
(?E1, ?E2 : Integer)
MeterEvent(meterID is h1, energy is ?E1) where (?E1 > 2)
or
MeterEvent(meterID is h2, energy is ?E2) where (?E2 >2)
```

This pattern looks for events for which the value of the *meterID* attribute is *h1* and the value of the *energy* attribute is higher than 2, or events for which the value of the *meterID* attribute is *h2* and the value of the *energy* attribute is higher than 2.

The processing model of *Rapide* corresponds to the dataflow model. The system builds a DAG which represents the causality relation between event and/or event patterns. Each event pattern is detected using an event processing agent that implements its corresponding event

processing rules. The event processing agents are executed in a centralized way, and there is no support for QoS. Because of these limitations, *Rapide* is inappropriate in a large scale, resource constrained context like a smart grid or the IoT.

Raced

Raced [CM09] is a distributed complex event processing middleware. *Raced* extends the content-based publish-subscribe paradigm to provide a complex event detection service for large scale scenarios. The *Raced* event definition language allows the definition of complex event types using a few set of operators optimized for a distributed detection: message filters, composition operators, parameters and windows. A limitation of the *Raced* event definition language is that aggregations are not supported. In consequence, the query **Q1** (see Section 2.1) cannot be answered using *Raced*. Let us consider the following alternative query:
"*Notify me whenever there is an alert at home h1 and no energy consumption at home h2 within a day.*"
This is written using the *Raced* event definition language as follows:

```
AlertEvent[meterID="h1"] AND NOT MeterEvent[(meterID="h2") AND energy > 0]
WITHIN  1 day
```

This pattern captures the occurrence of a alert at home *h1* and the absence of events reporting a positive energy consumption at home *h2* within a day.
The processing model of *Raced* corresponds to the dataflow model. The detection of a complex event is recursively decomposed into the detection of its parts, each part being handled by a detecting node called broker. Brokers are organized in a shortest path tree which is used to recursively assign event detection tasks to brokers. Using this approach, *Raced* gets the benefits of distributed processing (load is split, and information is filtered near the sources), while limiting the transmission of unneeded information from node to node. Therefore, the *Raced* deployment model is distributed and networked. While such a deployment model is appropriate in a highly distributed context, the limitation of *Raced* resides on the fact that stream processing is realized on dedicated nodes which provides the computing resources. In addition, there is no support for QoS. Our aim is to realize event stream composition on multiple heterogeneous devices connected over the network, considering QoS goals such as memory, CPU, latency and event priority.

Cayuga

Cayuga [DGP+07] is a general purpose event monitoring system. It is based on a language called *CEL* (Cayuga Event Language). Its structure strongly resembles that of traditional declarative languages for databases. Each *CEL* query has the following simple form:
SELECT < *attributes* >
FROM < *stream expession* >
PUBLISH < *output_stream* >

A query consists in a *SELECT* clause that filters input stream, a *FROM* clause that specifies a streaming expression, and a *PUBLISH* clause that produces the output. It includes typical SQL operators and constructs, like selection, projection, renaming, union and aggregates. The streaming expression contained in the *FROM* clause enables users to specify detection patterns, including sequences as well as iterations. Notice that *CEL* does not introduce any windowing operator. Therefore the query **Q1** (see Section 2.1) cannot be answered using *CEL*. Instead, let us consider the following query.
"*Notify me whenever the smart meter at home h1 or h2 reports a energy consumption greater than 1kWh.*"
This is written using *CEL* as follows:

```
SELECT meterID, energy
FROM
FILTER {(meterID='h1' OR meterID ='h2') AND energy > 1} (MeterEvent)
PUBLISH Alert
```

The semantics of all operators is formally defined using a query algebra [DGH+06]. *Cayuga* translates each rule into non deterministic automata for event evaluation. *Cayuga* does not allow distributed processing, the automata associated to different rules being strictly connected with each other. Furthermore, there is no support for QoS. Because of these limitations, *Cayuga* is not appropriate in large scale and constrained environments like smart grids.

NextCEP

NextCEP [SMMP09] is a distributed complex event processing system. Similarly to Cayuga, it uses a language that includes traditional SQL operators, like filtering and renaming, together with pattern detection operators, including sequences and iterations. There is no support for windows and aggregations. Therefore, it is not possible to answer query **Q1** (see Section 2.1) using *NextCEP*. Instead, let us consider the following query:
"*Notify me whenever the smart meter at home h1 reports an increase of the energy consumption which is greater than 2kWh.*"
This is answered using the *NextCEP* language as follows:

```
SELECT * FROM MeterEvent E1; MeterEvent E2
WHERE FILTER(E1.meterID = E2.meterID),
E2.energy > E1.nergy + 2
```

The complex event detection is performed by translating rules into non deterministic automata, that strongly resemble those defined in Cayuga in structure and semantics. In *NextCEP*, however, detection can be performed in a distributed way, by a set of strictly connected nodes in a cluster. The main focus of the *NextCEP* project is on rule optimization. In particular, the authors provide a cost model for operators that defines the output rate of each operator according to the rate of its input data. NextCEP exploits this model for query rewriting, a process

that changes the order in which operators are evaluated without changing the results of rules. The objective of query rewriting is that of obtaining the best possible evaluation plan, i.e. the one that minimizes the usage of CPU resources and the processing delay. However, the cost model does not captures the cost of sending events from event sources to processing nodes, as the authors assume the network not to be a bottleneck. Such an assumption, along with the clustered deployment restriction, are not acceptable in a large scale, resource constrained environment like a smart grid or the IoT.

SASE / SASE+

SASE [WDR06, GWC+06] is a monitoring system designed to perform complex event processing over real-time streams of RFID events. An event type is defined as a set of attributes. The *Sase* language is based on patterns. Its overall structure is:

[**FROM** <*stream name*>]
EVENT <*event pattern*>
[**WHERE** <*qualification*>]
[**WITHIN** <*window*>]
[**RETURN** <*return event pattern*>]

The FROM clause provides the name of an input stream. If it is omitted, the query refers to a default system input. The EVENT, WHERE and WITHIN clauses form the event matching block. The EVENT clause specifies an event pattern to be matched against the input stream. Patterns are expressed using logic operators and sequences. The WHERE clause, if present, imposes value-based constraints on the events addressed by the pattern. The WITHIN clause further specifies a sliding window over the event pattern. The event matching block transforms a stream of input events to a stream of new composite events. Finally, the RETURN clause transforms the stream of complex events for final output. The *SASE* language allows only the detection of given patterns of RFID events; it does not include any notion of aggregation. Therefore, the query **Q1** (see Section 2.1) cannot be answered using the SASE language. Let us consider the following query.
"*Notify me whenever the smart meter at home h1 reports an increase of the energy consumption which is greater than 2kWh within 1 hour.*"
This is answered using the *SASE* language as follows:

```
FROM    MeterEventStream
EVENT   SEQ(MeterEvent E1, MeterEvent E2)
WHERE   [meterID='h1']
   AND  (E2.energy > E1.energy +2)
WITHIN   1 hour
```

SASE+ [ADGI08, GADI08] extends the expressiveness of *SASE* by including iterations and aggregates in the pattern expressions. For example, using *SASE+*, it is possible to compute the total energy consumed by house *h1* during the last hour as follows:

```
FROM    MeterEventStream
EVENT   SEQ(MeterEvent+ e[ ])
WHERE   e[i].meterID='h1'
WITHIN  1 hour
RETURN  sum(e[ ].energy)
```

SASE and *SASE+* are implemented using a query plan-based approach, that is, a dataflow paradigm with pipelined operators as in relational query processing. A query plan is composed of six blocks, which sequentially process incoming information elements realizing a of pipeline: the first two blocks detect the events matching the pattern of the *event* clause by using finite state automata. Successive blocks check selections constraints, windows and build the desired output.
SASE and *SASE+* execute query plans in a centralized way. In addition, there is no QoS support in *SASE* and *SASE+*. Because of these limitations, *SASE* and *SASE+* are not appropriate in large scale, resource constrained contexts like smart grids or the IoT.

StreamBase

StreamBase [Str15] is a software platform that includes a data stream processing system, a set of adapters to gather information from heterogeneous sources, and a developer tool based on Eclipse. *StreamBase* has been built to commercialize the *Aurora* and *Borealis* systems. it uses a declarative, SQL-like language for rule specification, called *StreamSQL*. Beside traditional SQL operators, *StreamSQL* offers customizable windows operators. In addition, it includes a simple pattern language that captures conjunctions, disjunctions, negations, and sequences of events. For example, the query **Q1** defined in Section 2.1 is written in *StreamSQL* as follows:

```
SELECT meterID, sum(energy) AS t_energy
FROM MeterStream [SIZE TIME 3600 ADVANCE TIME 3600]
WHERE (meterID = 'h1' OR meterID ='h2')
GROUP BY (meterID)
HAVING (sum(energy) > 10)
```

Operators defined in StreamSQL can be combined using a graphical plan-based rule specification language, called *EventFlow*, which conforms to the dataflow model. User-defined functions, written in Java, C++ or Python, can be easily added as custom aggregates. StreamBase supports both centralized and clustered deployments, providing high availability in case of failures. Here again, the centralized and clustered deployment models are not suited for large scale, resource constrained contexts. In *StreamBase*, users can specify the maximum load for each used server, but the documentation does not specify how the load is actually distributed to meet these constraints. Moreover, QoS dimensions like event priority and latency are not considered. These QoS dimensions are required in our context, as we will see later in the following chapters.

Esper

Esper [Esp15] is a open-source software for complex event processing and event stream analysis. *Esper* is integrated into Java and .Net (*NEsper*) as libraries. Users specify event types using Java classes, Maps or XML. *Esper* defines a rich declarative, SQL-like language for rule specification, called EPL (Event Processing Language). EPL includes all the operators of SQL, adding custom constructs for windows definition, and output generation.

For example, the query **Q1** (see Section 2.1) can be answered using EPL as follows:

```
SELECT meterID, sum(energy) AS t_energy
FROM MeterStream(meterID in ('h1', 'h2')).win:time_batch(1 hour)
WHERE (meterID = 'h1' OR meterID ='h2')
GROUP BY (meterID)
HAVING (sum(energy) > 10)
```

EPL embeds two different ways to express event patterns: the first one exploits so called EPL patterns, that are defined as nested constraints including conjunctions, disjunctions, negations, sequences, and iterations. The second one uses flat regular expressions on a single event type.
The EPL pattern engine is a dynamic state machine in which states can have sub-states. The term "delta networks", a network of objects in which only changes to data are communicated across object boundaries and only when required, is at the foundation of the engine design. The pattern matching functionality is built using nondeterministic finite automata.
Esper supports a centralized deployment. *EsperHA* (Esper High Availability) is a commercialized version of *Esper*, which supports a clustered deployment. *EsperHA* combines *Esper* with high performance resilience options, ensuring state recoverabilty upon a failure or orderly shutdown.
The lack of QoS support, coupled with the centralized and clustered (for *EsperHA*) deployment model, make *Esper* not appropriate in large scale, resource constrained contexts like smart grids.

2.3 Discussion

Table 2.1 summarizes the studied stream processing systems.

As we can see, the dataflow model has been widely adopted as the processing model in data stream processing engines. This is justified by its flexibility as it allows to implement a large category of stream processing applications. In addition, the dataflow model is suitable for distributed stream processing, because it clearly separates the event processing implementation from the event communication between processing components.
Early stream processing engines like *STREAM*, *NiagaraCQ*, *Aurora* and *Gigascope* were centralized systems that run the stream processing dataflows within a single node. Those systems are not appropriate for large scale scenarios characterized by a large number of stream producers,

	Name	Processing model	Deployment model	QoS Goals
DSP	**Stream**	Dataflow	Centralized	memory, CPU
	NiagaraCQ	Dataflow	Centralized	response time
	Aurora	Dataflow	Centralized	latency, CPU, accuracy
	Borealis	Dataflow	Clustered	latency, CPU, accuracy, fault tolerance
	Gigascope	Dataflow	Centralized	-
	Spark Streaming	Dataflow	Clustered	latency, fault tolerance
	Storm	Dataflow	Clustered	latency, fault tolerance, throughput, memory, CPU
	Samza	Dataflow	Clustered	-
	Flink	Dataflow	Clustered	fault tolerance, latency, memory, throughput
CEP	**Rapide**	Dataflow	Centralized	-
	Raced	Dataflow	Networked	-
	Cayuga	FSA	Centralized	memory
	NextCEP	FSA	Clustered	CPU
	Sase / Sase+	Dataflow, FSA	Centralized	-
	StreamBase	Dataflow	Clustered	high availability
	Esper	Dataflow, FSA	Centralized	-
	NETAH	Dataflow	Networked	memory, CPU, network, event priority

Table 2.1 – Comparison of existing systems

potentially distributed.
Recent systems such as *Spark*, *Storm*, *Samza*, and *Flink* addressed distributed event stream processing within a cluster. Clusters generally provide certain levels of guarantee on the computing resources availables and the network latency. Since the design of thoses system relies on such guarantees, they are not suited for large scale distributed contexts like smart grids or the internet of things, which consist in a large variety of processing nodes having different computing resources, and which are connected by different network technologies with different characteristics (latency, bandwidth, etc). A few number of systems (e.g, *Borealis*, *Raced*) support the networked deployment model, which is needed in such contexts. Despite their deployment model which is adapted in our context, those systems (Borealis and Raced) have been designed under the assumption that the network bandwidth and the computing resources on all the processing nodes are enough. Such assumptions cannot be made in heterogenous contexts like smart grids, which consists in a large set of heterogeneous and resource limited computing nodes connected by high latency networks. Therefore, the QoS goals we are interested in are memory, CPU, latency and also event priority, the latter being introduced by smart grid applications requirements.

2.4 Conclusion

In this chapter, we presented existing works related to the work addressed in this thesis. We identified two systems categories which characterize the stream processing domain: data stream processing systems and complex event processing systems. We focused our study on the processing model and the deployment model adopted in each solution. We observed that because of its simplicity and flexibility, the dataflow processing model has been widely adopted for implementing stream processing engines.
The early stream processing systems like *Stream*, *Aurora*, *Cayuga*, etc., are centralized and therefore, they present scalability and availability issues in presence of a high number of stream sources. The new stream processing systems addressed these issues by implementing

stream processing on distributed architectures. Systems like *NextCEP*, *StreamBase*, *Spark Streaming*, *Storm*, *Flink*, etc., implemented stream processing on cluster architectures, taking advantages of the large computing and network resources available. With the growth of application domains such as the internet of things and smart grids, the demand for large scale distributed stream processing increases. The characteristics of such runtime environments, manifested by the heterogeneity of processing devices and network connections, requires to address QoS needs regarding memory, CPU, latency and priority.
Therefore, it is important to have stream processing models and systems which allow to implement stream processing applications on such contexts, considering the associated QoS. In the next chapter, we will present the event stream composition model that is used by the our NETAH framework.

3 The NETAH Event Stream Composition Model

This chapter presents our model for representing an event stream composition. Section 3.1 introduces some formalisms that we use all along this chapter. Section 3.2 introduces the concept of event, which is the basic entity manipulated in an event stream composition system, and the concept of event type which allows to categorize event instances. Section 3.3 presents the concept of event stream, which models unbounded sequence of event instances, and stream based composition operators. Section 3.4 describes the concept of event stream composition expression that models an event stream composition. Section 3.5 presents a model to represent the QoS associated to event stream composition. Finally, Section 3.6 concludes this chapter.

Contents

3.1 Preamble 36
3.2 Event, event type, event streams 37
3.3 Event stream composition operators 41
3.4 Event stream composition 49
3.5 QoS requirements 52
3.6 Conclusion 54

3.1 Preamble

This section introduces some formalisms to represent complex value types that are useful for introducing the concepts presented in this chapter. The proposed formalization is an adaptation of the one proposed in [CV11].

Let's assume a finite set of *domains*, each consisting of a possibly infinite set of values. In particular, we consider the domain $\mathbb{S}$ of strings, $\mathbb{B}$ of booleans ($\{true, false\}$), $\mathbb{Z}$ of integers, and $\mathbb{R}$ of real numbers. We also consider a domain $\mathbb{T}$ defined by the set $\mathbb{N} \bigcup \{0\}$, i.e. the set of natural numbers plus zero, which characterizes time values. These can be represented alternatively as *string, boolean, integer, real* and *time* respectively. In addition, we assume a set $\mathbb{A} = \{A_1,\ A_2,\ ...\} \subseteq \mathbb{S}$ of type names.

Complex value types

Complex value types are represented by lower-case letters with hats (e.g. $\hat{t}$) and are defined by a pair $A : def$, where A is the name of the type and def its definition. In order to enable access to both components we assume the functions *name* and *def*, which given a type will return the respective component of the type. For instance, for the type *Power : real, name(Power : real) = Power* whereas *def(Power : real) = real*. The set of all complex value types $\mathscr{T}$ is defined inductively as follows.

1. if D is a domain, then $A : D$ is an atomic type named A, where $A \in \mathbb{A}$;
2. if $\hat{t}$ is a type, then $A : \{\hat{t}\}$ is a type set named A;
3. if $\hat{t}_1, ..., \hat{t}_n$ are types with distinct names, then $A : \langle \hat{t}_1, ..., \hat{t}_n \rangle$ is a tuple type named A, and each $\hat{t}_i$ is an attribute type;
4. if $\hat{t}_1$ and $\hat{t}_2$ are types with distinct names, then $A : \hat{t}_1 \oplus \hat{t}_2$ is the alternative type.

Every type $\hat{t} \in \mathscr{T}$ denotes a set of complex value instances $[\![\hat{t}]\!]$ which is defined inductively as follows.

1. For each atomic type $A : D$, $[\![A : D]\!] = \{A : d \mid d \in D\}$, where we assume the values of the domain D given;
2. for set types of the form $A : \{\hat{t}\}$, $[\![A : \{\hat{t}\}]\!] = \{A : S \mid S \in \mathscr{P}([\![\hat{t}]\!])\}$, where $\mathscr{P}$ denotes the power set;
3. for tuple types of the form $A : \langle \hat{t}_1, ..., \hat{t}_n \rangle$, $[\![A : \langle \hat{t}_1, ..., \hat{t}_n \rangle]\!] = \{A : \langle A_1 : v_1, ..., A_n : v_n \rangle \mid A_i = name(\hat{t}_i) \bigwedge v_i \in [\![\hat{t}_i]\!] \bigwedge i \in [1..n]\}$;
4. for alternative type of the form $A : \hat{t}_1 \oplus \hat{t}_2$, $[\![A : \hat{t}_1 \oplus \hat{t}_2]\!] = \{A : v \mid A : v \in [\![\hat{t}_1]\!] \bigvee A : v \in [\![\hat{t}_2]\!]\}$

Access to components of complex value instances

We write $t \equiv \hat{t}$, to denote that t is an instance of a complex value type $\hat{t}$. We define the function val to obtain the value v associated to an instance t. In addition, for tuple values t of the type $A : \langle \hat{t}_1, ..., \hat{t}_n \rangle$ such that $name(\hat{t}_i) = A_i$, we adopt the dot notation $t.A_i$ to access the instance of the attribute type $\hat{t}_i$ of t. In particular, if we have $t = \langle A_1 : v_1, ..., A_n : v_n \rangle$, then $\forall i \in [1..n],\ v_i = val(t.A_i)$.

Example 3.1. Let's consider the tuple type $Account : \langle id : string,\ balance : real \rangle$. Then, $t = \langle id : B14,\ balance : 500 \rangle$ is an instance of the type named $Account$. In addition, we have $val(t.id) = B14$ and $val(t.balance) = 500$.

For simplicity, we also adopt the notation t, to denote the value $val(t)$ of an instance t. The difference between the value of the instance and the instance itself depends on the context. In Example 3.1, we have $t.id = B14$ and $t.balance = 500$.

3.2 Event, event type, event streams

3.2.1 Event

The literature proposes different definitions of an event. For example, in [MsSS97] an event is a happening of interest, which occurs instantaneously at a specific time. Another definition given by [RW97] characterizes an event as the instantaneous effect of the termination of an invocation of an operation on an object. According to the first definition, an event exists because some entity is interested in it; the second one defines events independently of any interested party. The second definition also subsumes an object model while the first one is neutral with respect to the model adopted for entities.

We adopt the following definition of event from [EN10].

Definition 3.1. (Event) An *event* is something that happened in a particular system or domain, and that is particularly significant, interesting or unusual. The word *event* is also used to mean a programming entity that represents such an occurrence in a computing system.

In computing systems the notion of event has a major importance since it provides a powerful abstraction to model dynamic aspects of applications. For instance, events can represent state changes in databases; signals in message systems; changes of existing objects or the creation of new objects in object-oriented systems; or "real-world" events such as the departure or arrival of vehicles, an alarm raised by a smart meter, or a smart meter measure. In event-driven programming an event is a message that indicates a situation that happened, such as a keystroke or a mouse click. In process control an event is an occurrence that happened and that has been registered. Examples are a purchase order, an email confirmation of an airline reservation, a stock tick message that reports a stock trade.

3.2.2 Event type

Applications that manage and process events sometimes deal with event objects having a similar structure and a similar meaning. Consider for example events coming from a current sensor. Those events contain the same kind of information, though of course each event will be associated to a different point in time, and will report a different current value. The *event type* concept allows to specify the structure of this entire class of events. This is similar to defining a reusable type in a programming language.

Definition 3.2. (Event type) An *event type* is an expression that characterizes a class of significant facts (events) and the context under which they occur. Facts of the same nature are denoted by events that have the same type. In the rest of this document we write *EventType* to denote the set of all event types.

Event type representation

According to the complexity of the event model, the event types are represented as sequences of strings [YBMM94], regular expressions [Bai94] or as expressions of an event algebra [CM94, GD94, CC96]. Other models represent an event type as a collection of parameters or attributes, allowing the type itself to contains implicitly the content of the message. This model is useful when we need to reason about events content. For example, $MeterAlarm : \langle idMeter : string, voltage : real, current : real \rangle$ is an event type that represents a smart meter reading, where the current and voltage values observed are represented by attributes named *voltage* and *current*.
Event composition, or specifically event streams composition supports the idea of performing operations on events. Some of those operations need to access event contents, as it will be presented in Section 3.3. Therefore, we represent an event type named E as a tuple type $E : \langle A_1 : D_1, ..., A_n : D_n \rangle$.
The event type attributes carry the information associated to the event type. We distinguish between two kinds of attributes, namely *meta-attributes* and *payload attributes*.

Meta-attributes carry meta-information about the event. They are common to all event types. Table 3.1 presents the meta-attributes associated to event types:

- the *producerID* attribute refers to the identifier of the entity that produced the event occurrence;
- the *detectionTime* attribute refers to the time at which the event occurrence has been detected by a source;
- The *productionTime* attribute refers to the time at which the event has been produced, in case the event is the result of processing other events;
- the *notificationTime* attribute refers to the time at which the event is notified to interested component;

- the *receptionTime* attribute refers to the time at which an interested consumer receives the event.

It is worth to mention that meta-attributes can be extended according to applications and domains.

Name	Domain
producerID	string
detectionTime	time
productionTime	time
notificationTime	time
receptionTime	time

Table 3.1 – Event type meta-attributes

On the other hand, payload attributes carry specific information about the event itself.

Example 3.2. (Event type) The event type named $MeterReading$ described in Figure 3.1) represents a smart meter reading. Its payload attributes are $voltage : real$ and $power : real$ which are the voltage and power values associated to each reading.

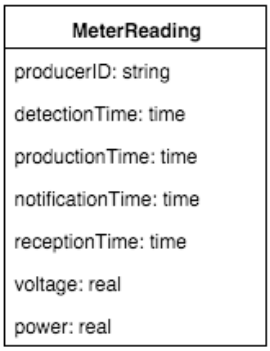

Figure 3.1 – Example of an event type

For writing simplicity, we will sometimes omit the meta-attributes in the description of event types and occurrences, when they are not relevant for the topic under consideration.

Example 3.3. (Omitting meta-attributes) For example, the $MeterReading$ event type previously defined can be simply defined as $MeterReading : \langle voltage : real,\ power : real \rangle$.

Event occurrence

An event occurrence (or event object, or simply event) e is an instance of an event type E, that is $e \equiv E$. The event occurrence e specifies the value of each attribute of E.

Example 3.4. (Event occurence) Let's consider the event type $MeterReading : \langle voltage : real,\ power : real \rangle$. Figure 3.2 gives an instance of this type produced by the source identified as *meter5* at time 1, notified at time 2, received at time 4, for which the voltage and current values are 9 and 1 respectively.

e ≡ MeterReading
producerID: "meter5"
detectionTime: 1
productionTime: 1
notificationTime: 2
receptionTime: 4
voltage: 9
power: 1

Figure 3.2 – Example of an event instance

Defining event priority

In many applications such as smart grids, there is a notion of *priority* associated to events. The goal is to allow events having higher priority to be processed and notified prior to events having lower priorities. In order to allow the definition of event priorities, we introduce an attribute $priority : integer$ in the set of meta-attributes of event types (See Table 3.2). The priority value of an event determines its level of prioriry: the higher is its priority value, the lower is its priority level.

Name	Domain
producerID	string
detectionTime	time
productionTime	time
notificationTime	time
receptionTime	time
priority	integer

Table 3.2 – Meta-attributes of prioritized event types

Example 3.5. (Prioritized events) The event instances e_1 and e_2 in Figure 3.3 represents two event instances of the type $MeterReading$, e_1 being more priority than e_2, since $e_1.priority < e_2.priority$.

$e_1 \equiv$ MeterReading	$e_2 \equiv$ MeterReading
producerID: "meter5"	producerID: "meter5"
detectionTime: 1	detectionTime: 3
productionTime: 1	productionTime: 3
notificationTime: 2	notificationTime: 4
receptionTime: 4	receptionTime: 6
priority: 2	priority: 3
voltage: 9	voltage: 12
power: 1	power: 1.2

Figure 3.3 – Example of prioritized event instances

3.2.3 Event stream

Definition 3.3. *An event stream is a continuous, append-only sequence of events* $\{e_1, ..., e_n, ...\}$.

In this thesis, we consider a category of event streams in which events have the same type, and we differentiate between event streams generated by a specific source.

Definition 3.4. (Typed stream) Let E be an event type. The typed stream induced by E, or simply the event stream of type E is denoted $Stream(E)$. We have:
$Stream(E) = \{e_1, e_2, ..., e_n, ... \mid \forall i, e_i \equiv E\}$.
Therefore, $type(Stream(E)) = \{E\}$. This means that typed streams are represented as typed sets.

Definition 3.5. (Bounded event stream) A bounded event stream is a typed stream generated by a specific source. The event stream of type E generated by the source s is noted $Stream(E, s)$. That is, $Stream(E, s) = \{e_1, ..., e_n \mid \forall e_i, e_i \equiv E \wedge e_i.producerID = s\}$.
$Stream(E, s)$ denotes the event stream of type E, "bounded" to the source s.

Remark: If S is a set of sources, then we have $Stream(T) = \{\bigcup Stream(T, s), s \in S\}$.

3.3 Event stream composition operators

Event stream composition operators specify the operations that can be performed on event streams. An event stream composition operator op takes one or many input event streams of a given type, and produces an output stream of a given type (see Figure 3.4). In the following, we adopt the notation ES_i to denote the stream of events of type E_i, that is $ES_i = Stream(E_i)$.

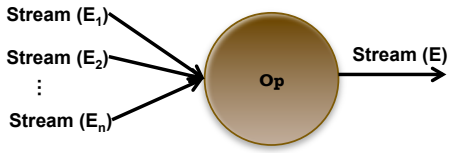

Figure 3.4 – Operator synopsis

3.3.1 Filter

The *filter* operator selects events in an input stream that satisfy a given predicate.

Definition 3.6. (Filter) Let ES_i be an event stream, and P be a predicate. Then, $filter_P(ES_i)$ denotes the event stream ES_i filtered according to the predicate P. We have:
$filter_P(ES_i) = \{e_j \mid e_j \in ES_i \wedge P(e_j)\ is\ true\}$.
The predicate P is defined according to the following rules:

- $A_i\ \theta\ v_i$ is a predicate, where

  - A_i is the name of an attribute of E_i, that is $\exists \hat{t}_j$ such that $\hat{t}_j$ is an attribute type of E_i and $name(\hat{t}_j) = A_i$

  - $\theta \in \{<, >, =, \leq, \geq\}$ is a comparison operator,

  - $v_i \in [\![\hat{t}_j]\!]$ is a value.

- if P_1 and P_2 are predicates, then

  - $P_1 \wedge P_2$ is a predicate. The symbol $\wedge$ denotes the conjunction.

  - $P_1 \vee P_2$ is a predicate. The symbol $\vee$ denotes the disjunction.

The output of the filter operator is an event stream having the same type as E_i.

Example 3.6. Let's consider the event type $E_i = MeterMeasure$: $\langle$ *meterID : string, realPower : double*$\rangle$[1] and some event instances of type E_i, $e_1 = \langle$"$meter1$",7$\rangle$, $e_2 = \langle$"$meter1$",5$\rangle$, $e_3 = \langle$"$meter1$",4$\rangle$, $e_4 = \langle$"$meter1$",6$\rangle$.
Let's also consider the event stream ES_i =*Stream(MeterMeasure)* = $\{e_1, e_2, e_3, e_4, ...\}$.
Then, $filter_{realPower>5}(ES_i) = \{e_1, e_4, ...\}$. Events e_2 and e_3 have been filtered out by the filter operator since they don't satisfy the predicate "realPower > 5".

3.3.2 Disjunction

The disjunction operator merges the input streams into one output stream.

Definition 3.7. (Disjunction) Let $ES_1, ES_2, \ldots, ES_n$ be n event streams. Then, $OR(ES_1, \ldots, ES_n)$ denotes the *disjunction* (merge) of event streams $ES_1, ES_2, \ldots, ES_n$. We have:

[1] We omitted the meta-attributes

$OR(ES_1,...,ES_n) = \{e_j \mid \exists i \in [1..n],\ e_j \in ES_i\} = \bigcup ES_i,\ i \in [1..n]$.
The disjunction operator produces events in the output stream at the occurrence of events in any of the input streams $ES_i,\ i \in [1..n]$.

Example 3.7. Let's consider the event streams ES_1 and ES_2 in Figure 3.5. In this example, the output of $OR(ES_1, ES_2)$ is the sequence $\{e_{1,1}, e_{2,1}, e_{2,2}, e_{1,2}, ...\}$.

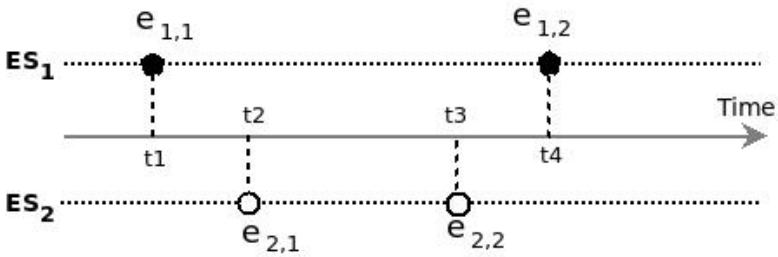

Figure 3.5 – Example situation 1. The time associated to events represents the time at which the events are processed by the operator.

3.3.3 Conjunction

The conjunction operator is used to ensure that at least one event occured in all input streams.

Definition 3.8. (Conjunction) Let $ES_1, \ldots, ES_n$ be n event streams. Then, $AND(ES_1, \ldots, ES_n)$ denotes the *conjunction* of event streams $ES_1, ES_2, \ldots, ES_n$. We have:
$e \in AND(ES_1,...,ES_2)$ iff $\forall i \in [1..n],\ \exists e_{i,j} \mid e_{i,j} \in ES_i$.
The output of the disjunction operator is an event stream of type $A : \langle context : \{E_1 \oplus ... \oplus E_n\}$[2].
The attribute named $context$ contains all the event $e_{i,j}$ that occur: $e.context = \{e_{i,j} \mid \forall i, e_{i,j} \in ES_i\}$.
In other words, an event e is produced in the output stream if events $e_1, e_2, \ldots, e_n$ occur respectively in input streams $ES_1, ES_2, \ldots, ES_n$, regardless their occurrence order. The attribute named *context* of the event e contains all events $e_1, e_2, \ldots, e_n$.

Example 3.8. By considering again the event streams ES_1 and ES_2 in Figure 3.5, the output of $AND(ES_1, ES_2)$ will be:

- An event e_1 at time t2 with the event parameters $e_{1,1}$ and $e_{2,1}$, such that $e_1.context = \{e_{1,1},\ e_{2,1}\}$
- An event e_2 at time t4 with the event parameters $e_{2,2}$ and $e_{1,2}$, such that $e_2.context = \{e_{2,2},\ e_{1,2}\}$.

3.3.4 Sequence

The sequence operator captures precedence order of events in input streams.

[2] We omitted the meta-attributes

Definition 3.9. (Sequence) Let $ES_1, \ldots, ES_n$ be n event streams. Then, $SEQ(ES_1, \ldots, ES_n)$ denotes a sequence between events in event streams $ES_1, \ldots, ES_n$. We have:
$e \in SEQ(ES_1, ..., ES_2)$ iff $\exists\, e_1 \in ES_1, ..., e_n \in ES_n \mid \forall i \in [1..n-1],\ e_i.receptionTime < e_{i+1}.receptionTime$.
The output of the sequence operator is an event stream of type $A : \langle context : \{E_1 \oplus E_2 \oplus ... \oplus E_n\}\rangle$[3].
The attribute named $context$ contains all the event $e_{i,j}$ that occur: $e.context = \{e_{i,j} \mid \forall i, e_{i,j} \in ES_i\}$.
In other words, the sequence operator produces an event e in output stream each time instances e_1 in ES_1, e_2 in ES_2,..., e_n in ES_n are received in the specified order. Then, sequence denotes that $\forall i$, occurrence e_i "is received before" occurrence e_{i+1}. The attribute named *context* of the event e contains all events $e_1, e_2, \ldots, e_n$.

Example 3.9. Let's consider the event streams ES_1 and ES_2 in Figure 3.6, the output of $SEQ(ES_1, ES_2)$ will be:

- An event e_1 at time t2 with the event parameters $e_{1,1}$ and $e_{2,1}$, such that $e_1.context = \{e_{1,1}, e_{2,1}\}$;
- an event e_2 at time t5 with the event parameters $e_{1,2}$ and $e_{2,3}$, such that $e_2.context = \{e_{1,2}, e_{2,3}\}$

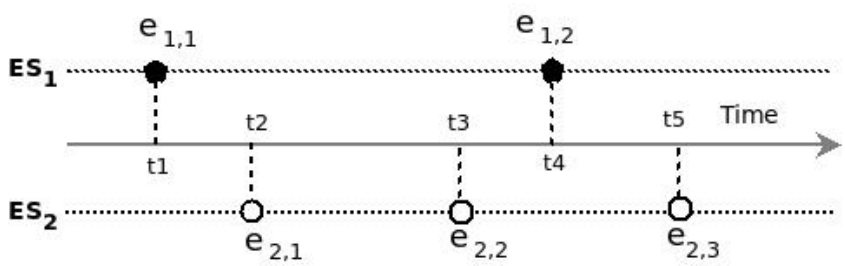

Figure 3.6 – Example situation 2. The time associated to events represents the time at which the events are processed by the operator.

3.3.5 Window operators

Window operators partition an event stream into finite parts of the original event stream. Let's consider an input stream $Stream(E)$. We denote by $Stream_f(E)$, a finite part of $Stream(E)$. A window operator applied on $Stream(E)$ results in a stream of finite streams, which we denote $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$. The way each finite stream is constructed depends on the window specification, which can be *time-based* or *tuple-based*.

[3] We omitted the meta-attributes

Time based windows

Time based windows define windows using time intervals.

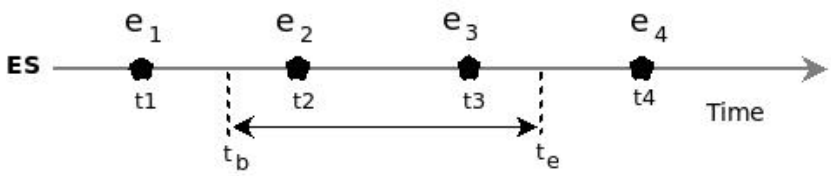

Figure 3.7 – Example situation 3. The time associated to events represents the time at which the events are received by the operator.

Definition 3.10. (Fixed Window) Let ES be an event stream, and $[t_b, t_e]$ be a time interval. Then, $win : within_{(t_b,t_e)}(ES)$ denotes the part of the event stream ES containing events between t_b and t_e. We have:
$win : within_{(t_b,t_e)}(ES) = ES_{f,1}$, such that $\forall e \in ES, e \in ES_{f,1}$ iff $t_b \leq e_i.receptionTime \leq t_e$.
The output stream contains a single finite event stream $ES_{f,1}$.

Example 3.10. Let's consider the example depicted in Figure 3.7. Therefore, the output of $win : within_{(t_b,t_e)}(ES)$ is the finite event stream $\{e_2, e_3\}$

Definition 3.11. (Landmark window) Let ES be an event stream, and t_b be a time span. Then, $win : since_{t_b}(ES)$ denotes the part of the event stream ES containing events from the time point t_b. We have:
$win : sincet_b(ES) = \{ES_{f,1}, ES_{f,2}...,\}$, such that $\forall i, e_i \in ES_{f,i}$ iff $e \in ES \wedge t_b \leq e_i.receptionTime$.
A finite event stream $ES_{f,i}$ is produced in output each time an event $e_i \in ES$ occurs, satisfying $e_i.receptionTime \geq t_b$.

Example 3.11. Let's consider the example depicted in Figure 3.7. Therefore, the output of $win : since_{t_b}(ES)$ is the finite event streams $\{e_2\}$, $\{e_2, e_3\}$, $\{e_2, e_3, e_4\}$, ...

Definition 3.12. (Sliding window) Let ES be an event stream, t_w and t_s two time durations. Then, $win : sliding_{(t_w,t_s)}(ES)$ denotes the sliding window of the event stream ES, having a time width t_w, which slides every t_s time duration. We have:
$win : sliding_{(t_w,t_s)}(ES) = \{ES_{f,1}, ES_{f,2}...,\}$, such that each $ES_{f,i}$ contains events from stream ES produced during last t_w time units. The finite event streams in the sequence are produced each t_s time unit. That is, if $ES_{f,i}$ is produced at time t, then $ES_{f,i+1}$ will be produced at time $t + t_s$.

Example 3.12. Let's consider the example depicted in Figure 3.8. Therefore, the output of $win : sliding_{(t_w,t_s)}(ES)$ are the finite event streams $\{e_1, e_2\}$ and $\{e_4\}$.

Size bounded windows

A size bounded window defines the number of events for each window. We distinguish between size fixed windows and sliding size fixed windows.

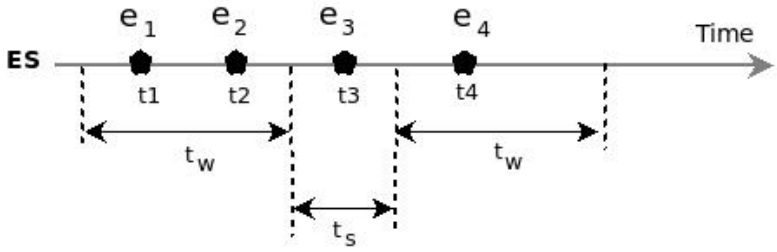

Figure 3.8 – Example situation 4. The time associated to events represents the time at which the events are processed by the operator.

Definition 3.13. (Fixed size windows) Let ES be an event stream, and nb a number. Then, $win : batch_{nb}(ES)$ denotes a partition of the event stream ES in finite event stream $ES_{f,1}, ES_{f,2}, ...$ such that each finite event stream contains nb most recent events from ES. The finite event streams $ES_{f,1}, ES_{f,2}, ...$ are non-overlapping.

Example 3.13. Let's consider the event stream $ES = \{e_1, e_2, e_3, e_4, e_5, e_6, \ldots\}$. Then, $win : batch_3(ES)$ will result in finite event streams $\{ES_{f,1}, ES_{f,2}, ...\}$ such that $ES_{f,1} = \{e_1, e_2, e_3\}$, $ES_{f,2} = \{e_4, e_5, e_6\}$, and so on.

Definition 3.14. (Sliding fixed size window) Let ES be an event stream, and nb, m two numbers. Then, $win : mbatch_{(nb,m)}(ES)$ denotes a sliding windows consisting in last nb events from ES. The windows slides each time m event occurs. We have:
$win : mbatch_{(nb,m)}(ES) = \{ES_{f,1}, ES_{f,2}, ...\}$ such that each $ES_{f,i}$ contains nb most recent events from ES, and $ES_{f,i+1}$ is started after m events are received in $ES_{f,i}$ (moving windows). As result, an event instance may be part of many finite event streams, specifically if $m \leq nb$. In such cases, they are overlapping.

Example 3.14. If we consider windows of size nb=3 moving after each m = 2 events, that is $win : mbatch_{(3,2)}(ES)$, the event stream $ES = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, \ldots\}$ will be partitioned into finite event streams $\{ES_{f,1}, ES_{f,2}, ES_{f,3}, \ldots\}$ such that $ES_{f,1} = \{e_1, e_2, e_3\}$, $ES_{f,2} = \{e_3, e_4, e_5\}$, $ES_{f,3} = \{e_5, e_6, e_7\}$, and so on.

Remark: In practice, window operators are used in conjunction with other operators. They specify the scope over which the associated operators should operate.

3.3.6 Aggregation operators

Applied on a stream of finite streams $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$, an aggregator operator $aggregate_{attr,\ aggrAttr}(Stream(Stream_f(E)))$ computes for each finite stream $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, the aggregated value of the attribute *attr* over events occurrences in $ES_{f,i}$. The attribute *attr* is the name of an attribute of E, that is $\exists \hat{t}_j$ such that $\hat{t}_j$ is an attribute type of E and $name(\hat{t}_j) = attr$.
The output is a event stream of type $A : \langle aggrAttr : D \rangle$[4] where $D = def(\hat{t}_j)$.

[4]We omitted the meta-attributes

Aggregate operators include *max, min, count, avg* and *sum.*

Max

The max operator compute the maximum value of the attribute over the event instances in $ES_{f,i}$.

Min

The min operator compute the minimum value of the attribute over the event instances in $ES_{f,i}$.

Count

The $count$ operator compute the number of event instances in $ES_{f,i}$ with the specified attribute.

Avg

The avg operator compute the average value of the attribute over the event instances in $ES_{f,i}$.

Sum

The sum operator compute the sum of the attribute's values over the event instances in $ES_{f,i}$.

3.3.7 Selection operators

A selection operator takes as input a stream of finite streams $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ...\}$ and produce as output an event stream of type E, with values $e_1, e_2, \ldots$ such that for all i, each e_i is a selection of a particular event from $ES_{f,i}$. The choice of that particular occurrence depends on the selection operator:

- **First occurrence**: $first(Stream(Stream_f(E)))$.
  For each finite stream $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, selects the first event occurrence. For example, if we apply the first operator to the stream of finite streams $\{\{e_1, e_2\}, \{e_3, e_4, e_5, e_6\}, \{e_7, e_8\}, \ldots\}$, we obtain as output the stream $\{e_1, e_3, e_7, \ldots\}$.

- **Last occurrence**: $last(Stream(Stream_f(E)))$.
  For each finite stream $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, selects the last event occurrence. For example, if we apply the last operator to the stream of finite streams $\{\{e_1, e_2\}, \{e_3, e_4, e_5, e_6\}, \{e_7, e_8\}, \ldots\}$, we obtain as output the stream $\{e_2, e_6, e_8, \ldots\}$.

3.3.8 Flatten

The *flatten* operator takes as input a stream of finite event streams $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$ and produces as output an event stream of type E, which contains the concatenation of events in $ES_{f,1}, ES_{f,2}, ES_{f,3}$, and so on. For example, if we consider the stream of finite streams $\{\{e_1\}, \{e_2, e_3\}, \{e_4, e_5, e_6\}, \ldots\}$, the flatten operator produces the output stream $\{e_1, e, e_2, e_3, e, e_4, e_5, e_6, e, \ldots\}$ where e is a special "empty" event that indicates the end of each finite stream in the output stream. We assume that for all event type E, the empty event e satisfies $e \equiv E$.

3.3.9 Computing the meta-attributes of events in output streams

An event stream composition operator $op(ES_1, ...ES_n)$ produces in output an event stream $ES' = \{e'_1, ..., e'_k, ...\}$. An event e'_k is produced by processing a finite set of events $\{e_{i,j} \mid \forall i, j\ e_{i,j} \in ES_i\}$ from the input streams. We say that the events $e_{i,j}$ are *event parameters* for the event e'_k, and we write $\{e_{i,j}\} \vdash e'_k$. This section specifies how to compute the value of meta-attributes of an event e'_k from the event set $\{e_{i,j}\}$.

Detection time

Whe have the following rule: $\{e_{i,j}\} \vdash e'_k \Rightarrow e'_k.detectionTime = min_{i,j}\{e_{i,j}.detectionTime\}$.

Production time

The production time of an event e'_k is the time at which the operator produces the event e'_k, which is not related to the production time of its event parameters.

NotificationTime

The notification time of an event e'_k is the time at which the outputs the event e'_k in the output stream, which is not related to the notification time of its event parameters.

Producer identifier

The producer identifier of a composite event e'_k is the the identifier of the component which executes the operator, which is not related to the producer identifiers of its event parameters.

Priority

The priotity of an event e'_k can be defined in a fixed way by providing a priority value for all events produced, or it can be derived from the priorities of its events parameters. In order to allow to choose between the two strategies, we let the user specify a function $f : \{integer\} \longrightarrow integer$, which takes as input a set of integers and returns an integer, such that if $\{e_{i,j}\} \vdash e'_k$,

then $e'_k.priority = f(\{e_{i,j}.priority\})$. The function f is called a *priority function*. Examples of priorities functions are:

- constant functions which define fixed priorities, that is $f = c,\ c \in \mathbb{N}$.
- aggregate functions like *min, max sum, avg*[5].

3.4 Event stream composition

3.4.1 Complex event stream, simple event stream

As mentionned before, the output an event stream composition operator is also an event stream. This makes us distinguish between two categories of event streams, namely *simple event streams* and *complex event streams*.

Complex event streams are event streams generated by event stream composition operators, in difference to simple event streams which are generated by event producers. As consequence, the output of an event stream composition expression, is a complex event stream.

We adopt the same naming convention for event instances. In particular, we will refer to event occurrences that feed complex event streams as *complex events*, and event occurrences that feed simple event streams as *simple events*.

3.4.2 Event stream composition expression

Stream based operators can be chained in order to produce complex event streams that capture particular situations. Event stream composition is defined by an event stream composition expression which is defined as:

- $A = ES$ is an event stream composition named A, where ES denotes an event stream.
- $A = Op(A_1, ...A_n)$ where Op is a stream based operator and $\forall i \in [1..n]$, A_i is an event stream composition expression. The components A_i are referred to as *subexpressions* of the event stream composition expression A.

An event stream composition expression A defines an event stream having the same name A.

For examle, let's consider the event type $MeterMeasure$: $\langle$ *meterID : string, realPower : double*$\rangle$ and the simple event stream $ES = Stream(MeterMeasure)$, we can define a complex event stream *ComplexStream* that computes the aggregated real power of meter *'AMI100'* between time 10 and 50 as follows:
$ComplexStream = avg_{realPower,\ avgP}(win:within_{(10,50)}(filter_{meterID='AMI100'}(ES)))$. The definition of the *ComplexStream* starts by filtering the event stream ES on the predicate *meterID='AMI100'*. Then, the result is used to compute a fixed windows (*win:within*) of events

[5]The usual average function, but rounded to the closest integer value

between time 10 and 50. The output stream, which is a finite stream is then aggregated on the attribute *realPower*. The aggregated value can be retrieved by accessing the *avgP* attribute of the complex event *e* in the *ComplexStream* complex output stream.

3.4.3 Labelled event stream composition expression

It is possible to assign an indentifier ID to each operator Op within an event stream composition expression as follow: $ID@Op$. We refer to this as a *labelled event stream composition expression*.

For example, $f@filter_{meterID='AMI100'}(ES)$ is a labelled event stream composition expression, where the filter operator is identified by f.

3.4.4 Well formed event stream composition expression

An event stream composition expression must be well formed. More precisely, the input of each stream based operator Op_i implicated in the event stream composition expression must be consistent with the operator definition.

For example, the input of an aggregate operator is a stream of finite streams, which can be computed using windows operators as in the *ComplexStream* example. Moreover, the result of a windows operator, which is a stream of finite event streams, cannot be given directly as input to a filter operator as in the expression $FilteredStream = filter_P(win: sliding_{(t_w,t_s)}(ES))$. This is due to the fact that the filter operator takes as input an event stream, an not a stream of finite event streams. In order to be consistent with the filter operator definition, the *FilteredStream* expression can be written $FilteredStream = filter_P(flatten(win: sliding_{(t_w,t_s)}(ES)))$.

3.4.5 Representing an event stream composition expression as a directed graph

An event stream composition expression can be represented as a directed graph. Graph representation of stream based event processing operations has been adopted in many systems including [STO13, Str15]. While allowing an easy human interpretation of stream processing operations, the graph representation is a good internal abstraction that allows to manipulate event composition expression using graph operations.

Definition 3.15. (Graph representation of a event stream composition expression) Let A be an event stream composition expression. Let **OP** be the set of event stream composition operators, and **ES** be the set of event streams. Then, A can be represented as a directed graph $A = (\mathbf{V}, \mathbf{E})$ where $\mathbf{V} \subseteq \mathbf{OP} \cup \mathbf{ES}$ is the vertex set and $\mathbf{E} \subseteq \mathbf{ES} \times \mathbf{OP} \cup \mathbf{OP} \times \mathbf{OP}$ is the edge set.
Let's assume that given a graph $G = (V, E)$, the functions *vertexset* and *edgeset* return its vertex set and edge set respectively: $vertexset(G) = V$, $edgeset(G) = E$. In particular, the vertex set and edge set of A are defined as follows:

- if $A = ES \mid ES \in \mathbf{ES}$, then we have: $vertexset(A) = \{ES\}$ and $edgeset(A) = \emptyset$;
- if $A = Op(A_1, ...A_n)$ where $Op \in \mathbf{OP}$ and $\forall i \in [1..n]$, A_i is an event stream composition expression, then we have:
  - $vertexset(A) = \bigcup_{i=1}^{n} vertexset(A_i) \bigcup \{Op\}$
  - $edgeset(A) = \bigcup_{i=1}^{n} edgeset(A_i) \bigcup \{ (firstcomponent(A_i), op), \forall i \in [1..n]\}$ where the function $firstcomponent(A)$ is defined as follows:

    if $A = ES$ where $ES \in \mathbf{ES}$, then $firstcomponent(A) = ES$;

    if $A = Op(A_1, ...A_n)$ where $Op \in \mathbf{OP}$ and $\forall i \in [1..n]$, A_i is an event stream composition, then $firstcomponent(A) = Op$.

For example, let's consider the event type $MeterMeasure$: ⟨ *meterID : string, realPower : double*⟩ and the simple event stream $ES = Stream(MeterMeasure)$. Let's also consider the event stream expression $A = avg_{realPower,\ avgP}(win : within_{(10,50)}(filter_{meterID='AMI100'}(ES)))$.
The event stream composition expression A can be rewritten as follows:
$A = avg_{realPower,\ avgP}(A_1)$, where
$A_1 = win : within_{(10,\ 50)}(A_2)$, where
$A_2 = filter_{meterID='AMI100'}(A_3)$, where
$A_3 = ES$.
Following Definition 3.15, we have:

- for A_3:

  - $vertexset(A_3) = \{ES\}$,

  - $edgeset(A_3) = \emptyset$.

- for A_2:

  - $vertexset(A_2) = \{filter_{meterID='AMI100'}\} \bigcup vertexset(A_3)$

  $= \{ES,\ filter_{meterID='AMI100'}\}$

  - $edgeset(A_2) = edgeset(A_3) \bigcup \{(firstcomponent(A_3),\ filter_{meterID='AMI100'})\}$

  $= \emptyset \bigcup \{(ES,\ filter_{meterID='AMI100'})\}$

  $= \{(ES,\ filter_{meterID='AMI100'})\}$

- for A_1:

  - $vertexset(A_1) = \{win : within_{(10,50)}\} \bigcup vertexset(A_2)$

  $= \{ES,\ filter_{meterID='AMI100'},\ win : within_{(10,\ 50)}\}$

  - $edgeset(A_1) = edgeset(A_2) \bigcup \{(firstcomponent(A_2),\ win : within_{(10,\ 50)})\}$

  $= edgeset(A_2) \bigcup \{(filter_{meterID='AMI100'},\ win : within_{(10,\ 50)})\}$

- for A:

  - $vertexset(A) = \{avg_{realPower,\ avgP}\} \bigcup vertexset(A_1)$

  $=\{avg_{realPower,\ avgP},\ win{:}within_{(10,50)},\ filter_{meterID='AMI100'},\ ES\}$

  - $edgeset(A) = edgeset(A_1) \bigcup \{(firstcomponent(A_1),\ avg_{realPower,\ avgP})\}$

  $= edgeset(A_2) \bigcup \{(filter_{meterID='AMI100'},\ win{:}within_{(10,\ 50)})\} \bigcup$

  $\{(firstcomponent(A_1),\ avg_{realPower,\ avgP})\}$

  $= \{(ES,\ filter_{meterID='AMI100'})\} \bigcup$

  $\{(filter_{meterID='AMI100'},\ win{:}within_{(10,\ 50)})\} \bigcup$

  $\{(firstcomponent(A_1),\ avg_{realPower,\ avgP})\}$

  $= \{(ES,\ filter_{meterID='AMI100'}),\ (filter_{meterID='AMI100'},\ win{:}within_{(10,50)}),$

  $(win{:}within_{(10,50)},\ avg_{realPower,\ avgP})\}$

Then, A can be represented as a graph $A = (V, E)$, having:
$V = \{avg_{realPower,\ avgP},\ win{:}within_{(10,50)},\ filter_{meterID='AMI100'},\ ES\}$ and
$E = \{(ES,\ filter_{meterID='AMI100'}),\ (filter_{meterID='AMI100'},\ win{:}within_{(10,50)}),$
$(win{:}within_{(10,50)},\ avg_{realPower,\ avgP})\}$.
The representation of A as a graph is given at Figure 3.9.

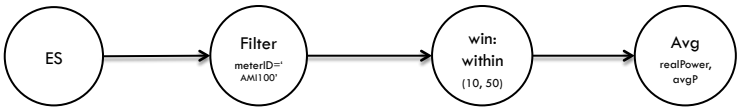

Figure 3.9 – Example of a graph reresentation of an event stream composition expression.

3.4.6 Developed form of an event stream

As mentionned in Section 3.2.3, an event stream $Stream(E)$ can be defined in terms of bounded event streams of type E. More precisely, if S is the set of source, then we have:
$Stream(E) = \{\bigcup Stream(E, s),\ s \in S\}$. This can be rewritten as an event composition expression using the disjunction operator as:
$Stream(E) = OR(Stream(E, s_1), ...Stream(E, s_n))$ such that $\forall i, s_i \in S$.
We refer to this as *the developed form* of the typed event stream $Stream(E)$. In the following, we assume the function *develop* which returns the developed form of a typed event stream $ES = Stream(E)$:
$develop(ES) = OR(ES^{s_1}, ..., ES^{s_n})$, having $\forall i \in [1..n],\ s_i \in S \wedge ES^{s_i} = Stream(E, s_i)$.

3.5 QoS requirements

Event stream composition must often meet particular QoS requirements regarding dimensions like latency, throughput, event priority etc. This section provides a model to represent the QoS

that characterizes an event stream composition.

3.5.1 QoS expression

Let us consider a set of QoS dimensions $\mathscr{D} = \{latency,\ throughput,\ memory, priority, ...\}$. Each QoS dimension $Q \in \mathscr{D}$ is associated to a domain D_Q, which corresponds to a set of QoS values. Given a domain D_Q, we assume a function *name(D_Q)* that returns its name,and a function *value(D_Q)* that returns the set of included values.

For example, if we consider the QoS dimension latency, then $D_{latency} \subseteq \mathbb{N}$, as the latency corresponds to a time delay (an expected value for measuring time belongs to the set of natural numbers). Thus, we have:

- *name($D_{latency}$)* = latency,
- *value($D_{latency}$)* = $\mathbb{N}$.

Definition 3.16. (QoS expression)
A QoS expression is of the form $(d\ :\ v)$ where

- d denotes a domain D_Q, $D_Q \in \mathscr{D}$,
- $v \in$ *value(D_Q)*,

For instance, the QoS expression (latency : 2000) specifies that the latency for notifying an event equals 2000 ms, assuming that the time unit is the millisecond.
The QoS expression (priority : 10) specifies that the value of the event priority is 10.

3.5.2 QoS tagged event stream composition expression

Let ξ be the set of all event stream composition expressions. A QoS tagged event stream composition expression is a 2-uplet s: $\langle expr,\ qos_specs\rangle$ where:

- $expr\ \in\ \xi$ is a labelled event stream composition expression.
- qos_specs is a set of QoS specifications $\{qos_spec_1, ..., qos_spec_n\}$, such that each QoS specification qos_spec_i is associated to an operator in $expr$.

A QoS specification qos_spec_i is defined as $ID[qos_1,\ ...,\ qos_n]$ where ID is the identifier of an event stream composition operator within $expr$, and each $qos_i = (name_i\ :\ value_i)$ is a QoS expression on a given QoS dimension named $name_i$.

Example 3.15. Let us consider the labelled event stream composition expression $expr = f@filter_{realPower>5}(ES)$. Then, s: $\langle expr, \{f[(memory : 10), (latency : 100)]\}\rangle$ defines an event stream composition expression *expr*, tagged with a QoS specification that defines the

memory expectation of the filter operator f as 10 Mo (assuming the memory unit is the megabyte), and its latency (processing time) as 100 ms (assuming the time unit is the millisecond).

3.6 Conclusion

In this chapter, we presented our model for event streams composition. We focused on the definition of concepts that are manipulated by an event stream composition system. First, we presented the concepts of event, event types, event occurrence and event streams. Then, we presented operators applicable to event streams with their associated semantic. After that, we introduced event stream composition and event stream composition expressions, as mechanisms for combining stream based operators in order to produce complex event streams.

In the next chapter, we will go one step forward, by presenting how the proposed model can be leveraged to define an event stream composition framework that deals with QoS.

4 The NETAH Framework

NETAH considers as input a consumer subscription which contains an event stream composition expression such as the one defined in Chapter 3, and generates the corresponding event stream composition network. Then, using a mapping algorithm, it deploys the generated event stream composition network on a distributed runtime environment, satisfying QoS requirements like memory occupation and latency. Section 4.1 presents the overview of NETAH and its architecture. Section 4.2 presents a subscription. Section 4.3 presents the main components of an event processing network, which are producers, consumers and event processing units. Section 4.4 focuses on the properties of the runtime environment its constraints. Section 4.5 presents the process used by NETAH for creating event stream composition networks from subscriptions. Section 4.6 focuses on event processing unit mapping, which is the process by which event processing units are associated to processing nodes on the runtime environment. Section 4.7 presents how NETAH deploys event processing units into their associated processing nodes on the runtime environment. Finally, Section 4.8 concludes this chapter.

Contents

4.1 Overview 56
4.2 Subscription 58
4.3 Event stream composition network 59
4.4 Runtime environment 64
4.5 Event stream composition network creation 65
4.6 Event processing unit mapping 68
4.7 Event processing unit deployment 73
4.8 Conclusion 75

4.1 Overview

NETAH is a framework which allows to implement event stream composition within a target runtime environment. In such an environment, NETAH distinguishes between producers which produce different types of event streams, and consumers which subscribe to complex event streams using *subscriptions*. A *subscription* includes an event stream composition expression (see Chapter 3), with the associated QoS requirements. When given a *subscription* as input, NETAH creates an *event stream composition network* which implements the event stream composition expression, and deploys it within the runtime environment in a way that satisfies the QoS requirements (See Figure 4.1).

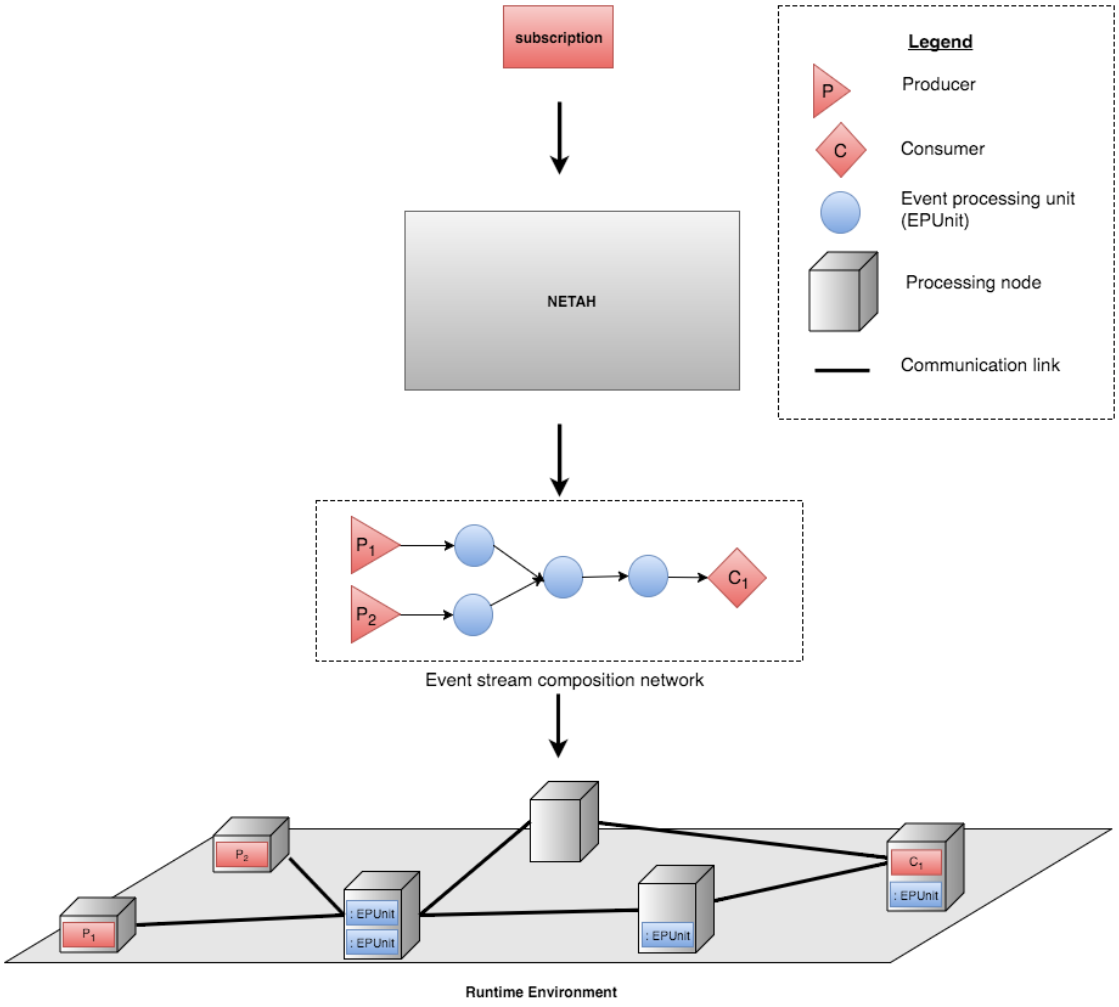

Figure 4.1 – Overview of NETAH

An event stream composition network consists of a set of *event processing units* (EPUs), *producers* and *consumers* organized as a directed acyclic graph (DAG). An event processing unit implements an event stream composition operator (see Section 3.3).

The architecture of the NETAH framework is depicted in Figure 4.2. It consists of three layers:

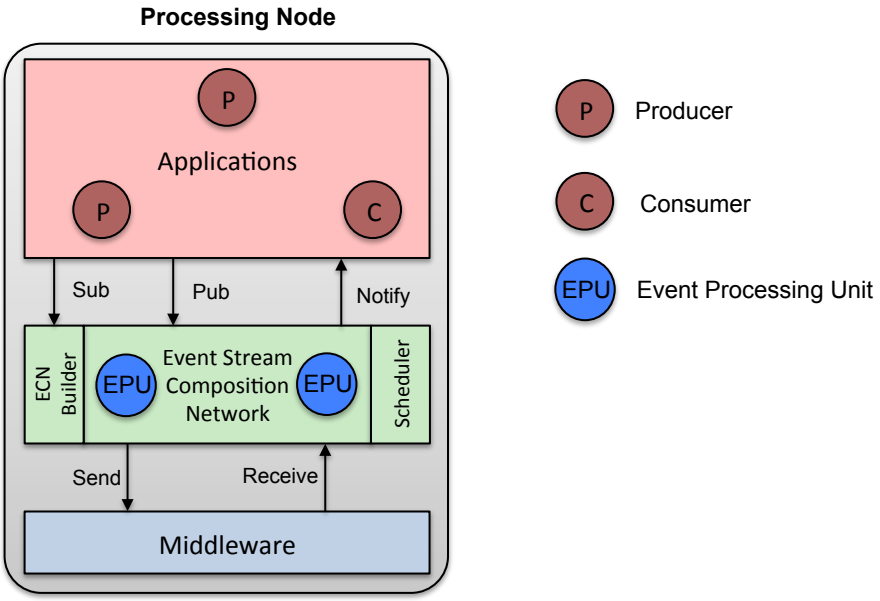

Figure 4.2 – Architecture of NETAH

- the application layer, which consists of producers which generate event streams, and consumers which subscribe to complex event streams via subscriptions;
- the event stream composition network layer, consisting of:

  - a set of distributed EPUs communicating via event channels. Those EPUs are created from consumers subscriptions and deployed in processing nodes of the runtime environment.

  - an event stream composition builder (ECN Builder), which creates event stream composition networks from subscriptions

  - A scheduler, which deploys EPUs on distributed processing nodes
- the middleware layer, providing a publish/subscribe communication style for event dissemination.

The class diagram associated to this architecture is presented in Figure 4.3.

The class *Node* represents a processing node of the runtime environment, and the class *Link* represents a communication link between two processing nodes. A processing node can host producers, consumers and EPUs. NETAH maintains information about the producers and consumers contained in the environment, with their corresponding location (the processing node on which they reside). Each processing node hosts a broker of a distributed publish/subscribe service, which provides publish/subscribe communication style between

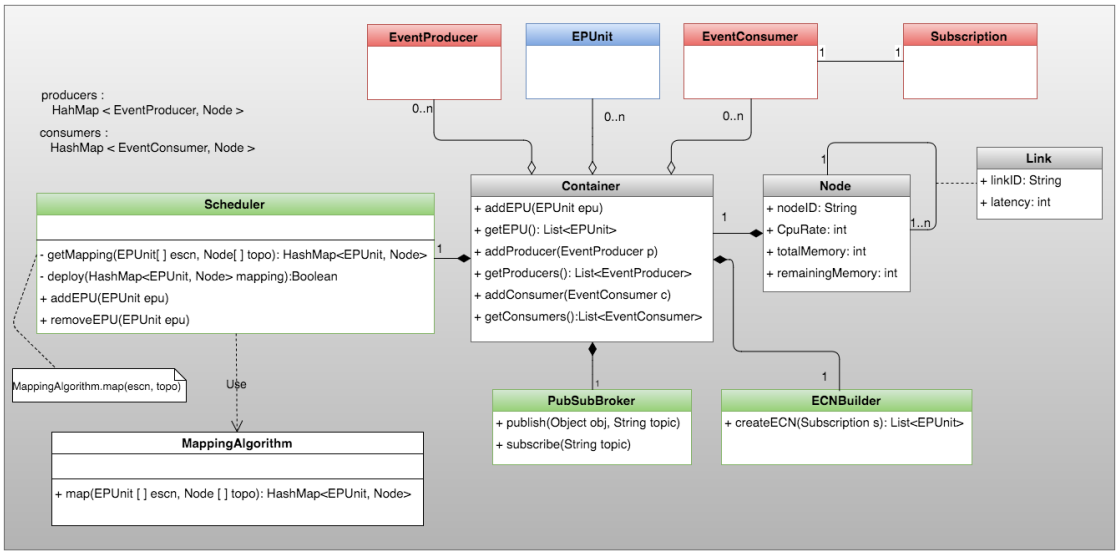

Figure 4.3 – Class diagram of a NETAH node

distributed producers, EPUs and consumers. Each processing node also hosts an event stream composition builder component (*ECNBuilder* class) and a scheduler component (*Scheduler* class).

4.2 Subscription

A subscription expresses the interest of a consumer in receiving an event stream of a specific type, with some correlated QoS requirements. A subscription is specified as a QoS tagged event stream composition expression (see Section 3.5).

We consider the following QoS domains:

- $D_{memory} \subseteq \mathbb{N}$, named *memory*, which denotes the memory requirement of the operator;
- $D_{time} \subseteq \mathbb{N}$, named *time*, which denotes the execution time required by the operator;
- $D_{pFunction} \subseteq \mathbb{N} \bigcup \{max, min, sum, avg\}$, named *pFunction*, which denotes the priority function used by the operator (see Section 3.3.9);
- $D_{selectionPolicy} = \{recent, chronologic, continous, priority\}$, named *sPolicy*, which denotes the selection policy associated to the operator. More on this in Section 4.3.3.

We also assume the functions *expression* and *qos* such as, when given a subscription will return the respective components of the subscription: $expression(s : \langle expr,\ qos_specs \rangle) = expr$, whereas $qos(s : \langle expr,\ qos_specs \rangle) = qos_specs$.

Figure 4.4 represents the class diagram of a subscription. A subscription contains a direct

acyclic graph which corresponds to the graph associated to its event stream composition expression (see Section 3.4.5).

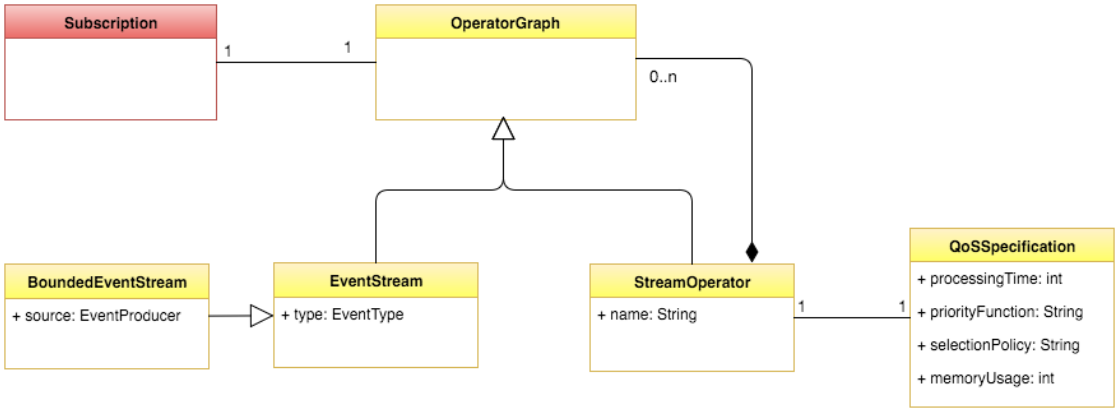

Figure 4.4 – Class diagram of a subscription

Example 4.1. Let us consider the event type $E_i = MeterMeasure : \langle$ *meterID : string, realPower : double*$\rangle$[1] and the event stream ES =*Stream(MeterMeasure)*. Let us also consider the labelled event stream composition expression $expr = f@filter_{realPower>5}(ES)$.
Then, $s : \{\langle expr, \{f[memory : 10, time : 2, pFunction : 2]\}\rangle\}$ determines a subscription s for which the event stream composition expression is $expr$ and the QoS specification $\{f[memory : 10, time : 2, pFunction : 2, sPolicy : recent]\}$ indicates that the filter operator f requires 10 memory units, its execution time is 2, it selection policy is "recent" and its priority function is the constant function $f = 2$, which defines the priority of complex events produced by f as 2.

4.3 Event stream composition network

An *event stream composition network* (see Figure 4.5) is the implementation of an event stream composition expression. It consists in a set of EPUs, producers and consumers organized into a directed acyclic graph (DAG). The edges of the DAG represent event streams. The producers produce event streams which are processed by EPUs. Each EPU receives some event streams in input and produces an event stream in output. The result is notified to the consumer.

4.3.1 Producer

A producer (see Figure 4.6) is a component that produces an event stream of a given type. A producer produces (the "*publish*" action) an event stream[2] of a particular type. It is hosted within the runtime environment, and the produced event stream reports on one or more aspects of its runtime environment. A producer component is modelled as a 2-uplet $P : \langle ID, E\rangle$ where:

[1] We omitted the meta-attributes.
[2] More precisely, a bounded event stream

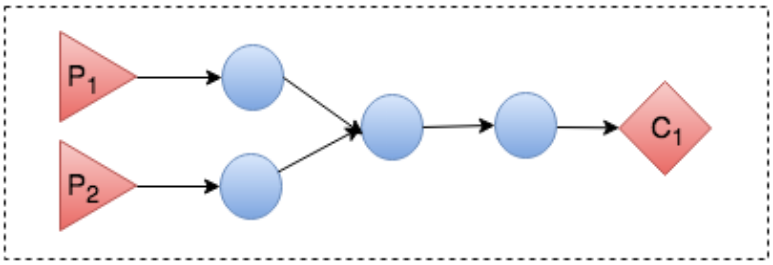

Figure 4.5 – An event stream composition network

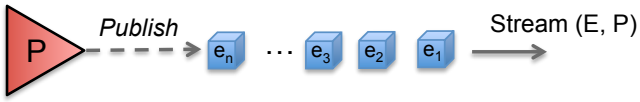

Figure 4.6 – A producer

- ID represents the identifier of the producer,
- E represents the type of events generated by the producer P.

A producer $P : \langle ID,\ E \rangle$ produces a bounded event stream denoted $Stream(E, ID)$.

Example 4.2. $P : \langle meter1,\ MeterMeasure \rangle$ determines a producer identified as *meter1* which generates an event stream of a type named *MeterMeasure*. The generated event stream is refered to as $Stream(MeterMeasure,\ meter1)$.

The class diagram of a producer is presented at Figure 4.7.

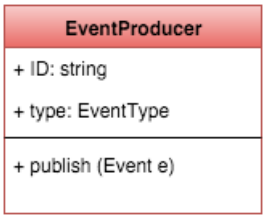

Figure 4.7 – Class diagram of a producer

4.3.2 Consumer

A consumer (see Figure 4.8) is a component that receives a specific type of event stream notifications. A consumer specifies the type of event stream it wants to receive using a subscription. A subscription specifies the complex event stream the consumer is interested in, with a QoS requirement (see Section 4.2). A consumer is modelled as a 2-uplet $C : \langle ID, s \rangle$ where:

- ID is the consumer identifier.

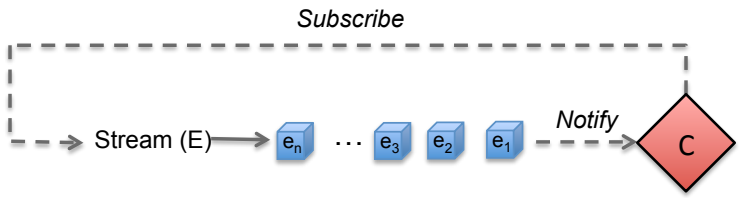

Figure 4.8 – A consumer

- *s* is the consumer subscription.

Figure 4.9 represents the class diagram associated to a consumer. The *subscribe* method is used to issue a subscription. The consumer receives event stream notifications via the invocation of its *notify* method, which implements the consumer reaction. We consider the processing logic of event consumers in response to incoming event streams as out of the scope of the NETAH framework.

Figure 4.9 – Class diagram of a consumer

4.3.3 Event processing unit

An *event processing unit* (EPU) is a component which implements an event stream composition operator (see Section 3.3). An EPU is composed by three types of components (see Figure 4.10):

- a set of input queues, on which input event streams are maintained.
- an operator, which implements a three step event processing logic: *fetch-process-notify*. In the first step (fetch), some events are selected from the input queues and marked as ready to be used to produce new complex events. In the second step (process), the events selected at step 1 are processed according to the operator semantic. Complex events produced are stored in the output queue. In the third step (notify), events in the output queue are notified either to other EPUs, or to interested consumers.
- an output queue, which contains events to be notified.

The class diagram associated to the event processing unit is depicted in Figure 4.11.

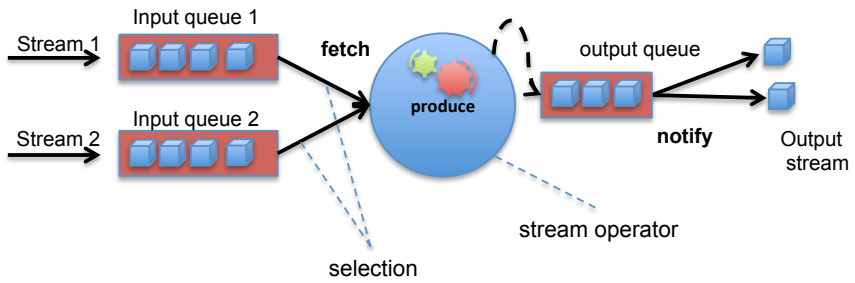

Figure 4.10 – Event processing unit

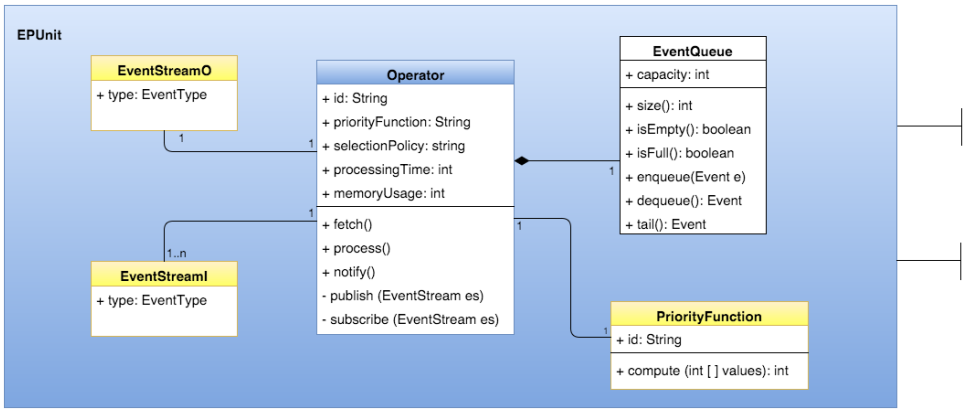

Figure 4.11 – Class diagram of an event processing unit

Selection policies

There are situations where an EPU has to choose among many events selected in input streams in order to produce an event in output stream. For example, consider the situation depicted in Figure 4.12 where an EPU implements the *and* operator on two event streams A and B. Suppose that two event occurrences a_1 and a_2 are received from input stream A at time t_1 and $t_2 > t_1$ respectively. According to the operator definition, a complex event has to be produced in the output stream when occurrences from input streams A and B are received. So, at time t_2, there is nothing produced in the output. Now let us suppose that an occurrence b_1 is received from input stream B at time $t_3 > t_2$. Now, the *and* operator should produce a composite event in the output. For that, it must be specified which occurrence between a_1 and a_2 should be considered for the construction of the complex event. It is the goal of selection policies, to specify the events to be selected in such situations. The notion of selction policy has been introduced in [CM94] with the name *parameter context*. We distinguish between four selection policies: *recent*, *chronologic*, *continuous* and *priority*.

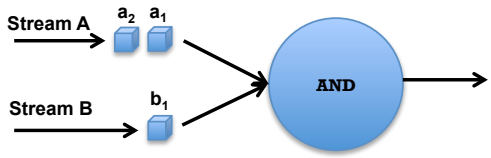

Figure 4.12 – Example situation where a selection policy should be applied

Recent Only the newest event occurrence is selected. In the example depicted in Figure 4.12, the event a_2 is selected between a_1 and a_2.

Chronologic Only the oldest event occurrence is selected. In the example depicted in Figure 4.12, the event a_1 is selected between a_1 and a_2.

Continuous All the event occurrences are selected. In the example depicted in Figure 4.12, the event a_1 and a_2 are selected. As result, two complex events are produced in the output stream, with event parameters $\{a_1, b_1\}$ and $\{a_1, b_2\}$ respectively.

Priority the event with the higher priority is selected in order to produce the complex event. In the example depicted in Figure 4.12, if we assume a_2 is higher priority than a_1, then a_2 is selected.

The selection policy of an operator is specified within the subscription.

Event Queue

An event queue is a component which is used to maintain a finite part of an event stream. The event queue is a priority-based FIFO queue whith a limited capacity. The priority relation noted $\prec$ is defined as follows:

Definition 4.1. (Priority relation) Let e_i and e_j be two events. Event e_i is said to be less priority than event e_j, which is noted $e_i \prec e_j$ iff:
$e_i.priority > e_j.priority \vee$
$e_i.priority = e_j.priority \wedge e_i.receptionTime \geq e_j.receptionTime$

Let Q, Q' be two event queues, and n be a natural number. Let's assume a function *set(Q)* which returns the set of all events in Q. The main operations applicable to an event queue are defined as follows:

size the function *size(Q)* returns the number of events in the queue:
$size(Q) = |set(Q)|$.

empty queue test the function *isEmpty* tests whether a queue is empty:

$$isEmpty(Q) = \begin{cases} true & \text{if } set(Q) = \emptyset \\ false & \text{otherwise} \end{cases}$$

full queue test the function *isFull(Q)* tests whether the queue is full. If the capacity of the queue is n, then we have:

$$isFull(Q) = \begin{cases} true & \text{if } size(Q) = n \\ false & \text{otherwise} \end{cases}$$

insertion the function *enqueue(Q, e)* adds the e event into the queue:
$enqueue(Q, e) = Q' \Rightarrow set(Q') = set(Q) \bigcup \{e\}$; The position of the inserted event e in Q' is defined as follows:

- if Q is empty, that is, $isEmpty(Q) = true$, then e is the only element of Q'.
- if Q is not empty, then
  - if $\forall e' \in set(Q),\ e' \prec e$, then e is the head of Q';
  - if $\forall e' \in set(Q),\ e \prec e'$, then e is the tail of Q';
  - if there is two consecutive events e_i and e_{i+1} in Q such that $e_i \prec e \wedge e \prec e_{i+1}$, then e is between e_i and e_{i+1} in Q'. In other words, the events e_i, e and e_{i+1} are consecutive in Q'.

In particular for insertion into a full queue, we propose two strategies:

ignore : do nothing. The new event is ignored;

replace : remove the less priority event from the queue (the tail of the queue) and insert the new event.

removal the function *dequeue(Q)* removes the higher priority event of a non empty queue, which is the head of the queue:
$dequeue(Q) = (Q', e) \Rightarrow e \in set(Q) \wedge set(Q') = set(Q) \setminus \{e\} \wedge \forall e' \in set(Q'),\ e' \prec e.$

tail of the queue the function *tail(Q)* removes the lower priority event of a non empty queue:
$tail(Q) = (Q', e) \Rightarrow e \in set(Q) \wedge set(Q') = set(Q) \setminus \{e\} \wedge \forall e' \in set(Q'),\ e \prec e'.$

4.4 Runtime environment

The event stream composition networks are deployed and executed in highly distributed and contrained runtime environments (see Figure 4.13). This section presents a model of the distributed runtime environment and its properties.

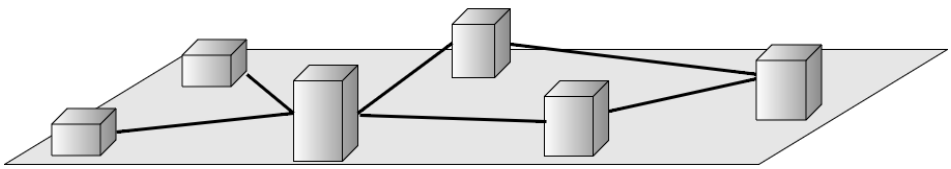

Figure 4.13 – The runtime environment

We consider that the distributed runtime environment is represented by an undirected graph $T = (\mathbf{N}, \mathbf{L})$ consisting in a set of distributed processing nodes $\mathbf{N}$ and their communication

links **L**. Processing nodes represent the device in which producers, consumers and EPUs are deployed.

Processing node

A processing node is characterized by its available*memory* and *CPU speed coefficient*.

The CPU speed coefficient is a number which indicates the relative speed of the processing node compared to others. The reference CPU coefficient is 1. We call *reference node* a processing node having CPU coefficient 1.

For example, a processing node having the CPU speed coefficient 2 is two times faster than a reference node.

Different processing nodes may have different available memory capacities. The event stream processing should be done on each processing node without violating its memory capacity.

Processing nodes that cannot host EPUs are simply given a zero memory capacity.

We assume that given a processing node n, the functions *amem* and *cpuCoef* return its available memory and CPU coefficient respectively.

Network link

The network link $l \in \mathbf{L}$ between two distinct processing nodes n_i and n_j is bidirectional and is characterized by its latency, which is the same for communication in both directions. We assume the function *lat* returns the latency of a given communication link.

4.5 Event stream composition network creation

Let P, C and O be the set of producers, consumers and EPUs respectively. Let $\mathcal{O} = P \cup O \cup C$.

For a given subscription s of a consumer $c \in C$, NETAH derives an event stream composition network ECN_s (see Figure 4.14), which is a DAG consisting in a set of producers $P_s \subseteq P$, a set of EPUs $O_s \subseteq O$ and a set of consumer $C_s \subseteq C$, all connected by directed edges $A_s \subseteq A$.
That is, $EPN_s = (\mathcal{O}_s, A_s)$, where $\mathcal{O}_s = P_s \cup O_s \cup C_s$ and $A_s \subseteq (P_s \times O_s) \cup (O_s \times C_s)$.

The set of EPUs O_s is created by NETAH. Each EPU in O_s implements an operator in the event stream composition expression contained in s. The producers P_s and consumer C_s are application components, and thus are not created by NETAH. NETAH maintains the references of all producers and consumers.

The component of NETAH which creates an event stream composition network from a subscription is called the *ECN Builder* (see Figure 4.2).

Consider a consumer c and a subcription s of c. Then, the event stream composition network

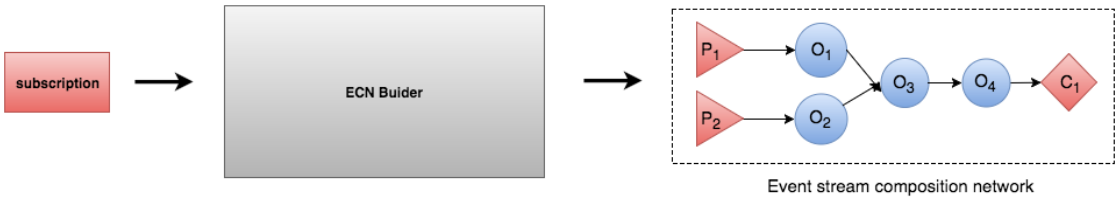

Figure 4.14 – Creation of EPUs from a subscription.

ECN_s associated to the subscription s can be derived from the graph representation of its event stream composition expression. The idea behind such a derivation is that bounded event streams define event producers, and stream operators define EPUs.

Let $expr$ be the event stream composition expression associated to s, that is *expr = expression(s)*. We assume that all the event streams in $expr$ are in their developed form (see Section 3.4.6). Let $G = (\mathbf{V}, \mathbf{E})$ be the graph representation of $expr$.

In order to define how ECN_s can be derived by G, let us introduce a function $h : \mathbf{V} \rightarrow O$ defined as follows:

- if a is a bounded event stream (see Section 3.2.3), that is $a = Stream(E_j, p_i)$ such that E_j is an event type and p_i is the ID of a producer, then $h(a)$ returns the event producer $P_i : \langle E_j,\ p_i \rangle$.
- if a is an event stream operator op, then $h(a)$ is an EPU epu_{op} which implements the operator op and applies all the specified parameters (selection policy, priority function, memory usage and processing time).

Then, $ECN_s = (\mathcal{O}_s, A_s)$ where:

1. $\mathcal{O}_s = \{\ h(a) \mid a \in \mathbf{V}\ \} \cup \{\ c\ \}$.
2. $A_s = \{\ (h(a),\ h(b)) \mid\ (a, b) \in \mathbf{E}\ \} \cup \{\ (h(r),\ c)\ \}$, where r is the downstream vertex of the graph G, that is the vertex with no outgoing edge.

After the direct acyclic graph EPN_s is generated, NETAH configures each vertex of the graph to subcribe to the appropriate input event streams. This is done according to the following rule:

- if $a = (o_i, o_j)$ is an edge in EPN_s, that is $a \in A_s$, then o_j subscribe to the output stream of o_i.

This mechanism ensures that, at the execution time, event streams will be disseminated in the way dictated by the event stream processing network.

At the end of this process, the set O_s contains the EPUs created for the subscription s. Those EPUs are ready to be deployed and executed in the runtime environment.

Example 4.3. Let us consider the event type $E_1 = MeterMeasure : \langle$ *meterID : string, realPower : double*$\rangle$ and the simple event stream $ES_1 = Stream(MeterMeasure)$. Let's also consider event producers $P_1 : \langle E_1, "source_1" \rangle$ and $P_2 : \langle E_1, "source_2" \rangle$. Finally, let us assume the event stream composition expression $expr = a@avg_{realPower,\ avgP}(w@win : within_{(10,50)}(f@filter_{realPower>2}(ES_1)))$, and the event consumer $c : \langle "consumer", s \rangle$ having a subscription s such that $expression(s) = expr$.

Let us compute the event stream composition network $ECN_s = (\mathcal{O}_s, A_s)$ associated to the subscription s.

By rewriting ES_1 in its developped form in $expr$, we have:
$expr = avg_{realPower,\ avgP}(\ win : within_{(10,50)}(\ filter_{realPower>2}(\ OR(\ ES_1^{P_1},\ ES_1^{P_2}\ ))))$, having $ES_1^{P_1} = Stream(E_1, P_1)$, and $ES_1^{P_2} = Stream(E_1, P_2)$. We assume that the QoS parameters associated to the stream operators are given by Table 4.1.

		Operators			
		Or	**Filter**	**Window**	**Avg**
Parameters	**Priority Function**	max	max	max	max
	Processing time	5	5	20	20
	Memory	10	10	50	50
	SelectionPolicy	priority	priority	priority	priority

Table 4.1 – Parameters associated to stream operators defined in *expr*.

The expression *expr* can be represented as a graph (see Figure 4.15).

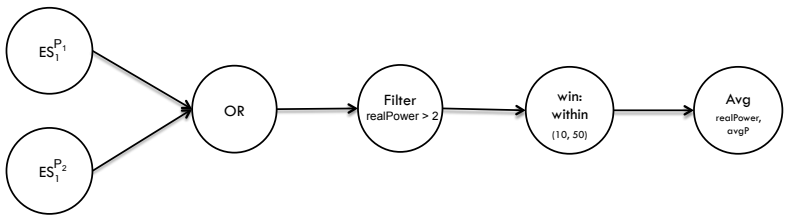

Figure 4.15 – Graph associated to the event stream composition expression *expr*.

For each vertex of the graph we have:

- $h(ES_1^{P_1}) = P_1 : \langle E_1, "source_1" \rangle$, that we simply refer to as P_1;
- $h(ES_1^{P_2}) = P_2 : \langle E_1, "source_1" \rangle$, that we simply refer to as P_2;
- $h(OR) = epu_{OR}$, such that epu_{OR} is an EPU implementing the operator OR of the graph G with the associated parameters;
- $h(Filter_{realPower>2}) = epu_{Filter}$, such that epu_{Filter} is an EPU implementing the operator $Filter_{realPower>2}$ of the graph G with the associated parameters;

- $h(win:within_{(10,50)}) = epu_{win}$, such that epu_{win} is an EPU implementing the operator $win:within_{(10,50)}$ of the graph G with the associated parameters;
- $h(Avg_{realPower,avgP}) = epu_{Avg}$, such that epu_{Avg} is an EPU implementing the operator $Avg_{realPower,avgP}$ of the graph G with the associated parameters;

Then we have $ECN_s = (\mathcal{O}_s, A_s)$ where:

1. $\mathcal{O} = \{P_1, P_2, epu_{OR}, epu_{Filter}, epu_{win}, epu_{Avg}, c\}$
2. $A = \{(P_1, epu_{OR}), (P_2, epu_{OR}), (epu_{OR}, epu_{Filter}), (epu_{Filter}, epu_{win}), (epu_{win}, epu_{Avg}), (epu_{Avg}, c)\}$

The resulting event stream composition network ECN_s is given in Figure 4.16. Then, NETAH configures EPUs and the consumer so that they subscribe to their corresponding input stream. In particular:

- the EPU epu_{OR} subscribes to the input streams $Stream(E_1, P_1)$ and $Stream(E_1, P_2)$;
- the EPU epu_{Filter} subscribes to the output stream produced by epu_{OR};
- the EPU epu_{win} subscribes to the output stream produced by epu_{Filter};
- the EPU epu_{Avg} subscribes to the output stream produced by epu_{win};
- the consumer c subscribes to the output stream produced by epu_{Avg}.

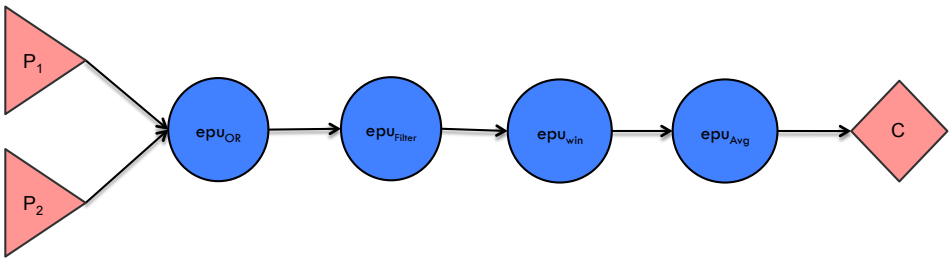

Figure 4.16 – The event stream composition network associated to the subscription s.

4.6 Event processing unit mapping

The EPUs created by NETAH have to be deployed in the runtime environment. Therefore, NETAH should decide for each EPU, the processing node on which it should be deployed (see Figure 4.17). We refer to this as the *EPU mapping*. The issue we face at this step is that there are many possible way to map EPUs to processing nodes in the runtime environment. Each possibility differs from others according to how it meets QoS requirements. We recall that the QoS dimensions addressed here are *memory occupation*, and *latency*. For example, deploying

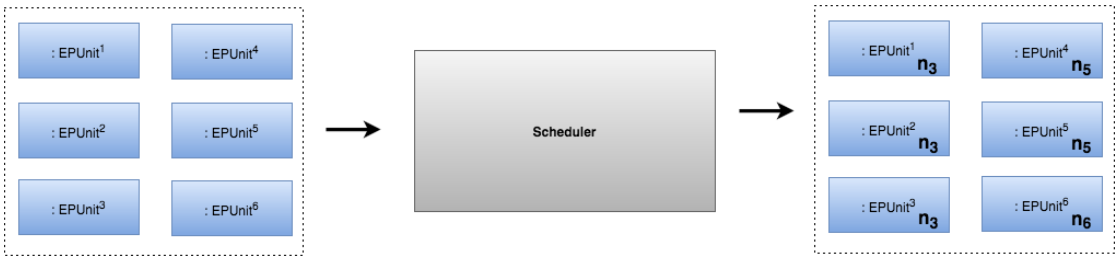

Figure 4.17 – Operator mapping

EPUs on a single node may potentially minimize the latency of events processing. In fact, this avoid the inter-node communication time, leading to a better latency. However, concentrating the event stream processing on one processing node may overflow its memory capacity, thus resulting in the violation of a QoS requirement (i.e., memory occupation). Therefore, the goal of the EPU mapping is to map EPUs on processing nodes in the best possible way, considering the memory limitation of each device while providing the better latency.
The component of NETAH which achieves EPU mapping is called the *scheduler* (see Figure 4.2).

In the following, we will focus on the EPU mapping problem. We will first illustrate the problem with an example. Then, we will present the properties that have to be satisfied by an EPU mapping algorithm. It is worth to mention that NETAH does not provides a default EPU mapping algorithm, but instead relies on an algorithm provided by the user.

The EPU mapping refers to the (close to) optimal selection of the physical processing nodes hosting the EPUs of an event stream composition network in order to satisfy a predefined global cost function. The EPU mapping 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 $\mathcal{O}(n^m)$ complexity. The EPU mapping problem is NP-complete.

The EPU mapping algorithm takes as input a specification of a physical network topology $T = (\mathbf{N}, \mathbf{L})$, which consists in a set of computing nodes **N** and their links **L**. The EPU mapping also considers the specification of the resources (i.e., memory and CPU coefficient) available on each processing node, and the latency of communication links. Figure 4.18 shows an example of network topology that comprises 9 computing nodes, each communication link being labeled with its corresponding latency. Table 4.2 shows the resources availability on each computing node.

The EPU mapping algorithm also takes as input an event stream composition network $ECN = (\mathcal{O}, A)$, which consists in a set of event streams producers $P \in \mathcal{O}$, a set of EPUs $O \in \mathcal{O}$ and a set of event stream consumers $C \in \mathcal{O}$. A represents the set of edges that connect the EPUs. Figure 4.19 shows an example of an event stream composition network, where P_1 and P_2 are two producers, C_1 is a consumer, o_1, o_2, o_3 and o_4 are EPUs. The EPUs are associated with measures or estimates of demand, such as the memory and CPU time that each EPU expects for processing a single input event. In our approach, those values are provided by the application developper. Table 4.3 shows the estimates associated to EPUs o_1, o_2, o_3 and o_4.

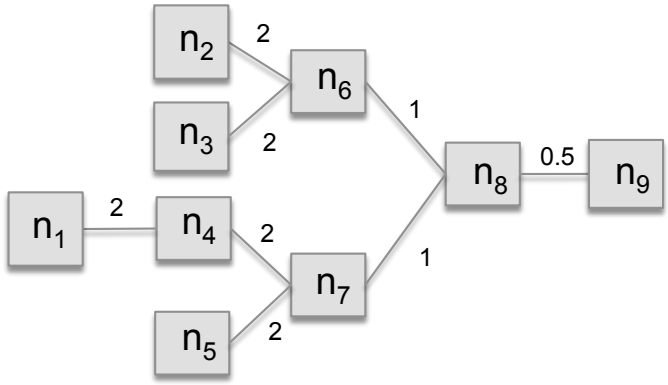

Figure 4.18 – A physical network topology

Table 4.2 – Resources availability on computing 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

We assume that each producer is permanently assigned to a network node. The same assumption hold with consumers. We say that producers and consumers are mapped.

Figure 4.20 shows an example of mapping of producers P_1, P_2 and consumer C_1, where P_1, P_2 and C_1 are mapped to nodes n_2, n_1 and n_9 respectively.

On the other hand, EPUs can be placed on arbitrary nodes having enough available resources for their execution.

The output of the EPU mapping algorithm is a mapping function $\lambda \subset \mathcal{O} \times \mathbf{N}$ that associates to each EPU, the node on the network topology in which it should be hosted. Figure 4.21

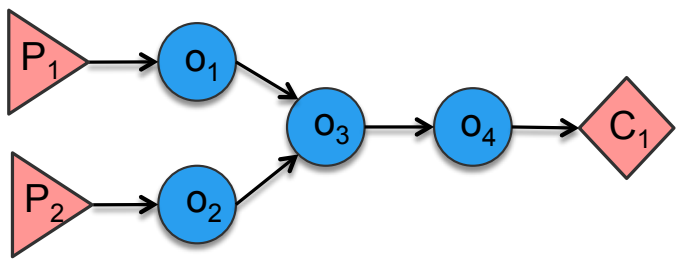

Figure 4.19 – Example of an event stream composition network

Table 4.3 – Estimates of the EPU resource requirements

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

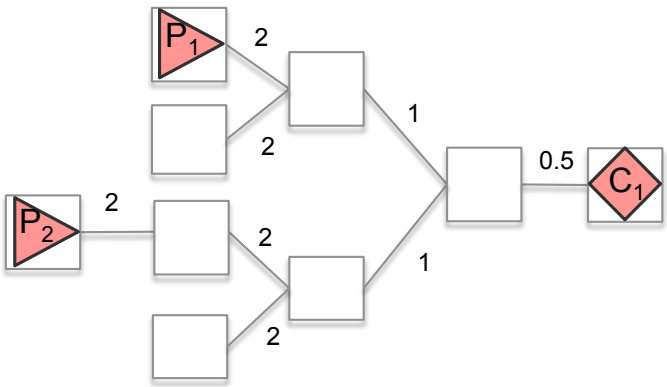

Figure 4.20 – Initial mapping of producers and consumers.

shows a possible mapping, where EPUs o_1, o_2, o_3 and o_4 are mapped to nodes n_6, n_4, n_8 and n_8, respectively. An EPU mapping algorithm assigns EPUs to processing nodes in a way that

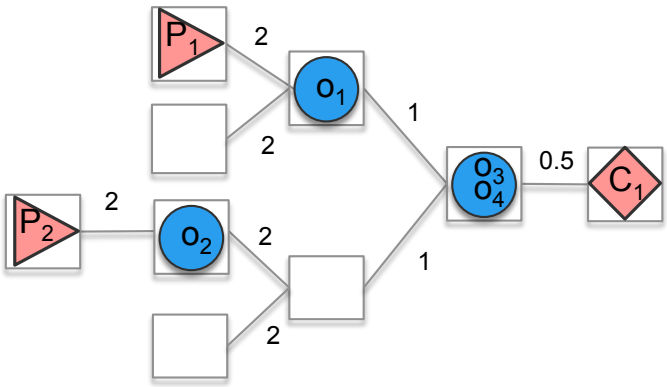

Figure 4.21 – Example of EPU mapping.

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.

In order to formally define the EPU mapping problem, let us consider the notations presented in Table 4.4.

Table 4.4 – Notations

Operator	Execution time
o	event processing unit
n	processing node
$init$	initial mapping of producers and consumers
$time(o)$	execution time of o on a reference node
$mem(o)$	memory required by event processing unit 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 event processing units connected by the edge a
$\lambda(o)$	the mapped location of event processing unit o

We formalize the EPU mapping problem as follows:

$$\underset{\lambda}{\text{minimize}} \;\; cost(\lambda) = \sum_{o \in O} p(o,\lambda(o)) + \sum_{a \in A} c(a)) \tag{4.1}$$

subject to:

$$\lambda(o) = init(o),\; if\; o \in P \cup C \tag{4.2}$$

$$\forall n \in N \sum_{o\,:\,\lambda(o)=n} mem(o) \leq amem(n) \tag{4.3}$$

where

$$p(o,n) = \frac{time(o)}{cpuCoef(n)} \tag{4.4}$$

$$c(a) = \begin{cases} 0 & \text{if for } a = (o_i, o_j),\; \lambda(o_i) = \lambda(o_j) \\ \beta(a) & \text{otherwise} \end{cases} \tag{4.5}$$

$$a = (o_i, o_j),\; \beta(a) = \sum_{e_i \,\in\, netPath(\lambda(o_i),\, \lambda(o_j))} lat(e_i) \tag{4.6}$$

Equation (4.1) states the cost of an EPU 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 (4.2) states that the mapping should be consistent with respect to the initial mapping of producers and consumers. Equation (4.3) states that the mapping should be defined such that no processing node is overloaded beyond its memory capacity. Equation (4.4) shows the formula that allows

to compute the processing time of a mapped EPU. Equations (4.5) and (4.6) show how to compute the network latency incurred by the communication between EPUs.

By using these formulas, we can compute as example the cost of the EPU mapping presented in Figure 4.21. First, note that this mapping is valid, since it does not violate Equation 4.2 and (4.3). Following Equation (4.4), we compute $p(o, \lambda(o))$ for EPUs 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 the communication between EPUs. 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 the communication between EPUs is then 7.5. Thus, the total cost of the EPU mapping is $cost(\lambda)$ = 23+7.5 = 30.5.

4.7 Event processing unit deployment

After NETAH computes the EPU mapping for a set of EPUs created after a subscription, it deploys each of these EPUs on the corresponding processing node (see Figure 4.22). The

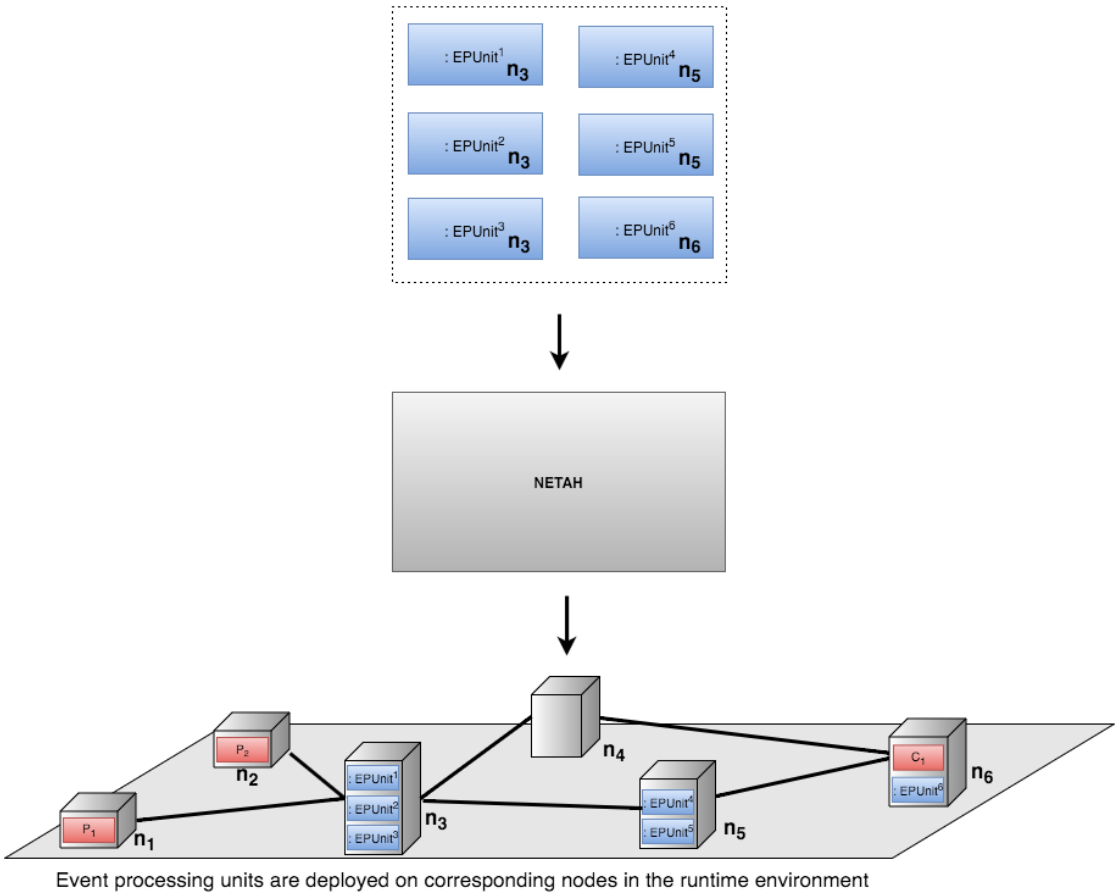

Figure 4.22 – Deployment of event processing units in the runtime environment

deployment is triggered by the scheduler collocated with the consumer which issued the sub-

scription. In the case of the example at Figure 4.22, it is the scheduler at node n_6 which triggers the deployment. Figure 4.23 presents the sequence diagram associated to the deployment phase.

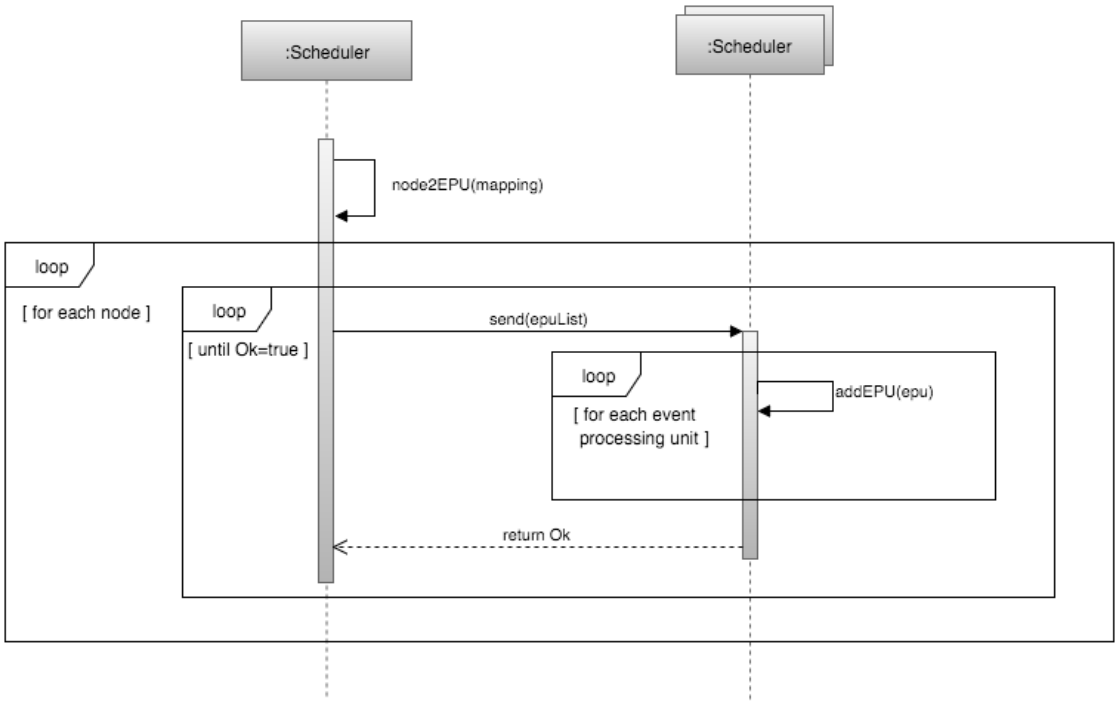

Figure 4.23 – The deployment sequence

The deployment happens as follows:

1. The local[3] scheduler computes for each designated processing node, its set of assigned EPUs.
   In the example at Figure 4.22, this results in $\{EPUnit^1, EPUnit^2, EPUnit^3\}$ for node n_3, $\{EPUnit^4, EPUnit^5\}$ for node n_5 and $\{EPUnit^6\}$ for the node n_6.

2. Then, the local scheduler sends to the remote[4] scheduler at each designated node its list of assigned EPUs.

3. When each remote scheduler receives its list of EPUs, it adds and starts them locally and sends a confirmation to the initial scheduler.

4. The process finishes when the initial scheduler receives all the confirmations from remotes schedulers.

[3] Relatively to the consumer which issued the subscription.
[4] See footnote 3.

4.8 Conclusion

This chapter described our framework for the generation and deployment of event stream composition networks in distributed runtime environments. We presented the core components of NETAH and its architecture. Producers represent components that produce event streams. Consumers represent components that subscribe to complex event streams by issuing subscriptions. The event streams generated by producers are processed by distributed event processing units (EPUs). The complex event stream generated in output is notified to the consumer. The communication between producers, EPUs and consumers is implemented by a publish subscribe middleware that provides a high level communication style which is adapted for event based communications.

NETAH considers as input a consumer's subscription, and generates an event stream composition network which implements the event stream composition expression contained in the subscription. An event stream composition networks consists of a set of producers,EPUs and consumers, organized into a directed acyclic graph. NETAH allows the definition of an EPU mapping algorithm, which is used for assigning the generated EPUs to the processing nodes of the runtime environment. Such an algorithm should satisfy the memory capacity of each processing nodes while ensuring a minimum end to end latency. Using such an EPU mapping algorithm, NETAH deploys the EPUs created for the given subscription in the runtime environment.

The next chapter is dedicated to the adoption of NETAH for generating and deploying event stream composition networks in smart grid environments.

5 NETAH in Smart Grids

This chapter presents the application of the NETAH framework in a smart grid. Section 5.1 presents a quick introduction to smart grids and smart grid data management issues, and details our approach for event stream composition in smart grids with NETAH. Section 5.2 presents an EPU mapping algorithm for NETAH in smart grids. Then, Section 5.3, presents a simulation of the smart grid network. Section 5.4 details the implementation of NETAH in a smart grid. Finally, Section 5.5 concludes this chapter.

Contents

5.1 Smart grid and NETAH 78
5.2 Operator mapping algorithm 80
5.3 Simulating the smart grid network 85
5.4 NETAH in the simulated smart grid network 89
5.5 Conclusion 95

5.1 Smart grid and NETAH

5.1.1 Overview of a smart grid

What is a smart grid?

A smart grid (see Figure 5.1) is as a modernised electricity grid that uses information and communications technology to monitor and actively control generation and demand in near real-time, which provides a more reliable and cost effective system for transporting electricity from generators to homes, businesses and industry [Dep14].

The smart grid is the result of the integration of sensing, embedded processing and digital communications to the electricity grid, which becomes observable (able to be measured and visualised), controllable (able to be manipulated and optimised), automated (able to adapt and self-heal), fully integrated (fully interoperable with existing systems and with the capacity to incorporate a diverse set of energy sources).

The literature [Eur06, Dep14] suggests the following attributes for the smart grid:

Minimise consumer bills: more efficient use of network assets helps reduce the need to invest in costly infrastructure and ultimately reduces the costs passed through to consumer bills.

Enable greater consumer and community participation: the smart grid can have a transformational impact on consumer and community interaction with the energy system. Smart meter systems will provide consumers with more accurate information on their energy use and suppliers will be able to offer more cost reflective tariffs that reward consumers for using energy at off-peak and lower price times or generating energy at peak times.

Enabling demand response and energy storage: with accurate and real time technical information on supply and demand, and options for balancing supply locally, complemented by new commercial arrangements such as flexible connection agreements, network operators will be able to free up existing capacity and make better use of existing assets.

Improve energy security and reliability: a more intelligent network that increases the visibility of real-time network use, as well as a means to control and manage the network more responsively, will improve the stability and reliability of the network. This will assist in the timely and efficient replacement of equipment, reducing the risk of any localised power outages or interruptions and ensure that power is restored more quickly when faults do occur.

Enable new low carbon technology to be deployed: heat pumps and electric vehicles will deliver significant carbon reductions, but will increase demand on the electricity network. Using smart grid technology to phase operating times, these devices can be incorporated into the network in conjunction with distributed generation to balance supply and demand, reducing the need for costly network reinforcement.

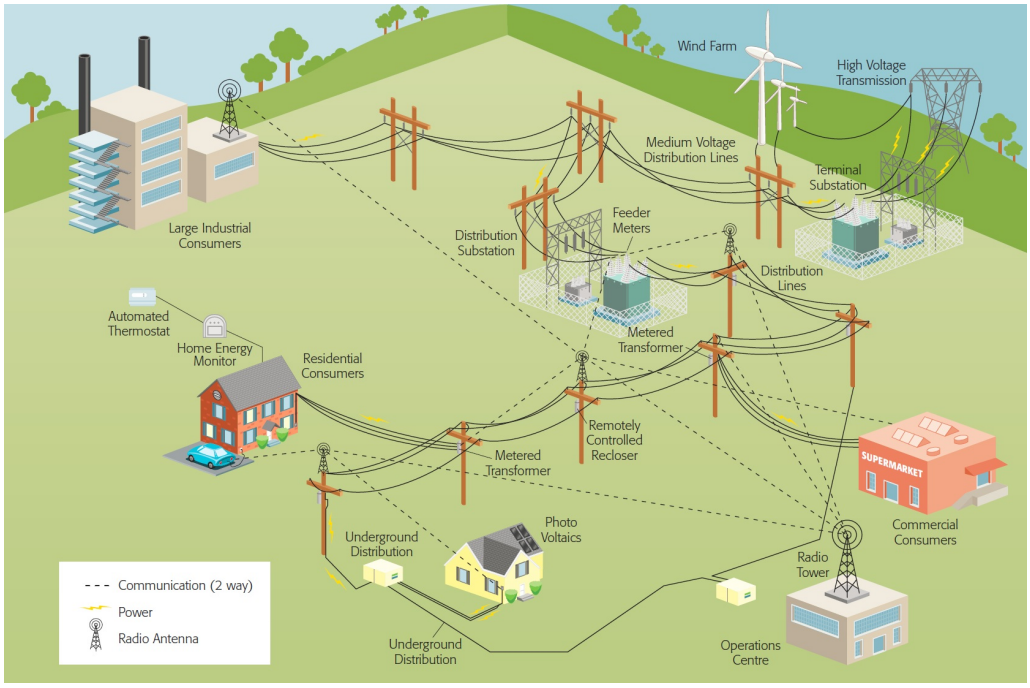

Figure 5.1 – A smart grid

Smart grid data management issues

Smart grid 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. Those data can be considered as event streams that refer to happenings of interest produced within the smart grid environment. Strategies to handle real-time event streams and notifications are critical for achieving smart grid utilities. Utility applications requires event streams to be filtered, aggregated and correlated in order to infer complex events that are sematically richer and useful. This requires event stream composition systems to be implemented in smart grids.

In order to meet real-time requirements, event streams must be processed on the fly and continuously, as they are flowing within the system. Therefore, event stream composition must be achieved in a distributed way across the smart grid network, using smart grid devices. Such devices include smart meters, data concentrators, smart sensors, etc. However, these are very contrained devices in terms of computational resources (memory and CPU). Thus, event stream composition should be distributed to these devices considering their resource limitations.

In addition to the computational resource limitations, there are limitations on the network resource, which generally integrates many communication channels, including power line carrier, wireless communication, wired communication. These communication channels are associated with different transmission delays which also have to be considered for event stream composition.

5.1.2 NETAH framework in a smart grid

The NETAH framework addresses the smart grid requirements in term of event stream composition:

Distribution level: The Smart grid requires the event stream composition to be distributed accross the smart grid network. The smart grid network is a special case of runtime environment on which NETAH can deploy event stream composition operators.

QoS constraints: The smart grid requires event stream composition to be done on each device considering its resources limitations. NETAH deploys event stream composition operators on each devices considering both their memory limitation and the expected end to end latency.

Figure 5.2 presents the application of NETAH to a smart grid. The smart grid network is considered as the runtime environment of NETAH. Smart grid devices (smart meters, data concentrators, sensors, etc...) can act as event stream producers or consumers. Examples of producers include smart meters, which generates event streams related to electricity consumption, and sensors, which generate event streams related to the state of electrical line. Examples of consumers are the utility servers, which consume complex event streams such as alarms derived by processing simple event streams. A consumer specifies the event streams he is interested in using a supscription. Such a subscription is processed by NETAH, which generates the corresponding event stream composition network and deploys it in the smart grid network in a way which is consistent with the resources limitations of smart grid devices.

5.2 Operator mapping algorithm

NETAH has to be provided with an EPU mapping algorithm which is used for assigning event processing units to processing nodes on the runtime environment. Such an algorithms should meet the constraints defined from Equations (4.1) to (4.6) at Section 4.6. In this subsection, we propose algorithms for EPU mapping in a smart grid environment.

Brute force approach.

The EPU mapping 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 Equations (4.2) and (4.3), 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 EPU mapping, 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

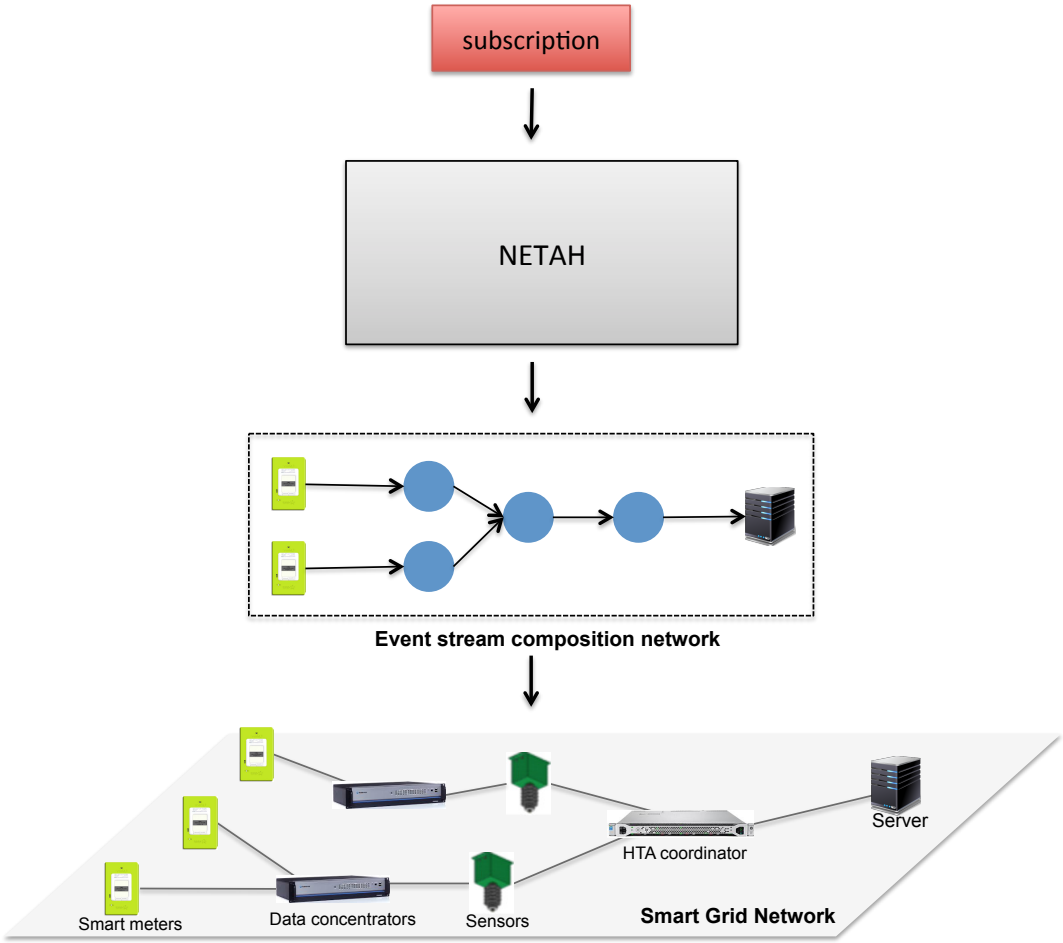

Figure 5.2 – The NETAH approach in smart grid

set of valid mappings, as shown in the following algorithm.

OpMapping(EventCompositionNetwork ecn, NetworkTopology topo, InitialMapping init)
$\lambda_{opt} \leftarrow null$;
$solver \leftarrow BinPackingSolver()$;
solver.constructBinPackingConstraint(ecn, 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 λ_opt;
```

The *OpMapping* 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 *OpMapping* algorithm is not appropriate for producing a result in an acceptable period of time for large event stream composition networks and network topologies. In order to deal with such large inputs, we proposed a greedy approach for EPU mapping.

Greedy approach.

In order to reduce the size of the inputs of the *OpMapping* algorithm, our idea consists in adopting a "divide and conquert" approach, in a greedy manner (see Figure 5.3).

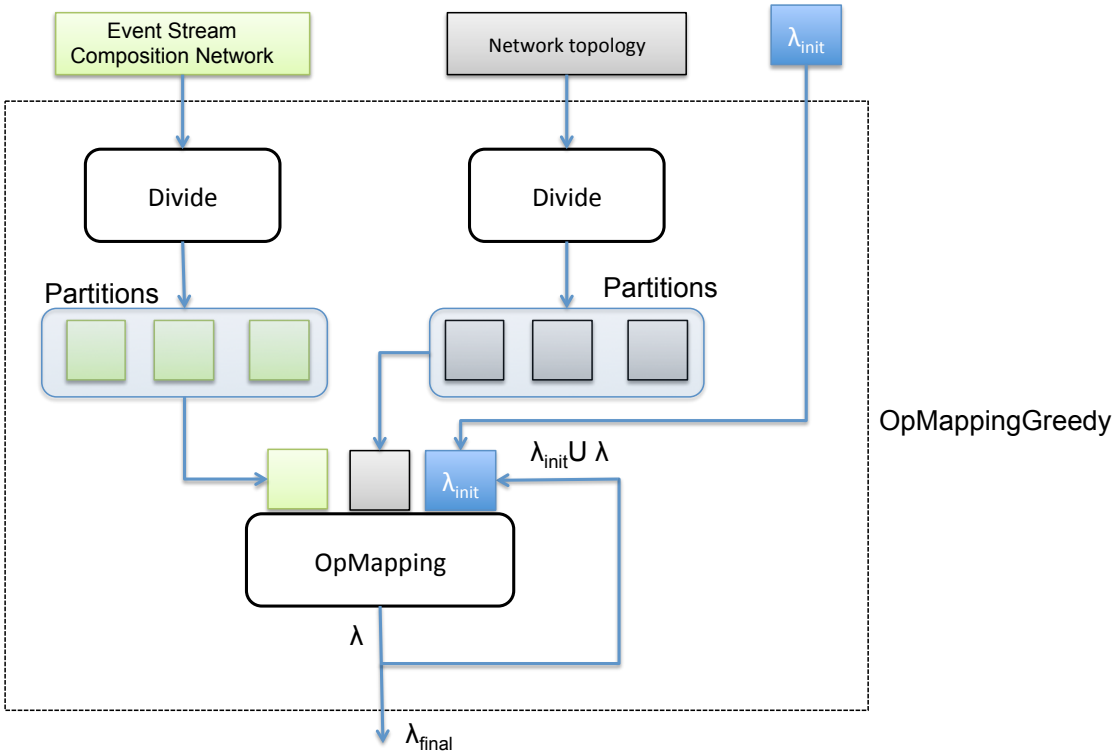

Figure 5.3 – Overview of the greedy approach

The idea is to incrementally map parts of the event stream composition network on specific parts of the network topologies using the *OpMapping* algorithm, combining the founded 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 stream composition network. Then, it should be specified how to compute the part of the network topology where a computed part of the event stream composition network should be mapped. In order to do that, we rely on the following hypothesis on event stream composition network and network topology respectively:

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

Computing subgraphs of the event stream composition network. Given the original event stream composition network, 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 $subgraph(ECN, P_i)$ that computes the subgraph associated to the producer P_i.

Example 5.1. For example, considering the event stream composition network in Figure 5.4, the result of the function $subgraph(ECN, P_2)$ is the subgraph that consists of the set of nodes $\mathcal{O}' = \{P_2, o_2, o_3, o_4, c\}$ and the set of edges $A' = \{(P_2, o_2), (o_2, o_3), (o_2, o_4), (o_3, c), (o_4, c)\}$.

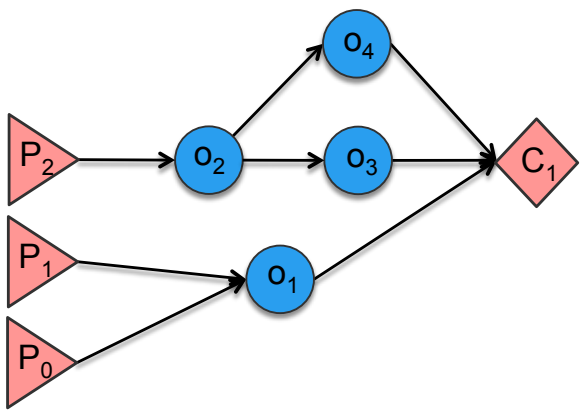

Figure 5.4 – Example of an event stream composition network

Computing a subgraph of the network topology. Once we compute a subgraph of an event stream composition network for a given producer P_i using $subgraph(ECN,\ P_i)$, 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 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 $init(c)$. Also, the size of the subgraph is of the order of $\mathcal{O}(log_m(n))$, where n corresponds to the number of nodes in the original network topology, assuming each node is connected to at most m nodes.
We assume that this subgraph is computed by the function $subgraphTopo(T, n_i, n_j)$.

Example 5.2. For example, considering the network topology in Figure 5.5, 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 $E' = \{(n_6, n_8), (n_8, n_9), (n_9, n_{10})\}$.

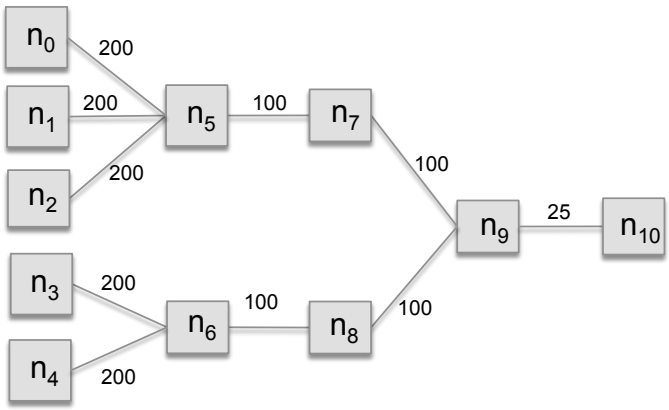

Figure 5.5 – Network topology

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

OpMappingGreedy(EventCompositionNetwork epg, NetworkTopology topo, InitialMapping init)
$\lambda \leftarrow init$;
for each producer P_i in epg **do**
  $epg' \leftarrow subgraph(epg, P_i)$;
  $topo' \leftarrow subgraphTopo(topo, init(P_i), init(c))$;
  $\lambda' \leftarrow OpMapping(epg', topo', \lambda)$;
  if λ' != null **then**
    $\lambda \leftarrow \lambda \cup \lambda'$;
    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 λ;

The *OpMappingGreedy* algorithm achieves local optimization for each computed subgraph of the original event stream composition network. 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 stream composition network fails, the algorithm stops and the mapping is considered as failed.

The experimental evaluation of *opMapping* and *OpMappingGreedy* algorithms is presented in Appendix A.

5.3 Simulating the smart grid network

This section presents the simulation of a smart grid network. Our simulation only considers the components of the smart grid network that are relevant to NETAH: network devices and communication links. Figure 5.6 presents the class diagram of smart grid network components.

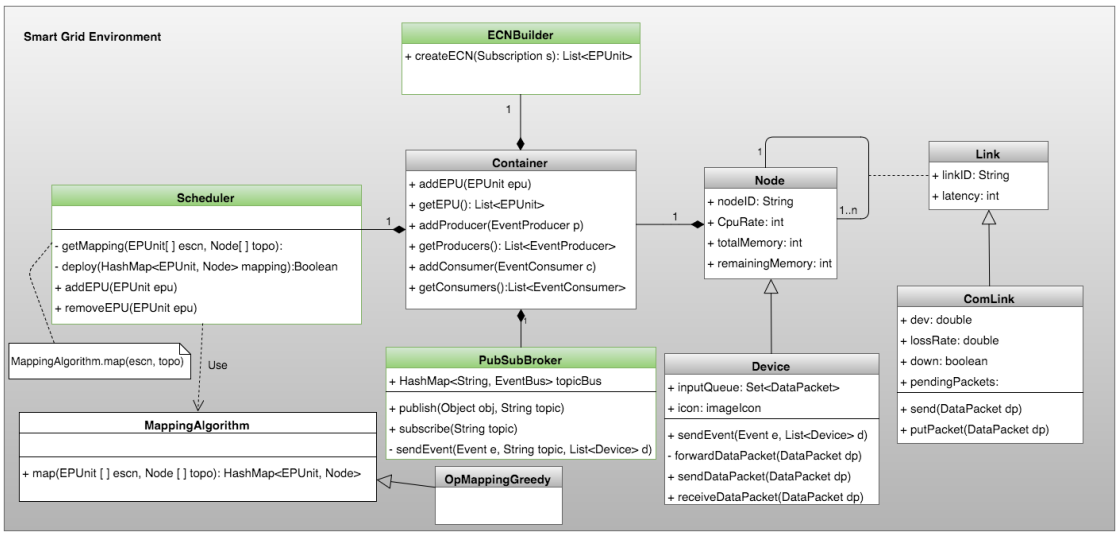

Figure 5.6 – Class diagram: the smart grid environment

Modeling a smart grid device

The *Device* class represents a smart grid device. Smart grid devices includes smart meters, sensors, data concentrators, HTA coordinators etc.

A smart grid device is identifiable by its name, and it can be assigned a quantity of memory (*availableMemory* attribute) and a CPU rate (*cpuRate* attribute).
A smart grid device also has an image (*icon* attribute) which will be used to display the device on a graphical user interface.
A smart grid device can be connected to other smart grid devices by network links. The *Device* class provides methods for data routing over the network, more on this in Section 5.3.1.
A smart grid device integrates the components of the NETAH architecture (See Section 4.1), that is a publish/subscribe broker, a scheduler and an event stream composition network builder (*ECNBuilder*). A smart grid device can host event producers, event consumers and event processing units.

Modeling a communication link

The *ComLink* class represents a communication link between two devices. This can be a wired link, wireless link or a power line link. The communication can happen in both directions. A communication link is identifiable by a name (*name* attribute), and can be assigned a given latency (*latency* attribute), a loss rate (*lossRate* attribute) and a state which indicates wether the link is down or not (*down* attribute).

Latency The latency of the communication link is the data transfer time over the link. Instead of having a fixed latency for every data transfer, we consider that the data transfert time is a random variable having a normal distribution $\mathcal{N}(latency, dev)$. The mean and standard deviation of the distribution are respectivelly defined by the attributes *latency* and *dev* of the *ComLink* class.

Loss rate The loss rate of a communication link defines the probability of a data loss during a transmission. Data which are considered as lost are retransmitted, resulting in an increase of the transmission latency. The communication link is then associated to a bernoulli variable, which decides the frequency of data loss. The probability of success (data loss) of the bernoulli variable is defined by the *lossRate* attribute of the *ComLink* class.

State The attribute *down* of a communication link is a boolean which indicates whether the link is in its "down" state or not. A link in the down state cannot convey data, while a link which is not in the down state can.

5.3.1 Routing data over the network

The data routing over the network is realized by the *Device* components. Data which are sended over the network are encapsulated into objects of the type *DataPacket*, which is presented in Figure 5.7.

The *source* attribute identifies the device at the origin of the message. The *origin* attribute

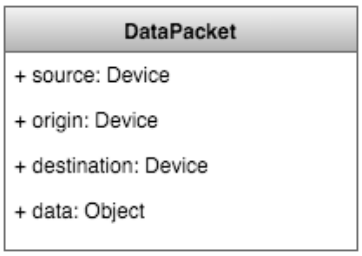

Figure 5.7 – Class diagram: producer

identifies the last device which forwards the message on the network. This information is useful for sending data in the right direction over bidirectional communication links. The *destination* attribute identifies the final destination of the data. Finally, the *data* attribute represents the data being transmitted.

In our setting, we assume that all smart grid devices share a global knowledge of the network topology. The delivery of a message msg from a device d_{source} to a destination d_{dest} happens as follows:

1. The device d_{source} encapsulate the message into a *DataPacket* object dp, and sets the *source* and *destination* attributes of dp as d_{source} and d_{dest} respectively.
2. The *origin* attribute of the packet is set to d_{source}.
3. The device d_{source} computes the shortest path to reach its destination, using the Dijkstra's shortest path algorithm.
4. The device d_{source} deduces the communication link over which the message should be routed, and put the packet on that link.
5. The link convey the packet dp to the device $d_{neighbor}$ located on its opposite endpoint.
6. When the device $d_{neighbor}$ receives dp, it checks whether he is the final the destination of the packet. If this is the case, the packet is arrived at its destination. In the opposite case, the device sends the packet to the next hop, by repeating the procedure starting from the instruction number 2.

5.3.2 Implementing a publish/subscribe communication style

In NETAH, publish/subscribe communication is topic based: a sender publishes a message to a topic, and a subscriber subscribes to a particular topic, receiving all message being published to that topic. Therefore the two main operations provided by a publish/subcribe service are *publish* and *subscribe* (see Figure 5.8).

In order to implement the publish/subscribe service, we relied on event bus proposed in [Eve16], which has been designed to replace traditional Java in-process event distribution

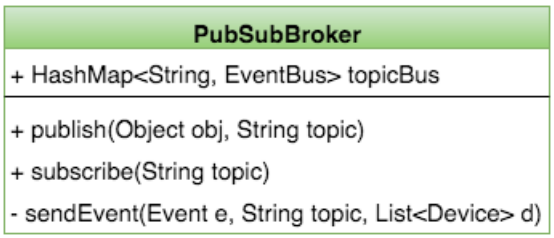

Figure 5.8 – Topic based publish/subcribe broker

using explicit registration.
A publish/subscribe broker maintains a set of event bus objects, each being associated to one topic (see the *topicBus* attribute). There is one publish/subcribe broker on each device.

We assume a shared[1] variable *topic2Device*, of the type *HashMap<String, Set<Device»* which maintains a correspondance between a topic and a set of devices.

Subscribing to a topic.

In order to subscribe to a topic, a subcriber calls the *subscribe* method of its publish/subscribe broker[2]. The publish/subscribe broker then do the following:

1. retrieves the event bus associated to the topic in the *topicBus* variable. If there is no such an event bus, a new event bus is created for the topic;
2. Register the consumer to the retrieved event bus. This allows the consumer to be notified when an event is published into the retrieved event bus.
3. Adds the device hosting the subscriber into the set of devices associated to the topic in the *topic2Device* variable. This information is useful for notifying the consumer in case events are produced in the topic from remote devices.

Figure 5.9 shows the sequence diagram of a subscription.

Publishing a message to a topic.

In order to publish a message to a topic, a sender calls the *publish* method of its publish/subscribe broker[3]. Then, the publish/subscribe broker retrieves the set of devices associated to the topic and sends the message to the publish/subscribe broker on each of those devices. When a publish/subscribe broker receives such a message, it retrieves the event bus associated to the topic (*topicBus* attribute). The message in then published into the retrieved event bus, which lets the event being notified to registered subscribers on that device.

[1] shared among all publish/subscribe brokers
[2] The one which is located on the same device
[3] The one which is located on the same device

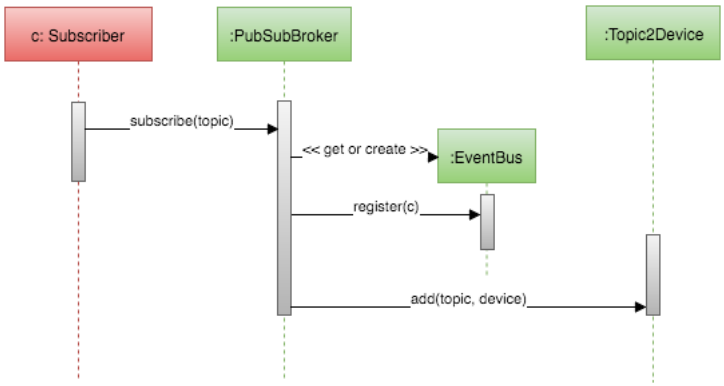

Figure 5.9 – Sequence diagram: subscribing to a topic

Figure 5.9 shows the sequence diagram of a publish.

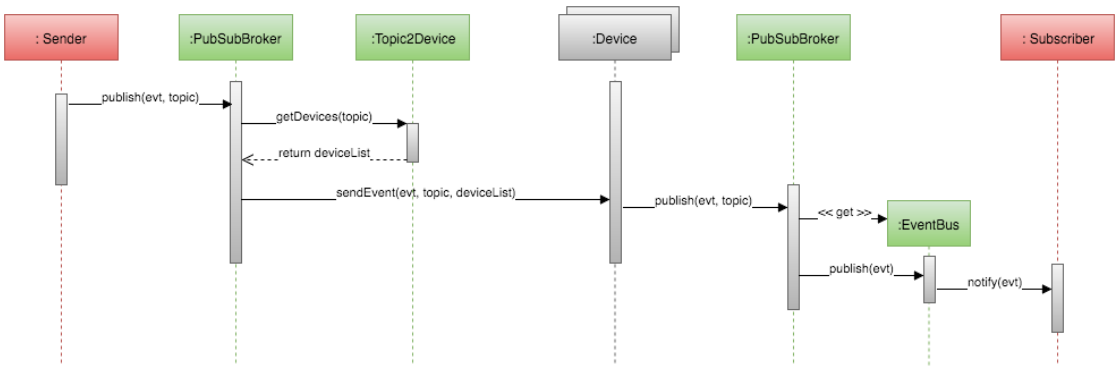

Figure 5.10 – Sequence diagram: publishing an event

5.4 NETAH in the simulated smart grid network

This section presents the adoption of our framework in a simulated smart grid network. Our goal is to allow the implementation of event stream composition scenarios in simulated smart grid networks using NETAH. A scenario is defined by a user, and includes:

- a set of producers with their associated workload;
- a set of consumers with their associated workload;
- a set of subscriptions or event stream composition networks for each consumer;
- a smart grid topology.

The adoption of NETAH in a simulated smart grid network is summarized in Figure 5.11. The simulation platform provides support for running user defined scenarios. It is implemented

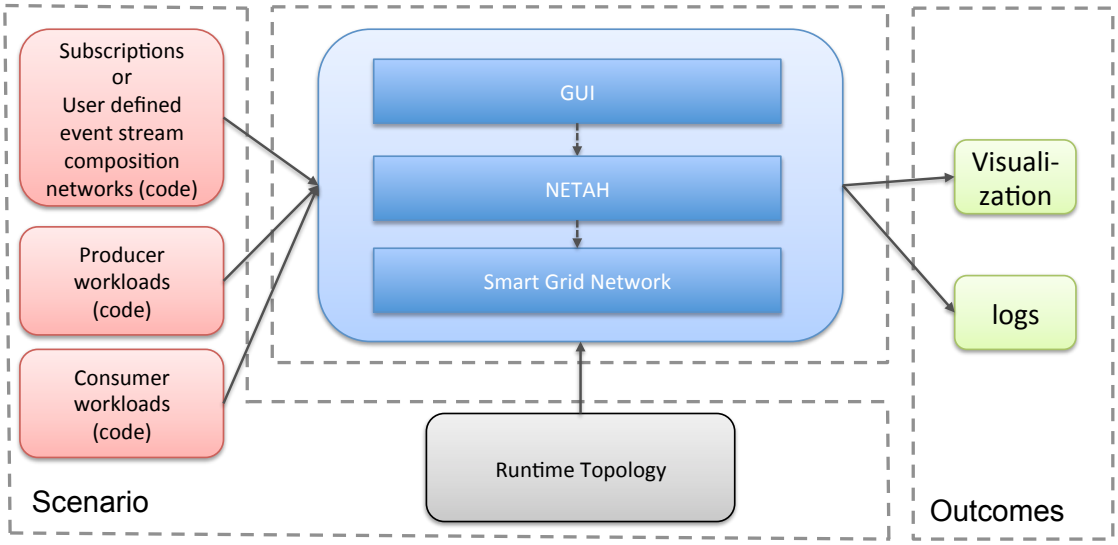

Figure 5.11 – NETAH for event stream composition in a simulated smart grid network

using the Java language, and is organized into two main modules called *Smart Grid Network* and *GUI*. The *Smart grid network* module implements the physical smart grid network, and the *GUI* module provides graphical user interfaces allowing to set, execute and visualize an event stream composition scenario.

The simulation platform is provided as input a specification of an event stream composition scenario.
As output, the simulation platform displays graphical user interfaces allowing to configure, execute and visualize the scenario. The simulation platform also produces logs which can be user defined (logs defined by producer code or consumer code) or logs generated by the platform itself (statistics of network links for example).

5.4.1 Defining an event stream composition scenario

The simulation platform allows to setup a scenario for event stream composition in the smart grid. Such a scenario is implemented by defining a set of producers, a set of consumers with their associated subscriptions and a smart grid nerwork as the runtime environment of the scenario.

Defining a custom producer

The class *EventProducer* (see Figure 5.12) is the base class for defining a producer. It defines a method *publish* used for publishing event instances of the particular event type. An event type is defined using a java class which conforms to the JavaBeans specification [Sun16]. A concrete producer is defined by extending the EventProducer class and implementing the logic of the concrete producer within the *run* method, as shown at Listing 5.1.

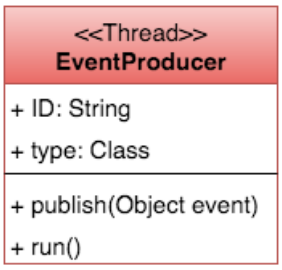

Figure 5.12 – Base class for defining a producer

```
public class AConcreteProducer extends EventProducer {
    AConcreteProducer (String ID, Class type){
        super(ID, type);
    }

    public void run(){
        // Custom producer code:
        // Create and publish event objects using publish(obj)
    }
}
```

Listing 5.1 – Defining a custom producer

When deployed on a device, an event producer class use the publish/subscribe broker of that device to convey the published events to interested parties[4]. The *publish* method of the *EventProducer* class publishes events in a topic named *<typeName>@<ID>* where *<typeName>* is the name of the class representing the event type and *<ID>* is the identifier of the producer. For example, an event producer identified as *meter1* for which the event type is given by a class named *MeterMeasure* will publish events [5] in a topic named *MeterMeasure@meter1*.

Defining a custom consumer

The class *EventConsumer* (see Figure 5.13) is the base class for defining a consumer. It contains an abstract method named *notify*, which is called each time events are notified to the consumer. A concrete consumer is defined by extending the *EventConsumer* class and implementing its

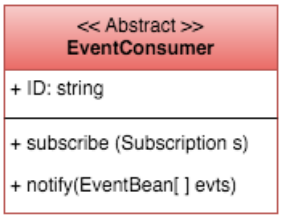

Figure 5.13 – Base class for defining a consumer

[4] This can be event processing units or consumers
[5] objects of the type MeterMeasure

notify method with the consumer specific reaction, as shown at Listing 5.2.

```
public class AConcreteConsumer extends EventConsumer {
    public void notify(Object[] evts){
    // Consumer specific reaction: show events on the console
     for(EventBean e: evts){
        System.out.println(e);
     }
}
```

Listing 5.2 – Defining a custom consumer

Defining a subscription

The class *Subscription* (see Figure 5.14)serves as the base class for defining a subscription. It contains a directed acyclic graph which represents the event stream composition expression. The node of the graph can be a event stream (which can be bounded) or a stream operator. The method named *addVertex* allows to add stream operators and event stream to the graph. The method named *addEdge* allows to add a connection between two vertices in the graph.

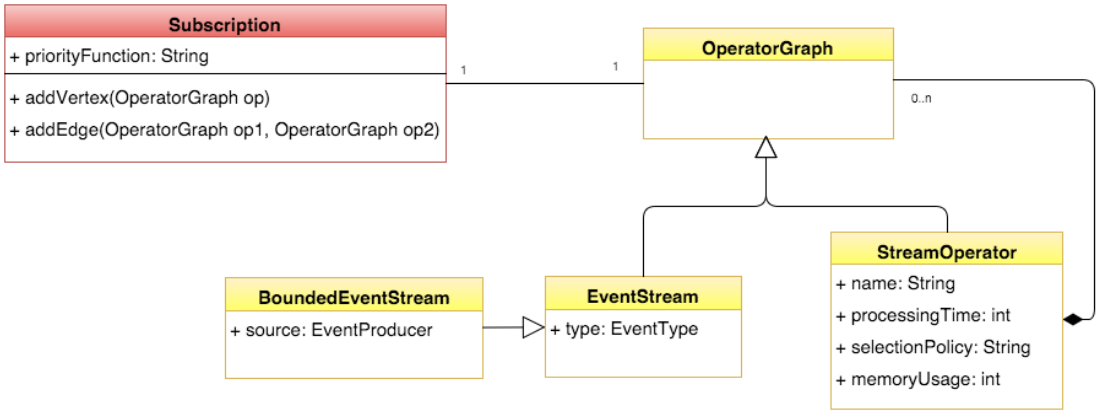

Figure 5.14 – Base class for defining a Subscription

For example, let us consider the subscription *s* created by the code at Listing 5.3. The directed acyclic graph associated to *s* is depicted at Figure 5.15.

```
// assume we have two producers P1 and P2
AConcreteProducer P1 = new AConcreteProducer("P1", MeterMeasure.class);
AConcreteProducer P2 = new AConcreteProducer("P2", MeterMeasure.class);
// Create a subscription
Subscription s = new Subscription();
s.setPriorityFunction(PriorityFunction.MAX);
// creation of operators and input streams
// the input streams
BoundedEventStream b1= new BoundedEventStream(P1);
BoundedEventStream b2= new BoundedEventStream(P2);
// The disjunction operator
Disjunction or = new Disjunction("or");
or.setProcessingTime(5);
or.setMemoryUsage(10);
or.setSelectionPolicy(SelectionPolicy.PRIORITY);
```

```
16 // The filter operator
17 Filter f = new Filter("filter");
18 f.addPredicate(new GreatherThan("realPower", 2));
19 f.setProcessingTime(5);
20 f.setMemoryUsage(10);
21 f.setSelectionPolicy(SelectionPolicy.PRIORITY);
22 // The aggregate operator
23 Aggregate avg = new Aggregate("aggr");
24 avg.setProcessingTime(50);
25 avg.setMemoryUsage(10);
26 avg.setSelectionPolicy(SelectionPolicy.PRIORITY);
27 avg.setType(Aggregate.AVG);
28 avg.aggregateOn("realPower", "avgP");
29 avg.setWindow(new TimeBatchWindow(10, TimeUnit.SECONDS));
30 // building the directed acyclic graph
31 s.addVertex(b1);
32 s.addVertex(b2);
33 s.addVertex(or);
34 s.addVertex(f);
35 s.addVertex(avg);
36 s.addEdge(b1, or);
37 s.addEdge(b2, or);
38 s.addEdge(or, f);
39 s.addEdge(f, avg);
```

Listing 5.3 – Defining a subscription

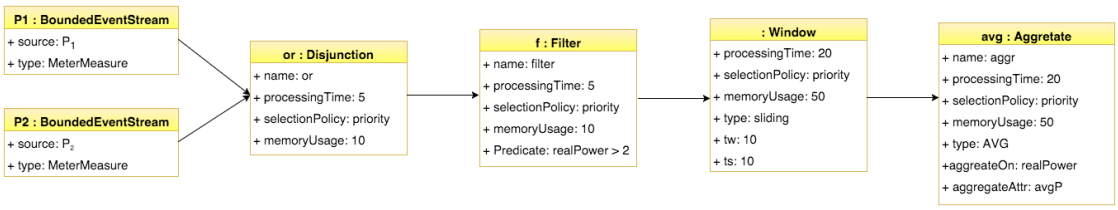

Figure 5.15 – Directed acyclic graph associated to the subcription created by Listing 5.3

Implementing a scenario

In order to simulate a scenario, the user should:

- create producer instances, consumer instances with their associated subscriptions;
- create the smart grid network on which the scenario should be executed.

The class named *Simulation* (see Figure 5.16) is an API for the creation and the execution of a scenario.

The methods *addProducer* and *addConsumer* allow to add producers instances and consumer instances respectively. The methods *addDevice* and *addComLink* allow to build network topology from the code. We mention that network topology can also be build via the graphical user interface, using the mouse. The method *run* creates the event stream composition network associated to each subscription using NETAH, and launches the simulation. This

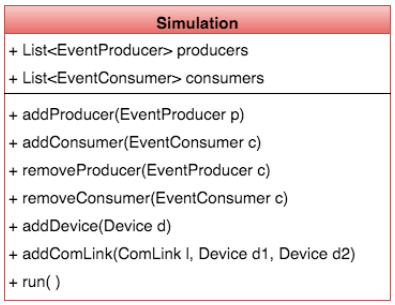

Figure 5.16 – The class *Simulation*

displayssimulator the simulator graphical user interface (see Section 5.4.2) on which the user can :

- create a network topology if this has not be done via the code
- assign producers and consumers to smart grid devices
- manually or automatically[6] deploy event processing units within the runtime environment
- execute the simulation

Listing 5.4 shows how to setup a simulation using the previously created producers, consumers and subscription.

```
Simulation simu = new Simulation();
// adding producers instances
simu.addProducer(P1);
simu.addProducer(P2);
// adding a consumer instance and its subscription
AConcreteConsumer c = new AConcreteConsumer("c");
c.setSubscription(s); // subscription s is created at Listing 5.3
simu.addConsumer(c);
// starting the simulation
simu.run();
```

Listing 5.4 – Setting and starting a simulation

5.4.2 The graphical user interface (GUI)

We implemented the *GUI* using the Java swing API [BRJ+02] in conjunction with the JUNG library [JUN10] for graph manipumation and rendering. The graphical user interface of the simulator (see Figure 5.17) allows to:

- manage network topologies, which includes:

[6] using NETAH

- create and configure a new network topology
- save or load an existing network topology

- display the event stream composition networks created in a scenario
- run the simulation

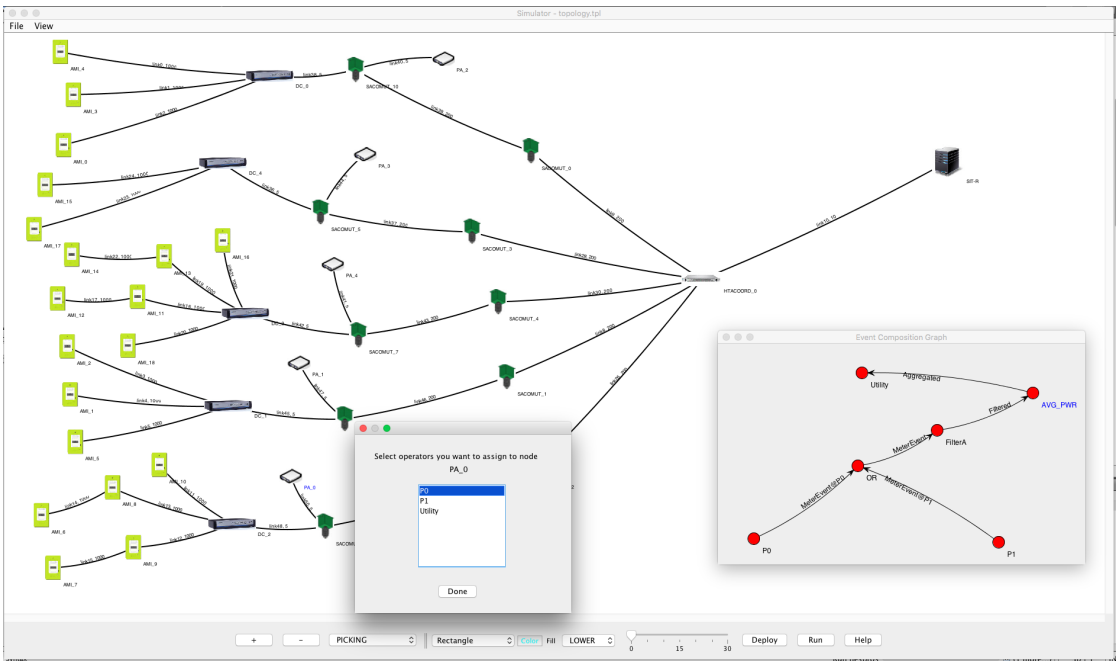

Figure 5.17 – The simulator graphical user interface

5.5 Conclusion

In this chapter, we presented the use of NETAH for building and deploying event stream composition networks in smart grids. First, we presented an overview of a smart grid, focusing on the smart grid requirements in term of event stream composition. Then, we presented event stream composition, as well as an EPU mapping for smart grids. Given that we cannot operate on a real smart grid network, we proposed a simulation of the smart grid network. Finally, we implemented our approach within simulator that allows to define and run event composition scenarios on a simulated smart grid.
In the next chapter, we will present a real smart grid scenario and its implementation with NETAH.

6 NETAH for Location of Resistive Failures

This chapter demonstrates the validity of our approach for event stream composition, in the context of a realistic use case, namely the location of resistive[1] failures in an energy network. This use case has been proposed by electrical energy experts within the SOGRID project [SOG16, ENLC+ 15, ENLC+ 16]. This chapter is organized as follows: Section 6.1 gives a brief overview of the energy supply chain. Section 6.2 presents the current practice of fault detection and location in energy networks, and the approach for resistive fault location proposed in the SOGRID project. Section 6.3 presents the implementation of such a scenario, using the NETAH framework for smart grid. This includes the implementation of the producers, consumers and event stream composition networks associated to this scenario. Section 6.4 presents the results of our experimentation, and Section 6.5 concludes this chapter.

Contents

6.1 Overview of the energy supply chain **98**
6.2 Detection and location of resistive faults **100**
6.3 Implementation **101**
6.4 Experimental results **115**
6.5 Conclusion **115**

[1] See Section 6.2.2

6.1 Overview of the energy supply chain

The general flow of the energy over the electricity network is summarized in Figure 6.1. The electricity produced at a source station is transported over long distances via medium-voltage lines (HTA). Then, the electricity is transported to local HTA/BT substations where they are being transformed into low voltage electricity (BT) before being distributed to consumers.

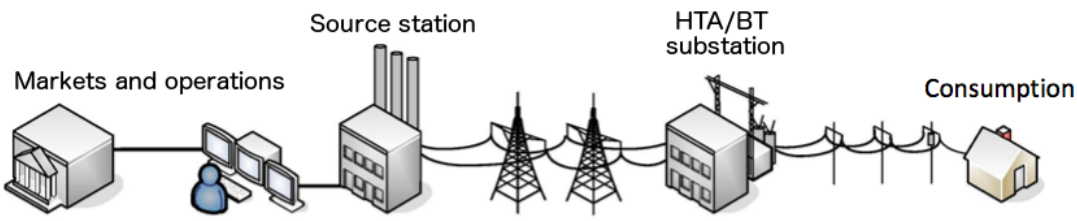

Figure 6.1 – Simplified view of a power grid

6.1.1 Source station

A source station tranforms the electricity produced at a high voltage (HTB, more than 50 KV) into a medium voltage electricity (HTA, 20 KV). Figure 6.2 shows the main components of a source station.
A source station includes a transformer which transforms the electricity high voltage electricity into medium voltage electricity. The medium voltage electricity is transported to HTA/BT substations before being delivered to consumers. A source station can feed many HTA/BT substations. A source station is equipped with protective devices such as protection relays and circuit breakers which observe the departure of the energy at the source station. The intelligent mechanisms of protection relay allows to detect the existence of abnormal situations on an HTA line. For example, a current or voltage which exceeds a threshold. The protection relay can order the opening or closure of cicuit breakers in response to the detected situation.
In the smart grid architecture considered in the *SOGRID* project, a source station also includes a device named *HTA coordinator*, which allows to communicate with other equipments (servers at the control center, other telecom devices on the grid).

6.1.2 HTA/BT substation

A HTA/BT substation transforms medium voltage (HTA) electricity into low voltage (BT) electricity and delivers it to consumers. Figure 6.3 shows the main components of a HTA/BT substation.
In the smart grid architecture considered in *SOGRID* project, a HTA/BT substation includes a coupler/sensor device named *SA-COMUT*, which embeds a sensor for gathering HTA measures

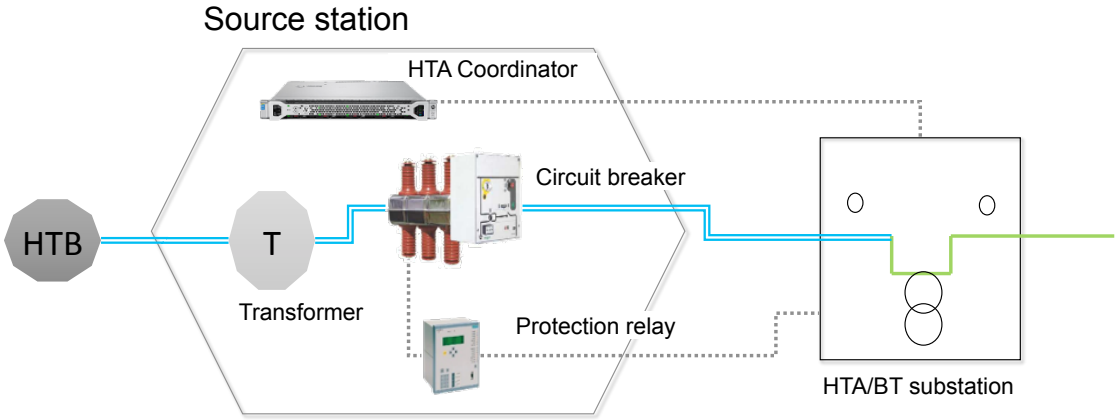

Figure 6.2 – A source station

(current, voltage, power, etc) and a coupler allowing to communicate over HTA lines[2]. A HTA/BT substation also includes a data concentrator (DC) which is connected to a set of smart meters located at consumer residences.

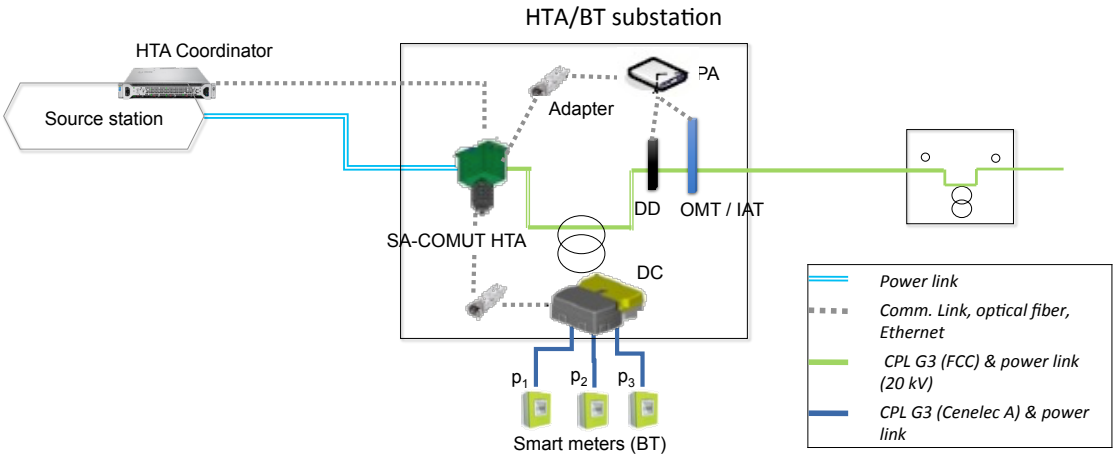

Figure 6.3 – A HTA/BT substation

In order to monitor the transmission network, some HTA/BT substation are also equipped with:

- actuator which allows to open and close a HTA line. It is refered to as *OMT* (a french abbreviation of "organe de manoeuvre télécommandé").
- a slave station (termed in french "poste asservi" or PA) which allows to control and query an OMT state (open/closed).

[2]The *SOGRID* network is based on the third generation of power line communication technology, which allows data to be transported over electrical lines

- default detectors (DD) which allows to indicate the presence of a fault. A Default detector can be communicating or luminous. A communicating default detector reports a failure to a remote control center, whereas a luminous default detector indicates a failure using a light signal.

6.2 Detection and location of resistive faults

6.2.1 The general practice of fault detection and location

The electrical network is subject to failures that can be caused by natural factors such as aging materials, tree contact, birds, or by abnormal electrotechnical conditions in the network such as an exceeded current or voltage threshold. Failures can result in outage and therefore they must be located as soon as possible.
The protection relay of the source station is responsible for detecting faults on the HTA network, with the identification of the HTA departure at the origin of the fault. The principle of fault detection on electrical network relies on the crossing of a predefined threshold of an electrical value. The threshold can be defined for the current, the voltage, or for their derivatives. The detected fault is notified to the network manager, who has to find out the fault location. The usual process of fault location consists in reconnecting the HTA departure at the origin of the fault, and each segment of the HTA network behind that departure. The network reconnection maneuverings (via telecommand or manual) exploit the information provided by default detectors (via telesignalisations or the color code of light signals) located on the HTA network in order to deduce suspicious segment on the HTA network.
When the segment in fault is connected, the fault reappears, and the segment in fault is located (see Figure 6.4). It is then isolated in order to investigate the root of the fault.

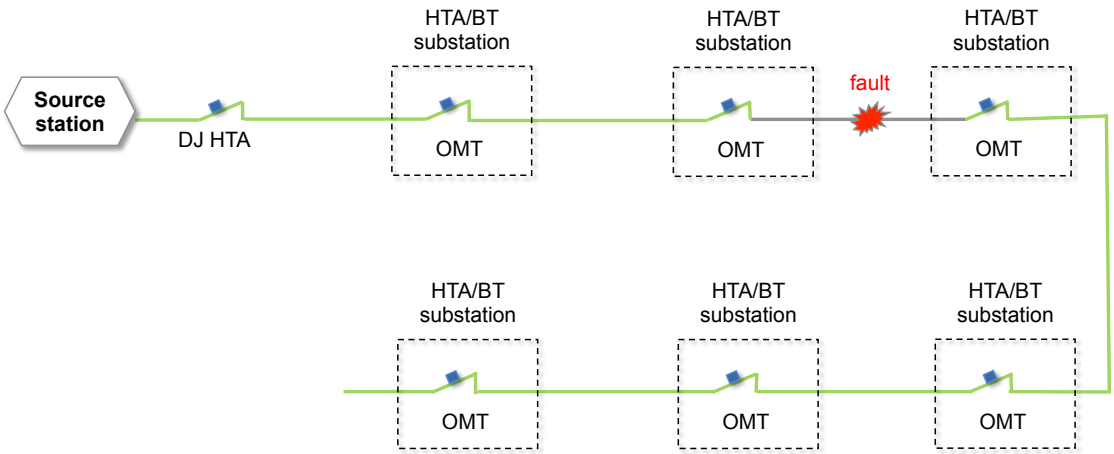

Figure 6.4 – Fault detection on a HTA network

6.2.2 Resistive fault

A resistive fault is a particular case of fault characterized by a bad current that is not high enough to be detected by the fault detectors located on the HTA network. While resistive faults are fairly infrequent (10% of to total amount of faults), their location using the classical fault location approach is extremely difficult and time consuming. This is due to the fact that when the faulty segment is reconnected, the fault is not detected since the generated faulty current is not high enough to be detected. In consequence, the network operator will wrongly designate another segment of the network as the faulty segment when its reconnection will make the fault reappear. Since the designated segment is not the faulty segment, the network operator has to find the fault by looking at other segments, which is generally time consuming. In order to reduce the location time of resistive faults, an approach proposed in the *SOGRID* project consists in correlating resistive faults in the medium voltage (HTA) network with observations of voltage imbalances in the low voltage (BT) network. The idea is to suggest locations that have to be investigated in priority, because they present a voltage imbalance which appears in a correlated way with respect to the resistive fault.

The voltage imbalance on a HTA/BT substation is characterized by an overtaking of the reverse voltage threshold at that substation. If the HTA/BT substation is downstream a failure on the medium voltage network, then the reverse voltage is systematically higher than the 5% of the nominal voltage (230 V).

The slave station (PA) located in each HTA/BT substation (see Figure 6.3) periodically computes the value of the inverse voltage. Values that are higher than 11.5 V indicate voltage imbalances on the HTA/BT substation which have to be correlated with resistive faults. A resistive fault and a voltage imbalance appearing within a 20 seconds time window are considered as relevant for the location of a resistive fault. In fact, when a resistive fault occurs, the communication[3] is still possible on the faulty line for the next 20 seconds. After that time, the line is disconnected by the protection relay, and the voltage imbalance cannot be notified.

Coupling a resistive fault and a voltage imbalance within a 20 seconds time window is a complex event named "*complex fault*" that must be notified to the network manager in order to inform:

1. the appearance of a resistive failure on the medium voltage network, and
2. the parameters related to the failure and to the voltage imbalance (e.g., concerned equipments, values of registered voltage measures, etc.) that must allow to identify the fault area.

6.3 Implementation

In this section, we present the implementation of the approach for resistive fault detection and location, using the NETAH framework for smart grid (see Chapter 5). First, we present

[3] Here, the power line communication

the main elements of the event stream composition approach, which are the event stream types, the event producers, the event consumers and their subscriptions. Then, we present the implementation of the approach on the simulation platform (See Section 5.4), and the experimental results.

6.3.1 Event stream compositions

Simple event types

The simple event types involved in resistive fault detection and location are:

- the $RVoltage : \langle voltage : double \rangle$ event type[4], which corresponds to a measure of the reverse voltage at a low voltage substation. The *voltage* attribute represents the value of the measured reverse voltage. In the *SOGRID* project, the priority level associated to $RVoltage$ event instances is 3.
- the $ResistiveFault : \langle lineID : String \rangle$ event type, which corresponds to a resistive fault on a medium voltage line. The *lineID* attribute represents the HTA line on which the default is detected. In the *SOGRID* project, the priority level associated to $ResistiveFault$ event instances is 1.

Complex event types

The complex event types involved in resistive fault detection and location are:

- the $UVoltage$ event type, which corresponds to a voltage imbalance on a HTA/BT substation. It is produced when a reverse voltage measure exceeds it threshold (11.5 V). In the *SOGRID* project, the priority level associated to $UVoltage$ event instances is 1.
- the *RFaultLocation* event type, which corresponds to the occurrence of a voltage imbalance (a *UVoltage* event instance) and a resistive fault (a *ResistiveFault* event instance) within a 20 seconds time window. The priority level associated to *ResistiveFault* event instances is 1.

Producers

The event stream producers involved in the detection and location of resistive faults are:

- The slave station (PA), which produces an event stream of the type *RVoltage*. There is one slave station PA_i on each HTA/BT substation $htabt_i$, $i = 1..n$ and thus, there is one bounded event stream $Stream(RVoltage, PA_i)$ produced by each slave station PA_i.

[4] We omitted meta-attributes

- The data concentrator (DC), which produces an event stream of the type *UVoltage*. There is one data concentrator $DC_i,\ i = 1..n$ on each HTA/BT substation $htabt_i,\ i = 1..n$ and thus, there is one bounded event stream $Stream(UVoltage, DC_i)$ produced by each data concentrator DC_i.
- The HTA coordinator, which produces an event stream of the type *ResistiveFault*. There is one HTA coordinator on each source station. In this scenario, we consider only one source station, so one HTA coordinator named *HTACoord*.

Consumers

The event stream consumers involved in the detection and location of resistive faults are the data concentrator and the SIT-R information system (i.e, the network monitoring system).

Data concentrator (DC) The data concentrator, which is interested in receiving voltage imbalance notifications (*UVoltage* event instances). For that, each data concentrator DC_i issues a subscription s_i such that:

- the event stream composition expression is:
  $expression(s_i) = fi@filter_{voltage>11.5}$ *(Stream(RVoltage,* PA_i*)).*
- the QoS expression is $qos(s_i)$ = *{fi[memory:1 time:1, pFunction:1, sPolicy:continous]}*, which indicates that the memory and CPU time required by the filter operator are 1Mo and 1ms respectively, the priotity function is the constant function $f = 1$ and the selection policy is *continuous* (see Section 4.3.3).
  In our setting, the production rate of reverse voltage events by each slave station is 2 events per minutes. The processing time of the filter operator per event is less than 1ms. This implies that the filter operator needs only to store one event (approximately 3Ko) per processing cycle. Thus, its memory footprint is less than 1Mo.

The subscription issued by a DC is processed by NETAH, which generates the corresponding event stream composition network and deploys it within the smart grid environment, as shown at Figure 6.5.

SIT-R information system The SIT-R information system, which is interested in reveiving *RFaultLocation* event notifications. For that, the SIT-R information system issues a subscription sub such that:

- the event stream composition expression is:
  $a@AND\ (f1@flatten(w1@win : sliding_{(20sec,\ 20sec)}$ *(Stream(UVoltage))),*
  $f2@flatten(w2@win : sliding_{(20sec,\ 20sec)}$ *(Stream(ResistiveFault))).*
  which is rewritten in developped form as:
  $a@AND\ (\ f1@flatten(\ w1@win : sliding_{(20sec,\ 20sec)}$ (o@OR(Stream(UVoltage, DC_1),...,

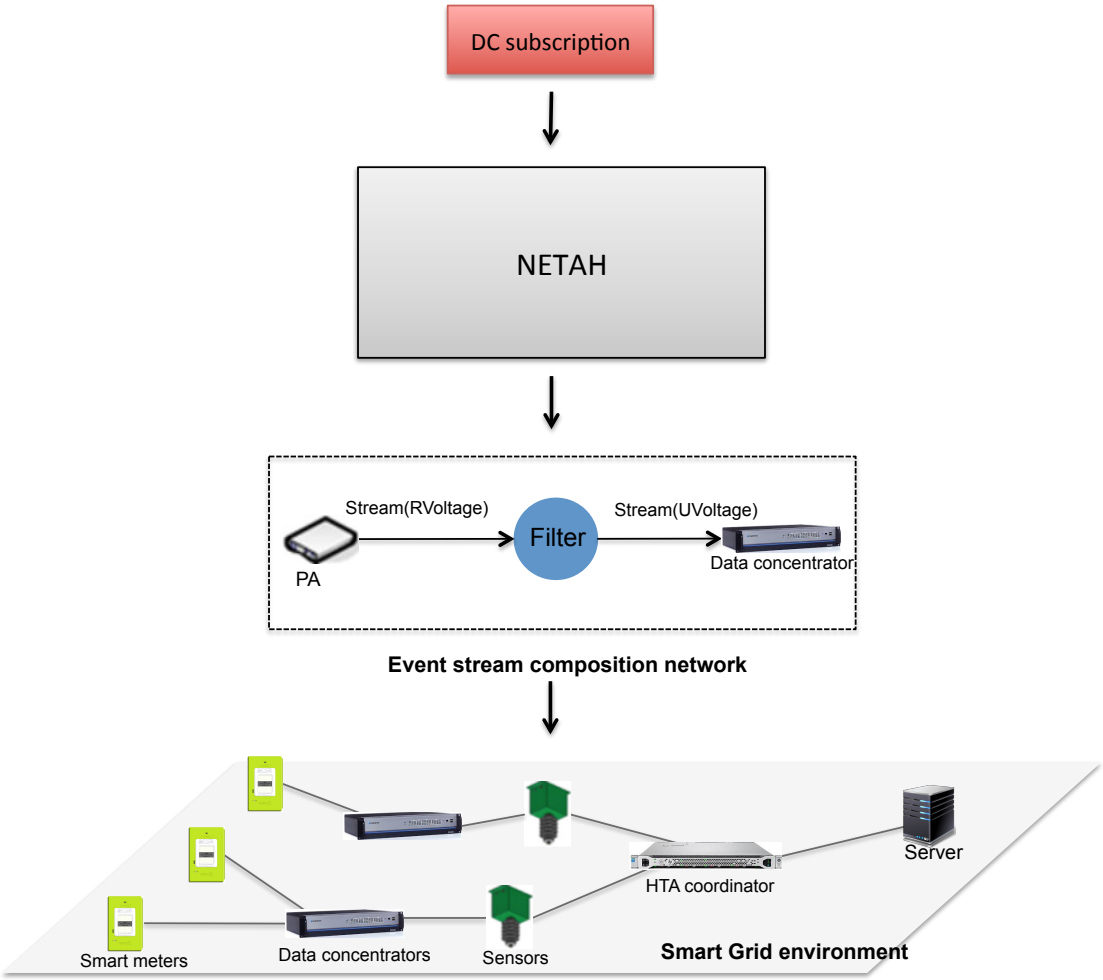

Figure 6.5 – Processing of a DC subscription

Stream(UVoltage, DC_5)))), $f2@flatten(\ w2@win:sliding_{(20sec,\ 20sec)}(\ Stream\ (\ ResistiveFault,\ HTACoord)))$.

- the QoS parameters associated to the expression are presented in Table 6.1.

  - *Memory*. In our setting, resistive faults and voltage imbalance events are produced sporadically, but with a maximum production rate of 2 events per minute for per producer. So, in the worst case, the disjunction operator (identified by *o*) will store 10 reverse voltage alarms[5]. In our setting, an event takes approximately 3Ko memory. Thus, the memory footprint of the disjunction operator is less than 1Mo. Furthermore, in the worst case, the window operators *w1* and *w2* will store 2 and 10 events respectively, which also requires less than 1Mo of memory. The flatten operators *f1* and *f2* process events from *w1* and *w2* respectively, so they also require less than 1Mo memory. The conjunction operator (identified by *a*) process events from flatten operators, that is 12 events in the worst case. So the memory footprint of the conjunction operator is also

[5]They are 5 data concentrators which produce voltage imbalance events.

less than 1Mo.

- *Processing time.* Except the window operators *w1* and *w2* which requires 20000ms of processing time, we set the processing time of remaining operators as 1ms since the number of events to be processed is reduced, and those operators have a linear time complexity.

- *Selection policy.* We specified the selection policy as *continous* in order to ensure that all events will be processed in case where there are many concurrent event instances at the input of an operator.

- *Priority function.* The priotity level associated to *RFaultLocation* event instances is 1. Setting the priotity function of the conjunction operator as the constant function $f = 1$ is enough to produce the expected behavior. The priority function of other operators has been set for consistency.

		QoS Parameters			
		memory (Mo)	**time (ms)**	**sPolicy**	**pFunction**
Operators	**w1**	1	20000	continuous	1
	f1	1	1	continuous	1
	o	1	1	continuous	1
	w2	1	20000	continuous	1
	f2	1	1	continuous	1
	a	1	1	continuous	1

Table 6.1 – QoS parameters of the subscription

The subscription issued by SIT-R is processed by NETAH, which generates the corresponding event stream composition network and deploys it within the smart grid environment, as shown at Figure 6.6.

6.3.2 Simulating a resistive fault detection and location scenario

This subsection presents the implementation of the event stream composition defined for resistive fault detection and location. For that purpose, we define a scenario on NETAH, in which we implement the event stream producers, consumers and subscriptions involved in the detection and location of resistive faults within a smart grid topology.

Implementing event producers

The slave station A slave station produces an event stream of the type $RVoltage : \langle voltage : double \rangle$. Our approach for simulating its event stream production logic is described as follow: The slave station is associated to an input file containing the event data. This file contains one voltage value per line. These values are read sequentially for creating event instances of the type *RVoltage*. Each voltage value corresponds to the value of the *voltage* attribute of an *RVoltage* event instance. There is a time delay between two consecutive event creations.

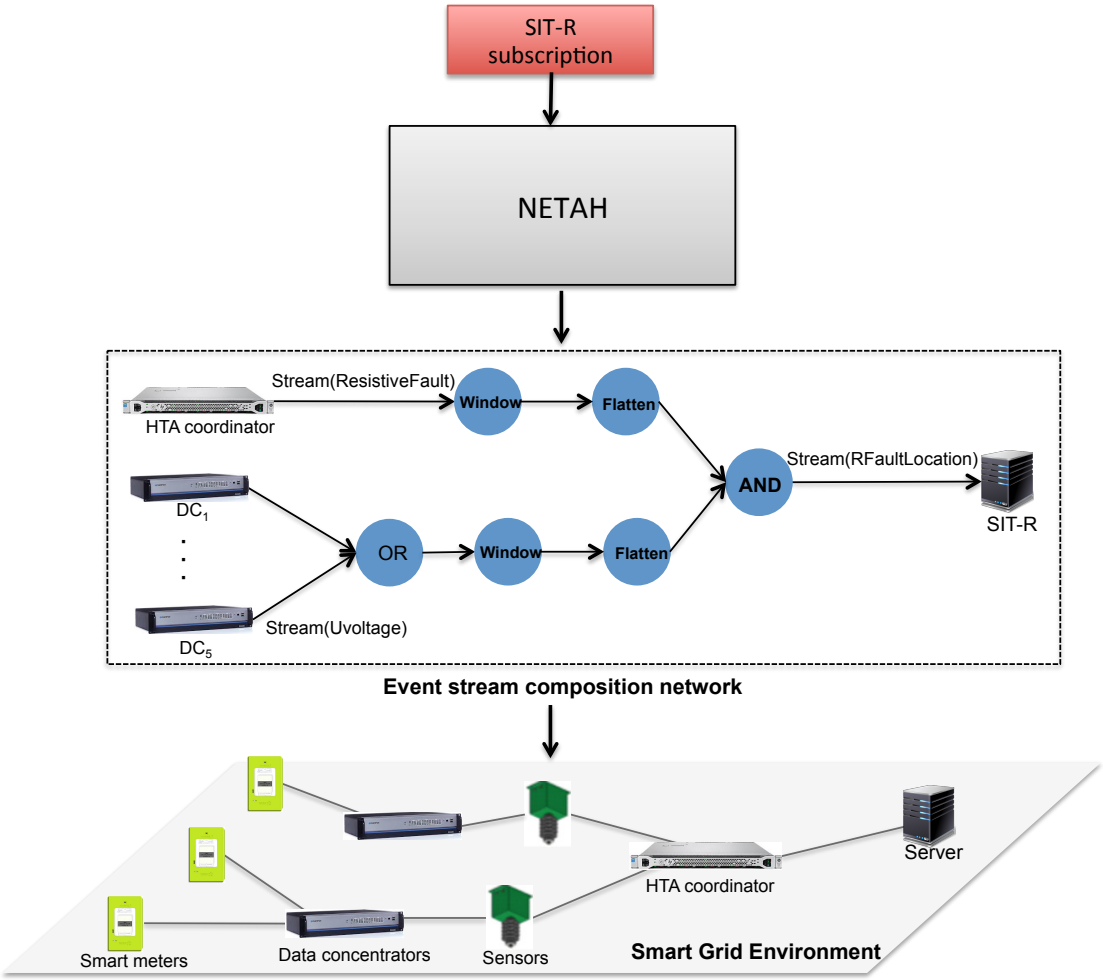

Figure 6.6 – Processing of the SIT-R subscription

For example, in the case of the input file depicted at Figure 6.7 containing five lines numbered from 1 to 5, five event instances e_1, e_2, e_3, e_4 and e_5 of type *RVoltage* will be produced such that $e_1.voltage = 8$, $e_2.voltage = 4.897$, $e_3.voltage = 6.2$, $e_4.voltage = 11.02$ and $e_5.voltage = 10.8$.

Listing 6.1 shows the code which implements an event stream producer of the type *RVoltage* associated to a slave station. Each line of the input file is read (line 14) and casted to a double value (line 16). Then, from line 17 to 22, an event instance is created and its attribute are set (voltage, priority, etc). The event is published (line 23) and the next line is processed after a defined delay (line 24), until the end of file.

```
1  public class SlaveStation extends EventProducer {
2      private long delay;
3      private File inputFile;
4
5      SlaveStation (String name, long delay, File inputFile){
6          super(name, RVoltage.class);
7          this.delay = delay;
8          this.inputFile = inputFile;
```

```
8
4.897
6.2
11.02
10.8
```

Figure 6.7 – Example of an input file for the slave station

```
    }

    public void run(){
        BufferedReader  input = new BufferedReader(new InputStreamReader(inputFile
          ));
        do{
            String line = input.readLine();
            if(line!=null){
               double voltage = Double.parseDouble(line);
                EventBean evt = new EventBean();
                evt.payload.put("voltage", voltage);
                evt.getHeader().setProducerID(name);
                evt.getHeader().setPriority(3);
                evt.getHeader().setDetectionTime(System.currentTimeMillis());
                evt.getHeader().setProductionTime(System.currentTimeMillis());
                publish(evt);
                sleep(delay);
            }
         } while(line!=null)
    }
}
```

Listing 6.1 – The event producer of a slave station

The data concentrator producer. The event stream produced by each data concentrator DC_i is the result of the event stream composition contained in its subscription, that is $filter_{voltage>11.5}(Stream(RVoltage, PA_i))$. Therefore, the data concentrator DC_i produces event instances of the type *UVoltage* each time it consumes an event from the output stream of the filter operator. A data concentrator is then associated with an event consumer *DC-Consumer* (defined later) which subscribes to the *UVoltage* event type and an event producer *DCProducer* which forwards the event stream consumed by the *DCConsumer*. Listing 6.2 presents the code of a *DCProducer*. A *DCProducer* has a queue (line 2) which contains the events consumed by the *DCConsumer*. The *DCProducer* retrieves those events and publishes them (lines 14 and 15).

```
1  public class DCProducer extends EventProducer {
2      private BlockingQueue<EventBean> queue;
3
4      DCProducer (String name){
5          super(name, UVoltage.class);
6          queue = new LinkedBlockingQueue();
7      }
8      public BlockingQueue<EventBean> getQueue() {
9          return queue;
10     }
```

```
11
12      public void run(){
13          while(true){
14              EventBean evt = queue.take();
15              publish(evt);
16          }
17  }
```

Listing 6.2 – The producer of a data concentrator

The HTA Coordinator producer. The HTA coordinator produces an event stream of the type $ResistiveFault : \langle lineID : String \rangle$. Similarly to slave stations, we simulate resistive faults on medium voltage lines (and the corresponding event stream) using input files. The HTA coordinator is associated to a set of CSV files. Each file corresponds to a medium voltage line connected to the HTA coordinator. Each line of the file indicates a line ID and a boolean value between "true" or "false". When a line <*lineID, boolean*> is read, if the boolean value equals "true", then a fault is declared on the medium voltage line identified by *lineID*. The lines are read sequentially, and there is a delay between two consecutive readings.

For example, in the case of the input file depicted at Figure 6.8 containing five lines numbered from 1 to 5, a *ResistiveFault* event instance *e* is produced after the reading of the line number 4, such that $e.lineID = link8$.

```
1  link8,false
2  link8,false
3  link8,false
4  link8,true
5  link8,false
```

Figure 6.8 – Example of an input file for the HTA coordinator

When a fault is declared on a HTA line, that line is declared as "down" (See Section 5.3) after 20 seconds.
In order to simulated faults over all its connected lines, the HTA coordinator reads its input files in parallel. In order to simplify the simulation of such a process, we define a producer which reads one input file, and thus simulating resistive faults on one medium voltage line. By instantiating as many producers as there are medium lines connected to the hta coordinator, and deploying them all on the HTA coordinator, we simulate faults on all the medium voltage lines.
Listing 6.3 shows the code which implements an event stream producer of the type *ResistiveFault* associated to a medium voltage line. Each line of the input file is parsed and the id of the HTA line and a boolean value are retrieved (lines 16 and 17). In case the boolean indicates a HTA fault (its value is true), then, an event instance is created and its attribute values (lineID, priority, etc.) are set (lines from 19 to 24). Then, the event is published (line 25) and the next line is processed after a defined delay (line 28), until the end of line.

```
1  public class ResistiveFaultProducer extends EventProducer {
2      private long delay;
```

```
3       private File inputFile;
4
5       SlaveStation (String name, long delay, File inputFile){
6       super(name, RVoltage.class);
7       this.delay = delay;
8       this.inputFile = inputFile;
9       }
10
11      public void run(){
12          BufferedReader  input = new BufferedReader(new InputStreamReader(inputFile
              ));
13          do{
14              String line = input.readLine();
15              if(line!=null){
16                  String[] data = line.split(",");
17                  boolean fault = Boolean.parseBoolean(data[1]);
18                  if(fault){
19                      EventBean evt = new EventBean();
20                      evt.payload.put("lineID", data[0]);
21                      evt.getHeader().setProducerID(name);
22                      evt.getHeader().setPriority(1);
23                      evt.getHeader().setDetectionTime(System.currentTimeMillis());
24                      evt.getHeader().setProductionTime(System.currentTimeMillis();

25                      publish(evt);
26                  }
27              }
28              sleep(delay);
29          } while(line!=null)
30      }
31  }
```

Listing 6.3 – The resistive fault event producer of a medium voltage line

Implementing event consumers

The data concentrator consumer. Listing 6.4 presents the code of the consumer on a data concentrator. A *DCConsumer* component in the data concentrator DC_i receives *RVoltage* event instances from the slave station PA_i for which the value of voltage attribute is greather than its threshold (11.5V) (line 8). It inserts those events into the event queue of the*DCProducer* component (line 10), which will publish them.

```
1   public class DCConsumer extends EventConsumer {
2       DCProducer dcProducer;
3
4       public(DCProducer dcProducer){
5           this.dcProducer = dcProducer;
6       }
7
8       public void notify(Event[] evts){
9           for(EventBean evt: evts){
10              dcProducer.getQueue().put(evt);
11          }
12      }
13  }
```

Listing 6.4 – The consumer of a data concentrator

The information system (SIT-R) consumer. Listing 6.5 presents the code of the consumer on the information system SIT-R. It receives notifications of *RFaultLocation* events via its *notify* method (line 9). Then, it retrieves the data associated to each event (lines from 15 to 26). That is:

- the medium voltage line in on which the resistive fault occurs
- the value of the reverse voltage
- the location of the voltage imbalance is given by the name of the device which measured the reverve voltage

Then, its displays a user friendly message dialog for signaling the complex event (line 27), and logs the retrieved data (line 28).

```
1  public class ISConsumer extends EventConsumer {
2      LoggerUtil logger;
3      String fileName ="complex_faults.txt";
4
5      public ISConsumer(){
6          logger = new LoggerUtil(fileName);
7      }
8
9      public void notify(EventBean[] evts){
10         for(EventBean evt: evts){
11             String htaline, deviceID;
12             double voltage;
13             long latency; // the event notification latency.
14
15             long latency = evt.getHeader().getReceptionTime() - evt.getHeader().
                 getDetectionTime();
16             EventBean[] data = (EventBean[]) evt.getValue("data");
17             if(data[0].payload.containsKey("depart")){
18                 htaline = (String) data[0].getValue("depart");
19                 deviceID = (String) data[1].getValue("deviceID");
20                 voltage = (double) data[1].getValue("voltage");
21             }
22             else{
23                 depart = (String) data[1].getValue("depart");
24                 deviceID = (String) data[0].getValue("deviceID");
25                 voltage = (double) data[0].getValue("voltage");
26             }
27             JOptionPane.showMessageDialog(null, "Resistive Fault at "+depart+"
                 whith voltage imbalance ("+voltage+"V) at "+deviceID);
28              logger.log(htaline+", "+ voltage+", "+deviceID+", "+latency);
29         }
30     }
31 }
```

Listing 6.5 – The consumer of the information system

6.3.3 Defining the smart grid topology

The adopted topology is given in Figure 6.9. It is a subset of the smart grid topology adopted in the *SOGRID* project. It consists in one source station represented by the HTA coordinator

named "HTACOORD_0", which is connected to five HTA/BT substations via medium voltage lines. Each HTA/BT substation $htabt_i$, $i = 0..4$ is represented by a slave station PA_i, a data concentrator DC_i and a sensor $SACOMUT_i$. Each data concentrator is connected to a set of smart meters. The device named *server* represents a server on the control center which hosts the information system of the utility (SIT-R).

The characteristics of those devices are presented in Table 6.2. Those values are derived from the characteristics of smart grid devices in the *SOGRID* project.

The slave stations which cannot host event processing units are given a zero memory capacity.

Table 6.2 – Resources availability on smart grid devices

Device type	Memory	CPU coefficient
Smart meter	128	1/4
Data Concentrator	512	1
Slave station	0	1/4
Sensor	1024	1
HTA coordinator	16384	4
Server	16384	8

6.3.4 Running the scenario

In order to launch the scenario, we instanciated the event producers and the event consumers with their subscriptions. More precisely, we instantiated:

- Five producers of the type *SlaveStation* (see Listing 6.1). There is one producer per slave station.
- Five producers of the type *DCProducer* (see Listing 6.2). There is one producer per data concentrator.
- Five producers of the type *ResistiveFaultProducer* (see Listing 6.3). There are all located on the HTA coordinator.
- Five consumers of the type *DCConsumer* (see Listing 6.4). There is one consumer per data concentrator.
- One consumer of the type ISConsumer (see Listing 6.5), which is located on the control center server.

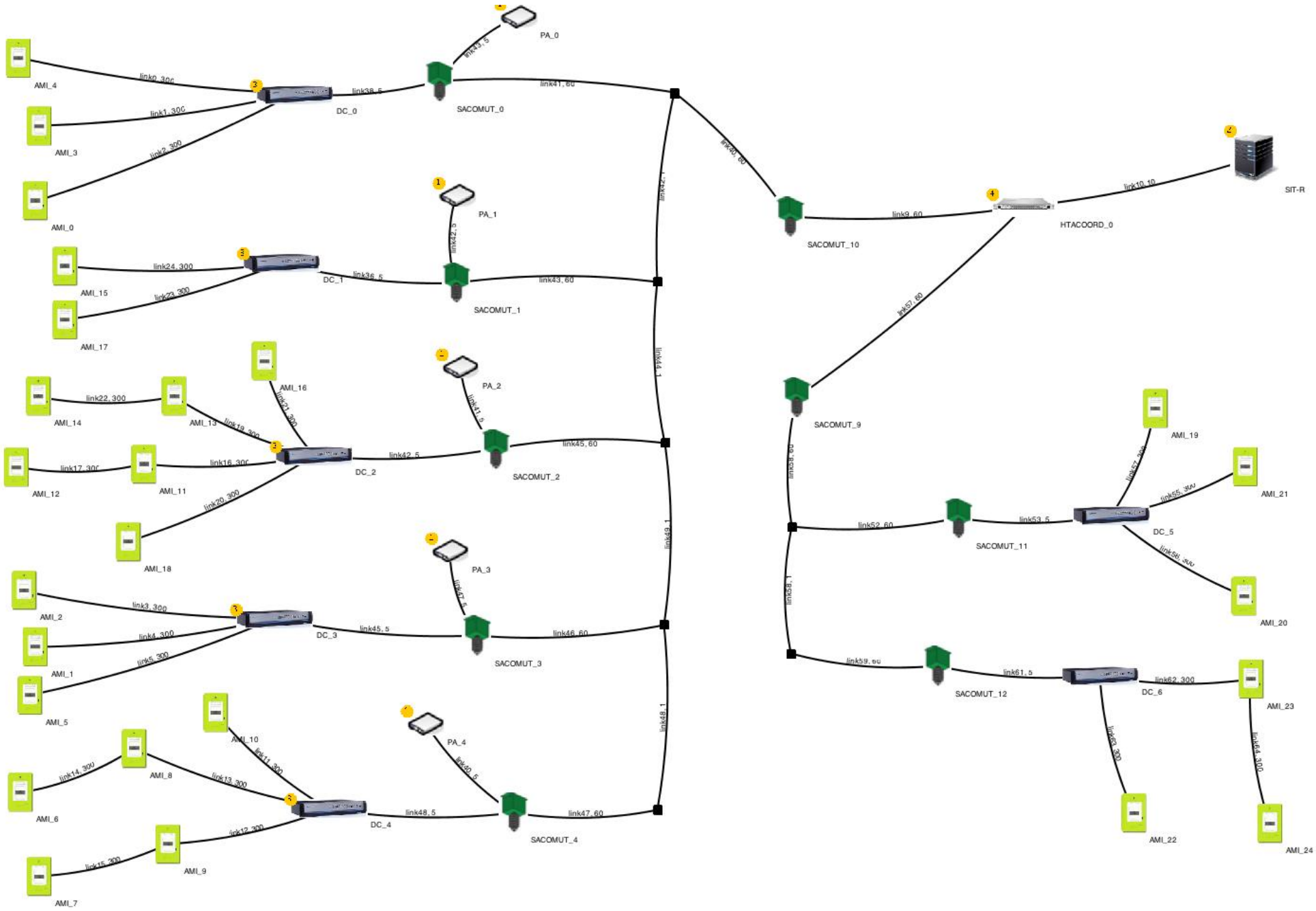

Figure 6.9 – The smart grid topology of the simulation

Then using NETAH, we instantiated the event stream composition network associated to each consumer subscription. The code at Listing 6.6 shows how to create the event stream composition network associated to the subscription of the data concentrator DC_0. The event stream composition network associated to the subscription of data concentrators DC_1 to DC_4 is created by a similar code.

```
// Create a subscription
Subscription s = new Subscription();
s.setPriorityFunction(1);
// Creation of operators and input streams
// The input stream
// Let us assume PA0 is the producer in the slave station PA0
BoundedEventStream b1= new BoundedEventStream(PA0);
// The filter operator
Filter f = new Filter("filter");
f.addPredicate(new GreatherThan("voltage", 11.5));
f.setProcessingTime(1);
f.setMemoryUsage(1);
f.setSelectionPolicy(SelectionPolicy.CONTINOUS);
// Building the directed acyclic graph
s.addVertex(b1);
s.addVertex(f);
s.addEdge(b1, f);
// Let us assume DC0 is the consumer in the data concentrator DC0
DC0.setSubscription(s);
```

Listing 6.6 – Defining a subscription for a consumer in a data concentrator

The code at Listing 6.7 shows how to create the subscription associated to the consumer on the control center server.

```
// Create a subscription
Subscription s = new Subscription();
s.setPriorityFunction(1);
// Creation of operators and input streams
// The input streams
// Let us assume P0..P4 are the producer in data concentrator DC0..DC4
BoundedEventStream b0= new BoundedEventStream(P0);
BoundedEventStream b1= new BoundedEventStream(P1);
BoundedEventStream b2= new BoundedEventStream(P2);
BoundedEventStream b3= new BoundedEventStream(P3);
BoundedEventStream b4= new BoundedEventStream(P4);
// Let us assume Pf0, Pf1 are the producers in the HTA coordinator
BoundedEventStream pf0= new BoundedEventStream(Pf0);
BoundedEventStream pf1= new BoundedEventStream(Pf1);
// operators
// the operator OR(b0,b1,b2,b3,b4)
Disjunction or1 = new Disjunction("or");
or.setProcessingTime(1);
or.setMemoryUsage(1);
or.setSelectionPolicy(SelectionPolicy.CONTINOUS);
// the operator OR(pf0,pf1);
Disjunction or2 = new Disjunction("or");
or.setProcessingTime(1);
or.setMemoryUsage(1);
or.setSelectionPolicy(SelectionPolicy.CONTINOUS);
// the operator and(or1, or2)
Conjunction and = new Conjunction("and");
and.setWindow(new TimeBatchWindow(20, TimeUnit.SECONDS));
and.setProcessingTime(1);
```

```
and.setMemoryUsage(1);
// Building the directed acyclic graph
// the vertices
s.addVertex(b0); s.addVertex(b1); s.addVertex(b2); s.addVertex(b3);
s.addVertex(b4); s.addVertex(pf0); s.addVertex(pf1); s.addVertex(pf2);
s.addVertex(pf3); s.addVertex(pf4); s.addVertex(or1); s.addVertex(or2);
s.addVertex(and);
// the edges for OR(b0,b1,b2,b3,b4)
s.addEdge(b0, or1); s.addEdge(b1, or1); s.addEdge(b2, or1);
s.addEdge(b3, or1); s.addEdge(b4, or1);
// the edges for OR(pf0,pf1)
s.addEdge(pf0, or2); s.addEdge(pf1, or2);
// the edge for and(or1, or2)
s.addEdge(or1, and); s.addEdge(or2, and);
// Let us assume SITR is the consumer in the control center server
SITR.setSubscription(s);
```

Listing 6.7 – Defining a subscription for a consumer in the control center server

Figure 6.10 shows the event stream composition network created by the defined subscriptions.

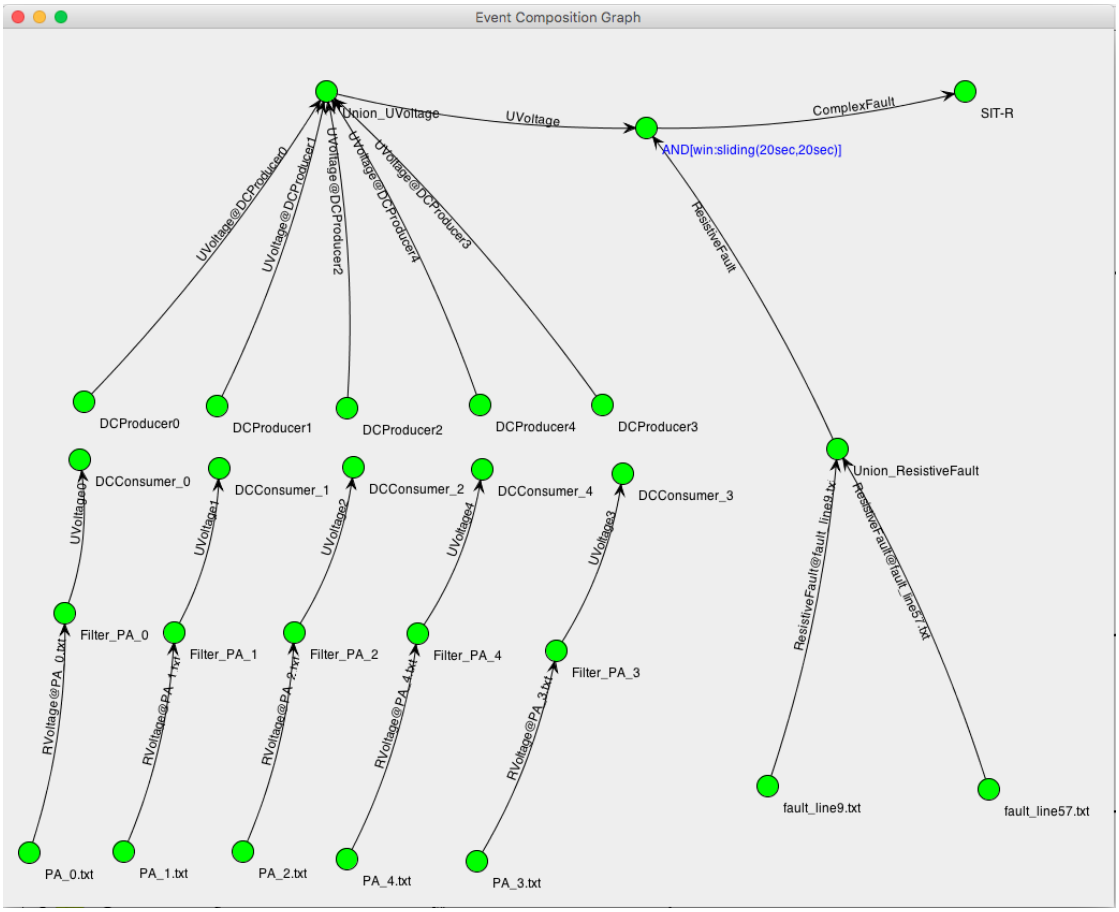

Figure 6.10 – Event stream composition networks for resistive faults detection and location

6.4 Experimental results

Our focus was on the EPU mapping decisions computed by NETAH, and the effectiveness of the deployed event stream composition networks in detection and location of resistive faults.

Using the EPU mapping algorithm proposed in Chapter 5, NETAH assigned the filter EPU generated by each DC subscription to the DC itself, which was one of the expected locations[6]. Moreover, using the same algorithm, NETAH assigned the EPUs generated by the SIT-R subscription to the SIT-R server itself. In fact, because of his highest computing power, the execution of the EPUs on the SIT-R server will minimize the processing latency of the EPUs.

In order to validate the effectiveness of the system in detection and location of resistive faults, we simulated a resistive fault on HTA lines, with some voltage imbalances on HTA/BT substations in order to detect correlation between them, via event stream composition.
Therefore, we simulated a resistive fault events e_1 and e_2 on the HTA departures *link9* and *link57* respectively (See Figure 6.9). We also simuated a voltage imbalance event e'_1 on the data concentrator DC_2, which happens at a timestamp which is close enough to the occurrence of e_1, that is $|e_1.productionTime - e'_1.productionTime| \leq 2sec$. This ensures that both events are correlated. Such a correlation have to be captured during the simulation in order to produce an instance of *RFaultLocation* event type. The event e_2, which is not correlated to a voltage imbalance should be ignored.

In Figure 6.11, the communication links colored in red (*link9* and *link57*) indicate the occurence of resistive faults at these lines, which are in the down state.
As expected, an *RFaultLocation* event instance is produced and notified to SIT-R, which presents the related information on a message dialog (see Figure 6.11) as implemented in Listing 6.5, line 27. The message indicates a correlation between the resistive fault on the HTA line *line9* and a voltage imbalance signaled on data concentrators DC_2, DC_3, DC_4 with the corresponding reverse voltage values. Using such an information, the network operator can suggest a good starting point for the location of the resistive fault. For example, the HTA/BT substation having the higher voltage imbalance (here, the substation hosting DC_2) can be checked in priority. The resistive fault that occurred at HTA line *link57* is not notified as expected.

6.5 Conclusion

This chapter presented a smart grid use case and its implementation using NETAH. The considered use case addressed the critical issue of resistive fault location in the electrical network, which is of particular interest for the industry. We showed how the resistive fault location issue can be modelled in terms of event stream composition, identifying event types, event stream producers, event stream consumers and subscriptions. We implemented the event stream composition network associated to each subscription in a simulated smart grid network. The results of the experimentation demonstrate the ability of NETAH to meet smart

[6]The other possible location was on the SACOMUT sensor.

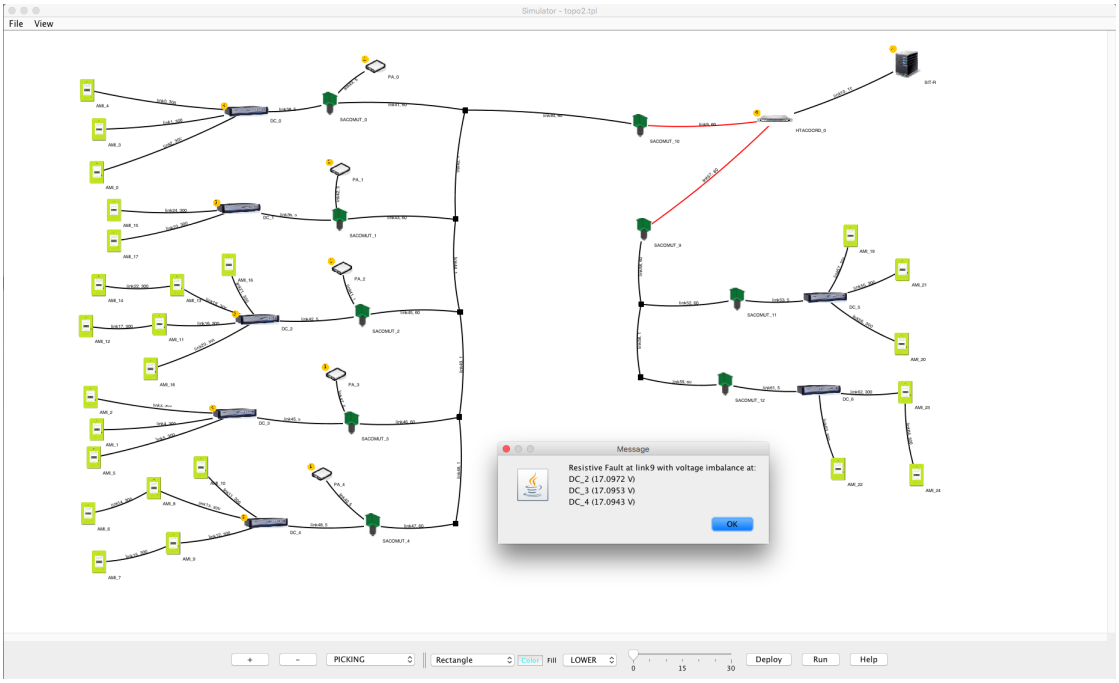

Figure 6.11 – Detection and location of a resistive fault

grids requirements in terms of event stream composition and notification.
However, the above experiment failed to demonstrate the scalibility of NETAH in terms of deployed EPUs, having only 8 EPUs deployed. A better experimentation would be on a use case involving a high number of EPUs. As shown in Appendix A, the EPU mapping algorithm proposed in Chapter 5 can handle up to 100 operators. From our own experience, it was difficult to have realistic use cases involving such a high number of EPUs.

7 Conclusions and perspectives

7.1 Summary

In this thesis, we have proposed an approach for event stream composition in highly distributed environments having limited computing resources like the IoT and smart grids. The efficiency of such a solution relies on the ability to access and process distributed event streams produced by different sources and notify the results to interested parties, considering both the limitations of the runtime environment and the application QoS requirements. Before these issues could be addressed, it was necessary to review the research areas of data stream processing systems and complex event processing systems, and to assess their applicability in large scale and resource-constrained environments. Centralized systems like [ABB+03, ACc+03, WDR06, Esp15] present scalability and single point of failure issues in large scale contexts. Clustered systems like [Aba05, SMMP09, STO13, ZCF+10, Apa16c, Apa16a] resolve the issues of centralized systems, but requires the event streams to be routed to the remote cluster for processing, which is not efficient using limited network connections. In addition, these solutions consider that there is enough computing resources on the processing nodes, an assumption which is not feasible in our context.

Our solution leverages the computing resources offered by the devices deployed within the environment to enable a large scale distributed event stream composition that deals with QoS.

Event stream composition modeling was the subject of Chapter 3. We proposed formalisms for representing different types of event streams. Then, we proposed a set of operators that operate on event streams, as well as how to combine them to obtain complex event stream composition expressions, which express complex queries. We also provided a way to describe the QoS requirements associated to such queries.

Chapter 4 presented NETAH, an event stream composition framework in distributed and QoS constrained runtime environments. NETAH considers as input a consumer subscription, which consists in an event stream composition expression and the associated QoS requirement. then, it generates an event stream composition network, which consists in a set of connected event processing units that implements stream operators. The event processing units communicate via a distributed publish/subscribe middleware. NETAH uses a mapping algorithm which assigns event processing units to distributed processing nodes. This

algorithm should consider the resources available on processing nodes, while minimizing the end to end latency of event notification. Using such an algorithm, NETAH deploys the generated event stream processing units on the computing nodes of the runtime environment. We presented the architecture of NETAH, as well as the processes of event stream composition network creation, mapping and deployment.

Chapter 5 and 6 have been dedicated to the application of NETAH in smart grid. Chapter 5 presented a specialization of NETAH in the context of a smart grid. We presented first an algorithm for deploying event processing units on the smart grid network. After that, we proposed a simulation of the smart grid network. This allowed us not to rely on a specific smart grid implementation, while keeping us away from the complexity of dealing with a real one. Then, we implemented NETAH for a simulated smart grid. Chapter 6 presented a real smart grid use case proposed within the context of the SOGRID project, and how it can be implemented using NETAH. The use case addressed the detection of resistive failures in a smart grid. We described this issue in terms of event stream composition. Then, using NETAH, we generated the corresponding event stream composition networks and we deployed them on a simulated smart grid topology. The experimentation demonstrated the relevance of our approach and solution.

7.2 Perspectives

Many challenges and possible improvements remain, covering various aspects of our solution. The following details some of them.

Short-term perspective

- **Event stream composition language**. Currently in our solution, in order to specify an event stream composition expression, a user should program the corresponding graph using an API. That is, by creating the corresponding set of vertices and edges manually[1]. This is fastiduous and not intuitive, and can be simplified with a query processing engine that provides:

  (i) A language that allows the definition of event stream composition expressions and the associated QoS in a declarative way.

  (ii) The parser that derives the event stream composition graph from the query defined using the proposed language.

Mid-term perspectives

- **Fault tolerance.** Highly distributed environments like smart grids and the IoT are subject to failures. The failure of a device can compromise the execution of a whole event stream composition network. Therefore, the event stream composition framework

[1] See an example at Listing 6.7 in Chapter 6.

should provide fault tolerance guarantees. More precisely, it should implement a failure detection service like [SRGR06, vRMH98] for detecting nodes that fails and reassign their event processing units on active nodes.

- **Capacity planning.** In our solution, a user specifies for each operator in an event stream composition expression, the estimates of memory size and CPU time necessary for its execution. Obtaining the corresponding values is a difficult task, as many aspects have to be considered, like the input rate of event streams, the complexity of the operator, the CPU rate of the target device, etc. This task can be delegated to a special component, that we refer to as a *capacity planner*, for which the role is to automatically estimate the memory and CPU time necessary for executing each operator of a given event stream composition expression. This can be done using elements of the queuing theory[AR02]. In fact, it is possible to model the event stream composition network as queuing system. Under this queuing model, both memory requirement and processing time of operators can be estimated.

- **Runtime QoS verification.** In our solution, QoS requirements are only considered during the deployment of event processing units, based on a static view of the environment states (network topology, network latency, etc.). There are changes to the runtime environment that can lead to QoS violation, like the modification of the topology, or a significant variation of the network latency. Therefore, a method for readapting the deployed event stream composition networks in response to a QoS violation at runtime is another possible extension of our work.

Long-term perspective

- **Security.** Security is another important aspect that has to be considered in a large scale event stream composition system. As the event streams produced within these systems report on the internal behavior of the system components (which can be sensitive), the event stream composition system can be the target of malicous attacks. The security issue needs to be addressed before event stream composition systems can be employed on a widespread basis. An idea could be the implementation of a security service within the event stream composition system, which could provide authentication for producers and consumers, and allow to set up access restrictions (e.g, access control lists) on event streams.

A Experimental evaluation of the operator mapping algorithms

We implemented the *OpMapping* and *OpMappingGreedy* algorithms using the Java programming language. We relied on the Jacop CSP solver [JaC15] to implement the bin packing constraint. For our experiments, we defined different types of processing 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 profiles. The latency of the communication links among the different devices was fixed too. We followed the same idea with event processing units. We defined different kind of event processing units with their associated memory and CPU time requirements. We generated event stream composition networks of different sizes, and comprising the specified event processing units.

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 the resulting operator mapping, which is captured by its cost. We generate 20 different inputs for the algorithm, each consisting in a network topology and an event stream composition network. Each network topology consisted in 15 nodes, and the number of event processing units in each event stream composition networks ranged from 7 to 10. For each input, we executed the *OpMapping* algorithm (brute force) and the *OpMappingGreedy*. We choose to run this experiment over a small network topology and small event stream composition networks 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 A.1, 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 A.2 compares the cost of operator mapping 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%.

We conducted another experiment in order to test how the greedy algorithm behaves on

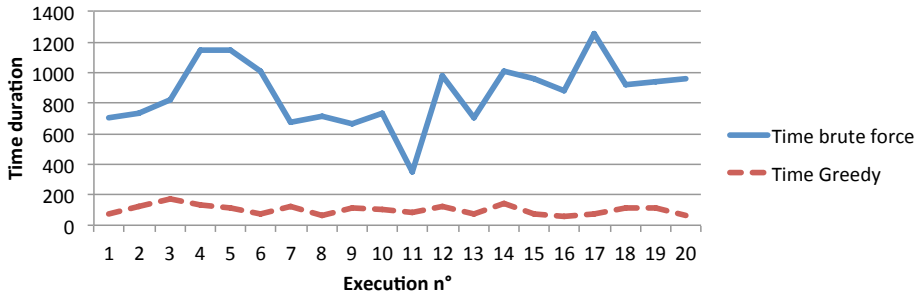

Figure A.1 – Operator deployment execution time: comparison between brute force and greedy algorithm

Figure A.2 – Cost of operator mapping: comparison between brute force and greedy algorithm

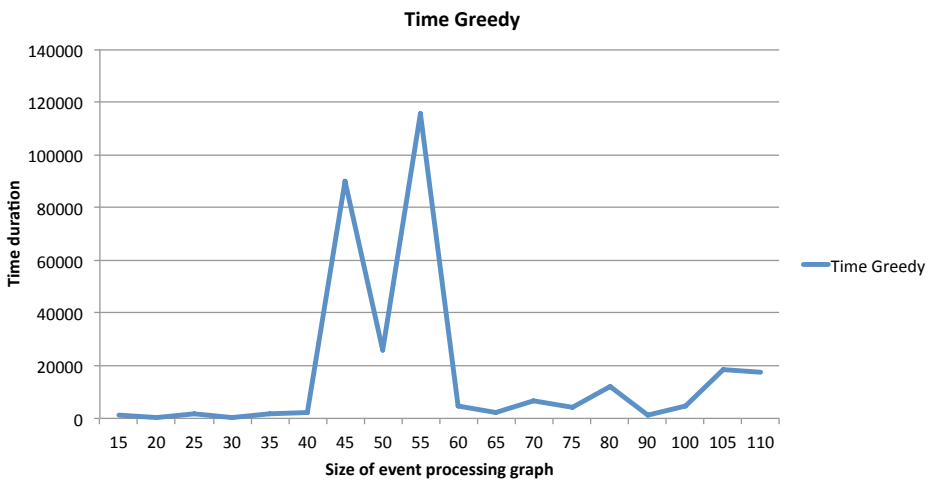

Figure A.3 – Scale up of the greedy algorithm

large event stream composition networks. We execute the algorithm over a network topology consisting in 50 nodes. The size of event stream composition networks 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 A.3 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 stream composition network growths. In fact, the structure of the event stream composition network is another factor that impacts the performance of the algorithm. In event stream composition networks 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 stream composition networks having size 45 and 60 were dense, this explains the peaks we observed in the duration time.

Bibliography

[Aba05] The Design of the Borealis Stream Processing Engine. *Cidr*, pages 277–289, 2005.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinl??nder, Matthias J. Sax, Sebastian Schelter, Mareike H??ger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[Apa16a] Homepage of Apache Flink, 2016. URL: https://flink.apache.org/ [accessed: 2016-06-05].

[Apa16b] Homepage of Apache Kafka, 2016. URL: http://kafka.apache.org/ [accessed: 2016-06-08].

[Apa16c] Homepage of Apache Samza, 2016. URL: https://samza.apache.org/ [accessed: 2016-06-05].

[AR02] Ivo Adan and Jacques Resing. Queueing Theory. *Technology*, 15(x):180, 2002.

[ASB10] Stefan Appel, Kai Sachs, and Alejandro Buchmann. Quality of service in event-based systems. In *CEUR Workshop Proceedings*, volume 581, pages 1–5, 2010.

[Bai94] PATHFINDER: A Pattern-Based Packet Classifier. (JANUARY 1994):115–123, 1994.

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *VLDB Journal*, 13(4):333–353, 2004.

[BCMR09] Krysia Broda, Keith Clark, Rob Miller, and Alessandra Russo. Sage: A logical agent-based environment monitoring and control system. In *Proceedings of the European Conference on Ambient Intelligence*, AmI '09, pages 112–117, Berlin, Heidelberg, 2009. Springer-Verlag.

[BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 350–, Washington, DC, USA, 2004. IEEE Computer Society.

[BRJ+02] Cole Brian, Eckstein Robert, Elliott James, Loy Marc, and Wood David. *Java Swing, 2nd Edition*. O'Reilly, 2nd edition, November 2002.

[CBB+03] M Cherniack, H Balakrishnan, M Balazinska, D Carney, U Cetintemel, Y Xing, and S Zdonik. Scalable distributed stream processing. *Proc. of CIDR*, 3:257–268, 2003.

[CC96] Christine Collet and T. Coupaye. Primitive and Composite Events in O2 Rules. In *Actes des 12ièmes Journées Bases de Données Avancées*, 1996.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, May 2000.

[CGJ+02] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: High performance network monitoring with an sql interface. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 623–623, New York, NY, USA, 2002. ACM.

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM.

[CKAK94] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. 14(1):1–26, 1994.

[CM09] Gianpaolo Cugola and Alessandro Margara. RACED : an Adaptive Middleware for Complex Event Detection. *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware*, pages 1–6, 2009.

[CM10] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 50–61, 2010.

[CV11] Victor Cuevas-Vicenttin. *Evaluation of hybrid queries based on service coordination*. Thesis, Université de Grenoble, 2011.

[Dep14] Smart grid vision and routemap, 2014. URL: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/285417/Smart_Grid_Vision_and_RoutemapFINAL.pdf [accessed: 2016-02-20].

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks*, 31(11-16):1155–1169, 1999.

[DGH+06] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.

[DGP+07] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. Cayuga : A General Purpose Event Monitoring System. *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, pages 412–422, 2007.

[EN10] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[ENLC+15] Orleant Epal Njamen, Martinez Lourdes, Collet Christine, Vargas-solar Genoveva, and Christophe Bobineau. Sogrid Deliverable 6.3.a: Quality of service based event streams composition and notification. Technical report, Grenoble INP - LIG, 2015.

[ENLC+16] Orleant Epal Njamen, Martinez Lourdes, Collet Christine, Vargas-solar Genoveva, and Christophe Bobineau. Sogrid Deliverable 6.3.b: Quality of service based event streams procesing system. Technical report, Grenoble INP - LIG, 2016.

[Esp15] Homepage of Esper, 2015. URL: http://esper.codehaus.org/ [Accessed: 2015-03-27].

[Eur06] Vision and strategy for europe's electricity networks of the future, 2006. URL: http://bookshop.europa.eu/uri?target=EUB:NOTICE:KINA22040:EN:HTML [accessed: 2016-02-20].

[Eve16] Google guava eventbus, 2016. URL: https://github.com/google/guava/wiki/EventBusExplained [accessed: 2016-03-11].

[GADI08] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immennan. On supporting kleene closure over event streams. In *Proceedings - International Conference on Data Engineering*, pages 1391–1393, 2008.

[GD94] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using Petri nets. *Proceedings of IEEE International Workshop on Research Issues in Data Engineering: Active Databases Systems*, 1994.

[GWC+06] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. SASE: Complex Event Processing over Streams. *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 1–5, 2006.

[Hin03] Annika Hinze. Efficient filtering of composite events. In *Proceedings of the british national database conference*, pages 207–225, 2003.

[JaC15] Homepage of JaCoP, 2015. URL: http://jacop.osolpro.com/ [Accessed: 2015-08-30].

[JLKY08] X. Jin, X. Lee, N. Kong, and B. Yan. Efficient complex event processing over rfid data stream. In *International Conference on Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS*, pages 75–81, May 2008.

[JM05] Theodore Johnson and S Muthukrishnan. A heartbeat mechanism and its application in gigascope. In *Proceeding VLDB '05 Proceedings of the 31st international conference on Very large data bases*, pages 1079–1088, 2005.

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment*, 1:1379–1390, 2008.

[JUN10] Java universal network/graph framework, 2010. URL: http://jung.sourceforge.net/ [accessed: 2016-03-10].

[LS03] Alberto Lerner and Dennis Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 345–356, 2003.

[Luc96] David Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Technical report, Stanford University, 1996.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[LWK13] B Lohrmann, Daniel Warneke, and Odej Kao. Nephele streaming: stream processing under QoS constraints at scale. *Cluster Computing*, 2013.

[MsSS97] Masoud Mansouri-samani, Morris Sloman, and Morris Sloman. Gem - a generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4, 1997.

[NDM+01] Jeffrey Naughton, David Dewitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, and Stratis Viglas. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24:27–33, 2001.

[PCH+15] Boyang Peng, Roy Campbell, Mohammad Hosseini, Zhihao Hong, and Reza Farivar. R-Storm : Resource-Aware Scheduling in Storm. *ACM Middleware*, 2015.

[RW97] David S Rosenblum and Alexander L Wolf. A design framework for Internet-scale event observation and notification. *ACM SIGSOFT Software Engineering Notes*, 22(6):344–360, 1997.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. *Proceedings of the Third ACM International Conference on Distributed EventBased Systems DEBS 09*, page 1, 2009.

[SOG16] Webpage of the sogrid project, 2016. URL: http://www.sogrid.info/ [accessed: 2016-03-23].

[SRGR06] Rajagopal Subramaniyan, Pirabhu Raman, Alan D. George, and Matthew Radlinski. Gems: Gossip-enabled monitoring service for scalable heterogeneous distributed systems. *Cluster Computing*, 9(1):101–120, January 2006.

[STO13] Storm: Distributed and fault-tolerant real-time computation, 2013. URL: http://storm.apache.org/ [Accessed: 2016-06-05].

[Str15] Homepage of TIBCO StreamBase, 2015. URL: http://www.streambase.com/ [accessed: 2016-06-5].

[Sun16] Sun Microsystems. JavaBeans(TM) Specification, 2016. URL: http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html [Accessed: 2016-01-10].

[SW04] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 324–335. VLDB Endowment, 2004.

[vRMH98] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, London, UK, UK, 1998. Springer-Verlag.

[VSC02] Genoveva Vargas-Solar and Christine Collet. *ADEES: An Adaptable and Extensible Event Based Infrastructure*. 2002.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[XCTS14] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *Proceedings - International Conference on Distributed Computing Systems*, pages 535–544, 2014.

[YBMM94] Masanobu Yuhara, Brian N Bershad, Chris Maeda, and J Eliot B Moss. Efficient Packet Demultiplexing for Multiple Endpoints and LargeMessages. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, page 13, 1994.

[ZCDD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.