SEMIoTICS

Deliverable D4.9
SEMIoTICS Monitoring, Prediction and Diagnosis Mechanisms (final)

Deliverable release date	30.04.2020
Authors	1. Domenico Presenza, Keven T. Kearney (ENG) 2. Christos Tzagkarakis, Manolis Michalodimitrakis, Nikolaos Petroulakis (FORTH) 3. Felix Klement, Korbinian Spielvogel, Henrich C. Pöhls (UP) 4. Łukasz Ciechomski (BS) 5. Mirko Falchetto (ST) 6. Konstantinos Fysarakis, Manolis Chatzimpyrros, Thodoris Galousis, Michalis Smyrlis (STS)
Responsible person	Domenico Presenza (ENG)
Reviewed by	Manolis Michalodimitrakis, Nikolaos Petroulakis (FORTH), Jordi Serra (CTTC), Konstantinos Fysarakis (STS), Mirko Falchetto (ST), Urszula Stawicka, Pawael Gawron (BS)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Christos Verikoukis) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Christos Verikoukis, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final
Version	1.0
Dissemination level	Public

Table of Contents

Acronym	Definition
AE	Auto Encoder
AI	Artificial Intelligence
AWS	Amazon Web Services
BPTT	Back Propagation Through Time
CEP	Complex Event Processor
CGNN	Causal Generative Neural Network
DoW	Description of Work
eBPF	extended Berkley Packet Filter
ETC	Event Triggered Causality
FPR	False Positive Rate
GAN	Generative Adversarial Network
GUI	Graphical User Interface
IIoT	Industrial Internet of Things
IoT	Internet of Things
KPI	Key Performance Indicator
LLDP	Link Layer Discovery Protocol
LOOCV	leave-out-one-device cross validation
LSTM	Long-Short-Term-Memory network
MPD	Monitoring, Prediction and Diagnosis Component
MCU	Micro Controller Unit
ML	Machine Learning
NMS	Network Management System
NSGI	Next Generation Service Interface
ODL	OpenDaylight
OF	OpenFlow
OMP	Orthogonal Matching Pursuit algorithm
PaaS	Platform-as-a-Service
PO	Pattern Orchestrator

QoS	Quality of Service
RC	Recipe Cooker
RNN	Recurrent Neural Network
SDK	Software Development Kit
SDN	Software Defined Network
SNMP	Simple Network Management Protocol
SNS	Simple Notification Service
SPDI	Security, Privacy, Dependability, Interoperability
SR	Sparse Representation
SVDD	Support Vector Data Description
TPR	True Positive Rate
VM	Virtual Machine
WoT	Web of Thing

1. INTRODUCTION

This deliverable presents the design of the SEMIoTICS Monitoring, Prediction and Diagnosis (MPD) Component along with algorithmic and technological options considered for the implementation of its key functionalities.

The main objective of the MPD is the fusion of intra and cross-layer monitoring results generated by monitors that may exist on the platforms. The MPD component detects violations of SPDI patterns and other conditions related to different smart objects and components of available IoT applications. Fusion of cross-layer monitoring data is necessary for detecting specific types of attacks. In such cases, multi-sensor cross-layer data need to be fused in specific ways to detect the attack.

To achieve the intra-layer and cross-layer monitoring objective, the design of the MPD includes mechanisms to connect the MPD respectively with different IoT platform, cloud monitors and smart object event captors.

Monitoring of SPDI patterns must be continuous in spite of the dynamicity of IoT applications (e.g. binding of new smart objects and components). Therefore, the design of the MPD includes also mechanisms to adapt dynamically the monitoring configuration (e.g. select an alternative event source whenever one in use is no longer available).

The MPD offers also predictive and diagnostic mechanisms. The predictive mechanisms are needed to ensure that the business applications or other components of the SEMIoTICS architecture can set up countermeasures before the SPDI patterns are violated. The diagnostic mechanisms are needed to ensure the effectiveness of the countermeasures, by managing to correctly identify and resolve the sources of actual/potential SPDI property violations in order to resolve them. Both mechanisms rely on models generated using the information obtained from the monitoring mechanism.

There is a mutual dependency between the monitoring mechanisms and the predictive/diagnostic mechanisms. On one side the monitoring mechanisms provide the information (i.e. observations) enabling the production of diagnosis and predictions. On the other side the predictive and diagnostic mechanisms provide the monitoring mechanisms with those prediction and diagnosis capabilities enabling the proper adaptation of the monitoring configurations and, hence, the continuous production of monitoring information.

1.1. Addressed SEMIoTICS Requirements

Table 1 shows which are the SEMIoTICS Requirements addressed by the Monitoring component described in the present deliverable:

TABLE 1. REQUIREMENTS ADDRESSED BY THE MPD

Req. ID	Requirement Description	How it is addressed by MPD	Section
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern	The MPD allows detecting Network level events thanks to the availability of adapters able to capture the events generated by the SDN Controller and Virtual Infrastructure Manager (VIM).	2.2.7
R.P.4	A short data retention period MUST be enforced, and maintaining data for longer than necessary avoided.	The MPD uses Complex Event Processing technology to aggregate data. In fact, CEP technology allow detecting events patterns directly in the stream of events without the need to store the events in a database for subsequent processing.	2.3

Req. ID	Requirement Description	How it is addressed by MPD	Section
R9.4	The cloud platform SHALL to be able to monitor the execution of an app, in particular its interactions with other apps, the network interface, and APIs.	The MPD provides adapters that enable to monitor the execution of apps by means of the native monitoring capabilities of Cloud and IoT platforms	2.2
R.BC.20	The backend layer must aggregate intra-layer as well as inter-layer SPDI status information to enable local and global intelligence reasoning and adaptation.	The MPD provides adapters to capture events generated by the backend layer. The MPD aggregates events using CEP technology. MPD defines strategies to translate SPDI pattern into monitoring policies.	1.2 2.5 2.6
R.NL.13	The network layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation.	The MPD provides adapters to capture events generated by network layer. The MPD aggregates events using CEP technology. MPD defines strategies to translate SPDI patterns into monitoring policies.	1.2 2.5 2.6
R.FD.15	The field layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation.	The MPD provides adapters to capture events generated by field devices. The MPD aggregates events using CEP technology. MPD defines strategies to translate SPDI pattern into monitoring policies.	1.2 2.5 2.6
R.UC2.10	The SEMIoTICS platform SHOULD allow the SARA components (e.g. SARA Hubs) to query and aggregate (e.g. to average) the values of a resource (e.g. current measured temperature) hosted by a group of field devices. The SARA solution defines a group of devices by specifying filtering criteria over the set of registered devices.	The MPD provides adapters to capture events generated by field devices. The Query language of the MPD provides means to express filtering conditions over the sources of events.	2.2.8, 2.5
R.UC2.12	The SEMIoTICS platform SHOULD allow SARA components to delegate to the platform the computation of complex functions over the data received by field devices. These computations may result either in the generation of higher level observation events (e.g. significant Patient events abstracted form sensor data) towards the ACS or in sensors configuration parameters (including actuators command).	The MPD provides adapters to capture events generated by field devices. Moreover, The Query language of the MPD provide business IoT applications (e.g. SARA) with means to specify an high level observation event as the occurrence of a specific pattern of events within the stream of events generated by field devices.	2.2.8, 2.5

1.2. Relations with other SEMIoTICS components

The diagram in Figure 1 shows the positioning of the MDP in the context the SEMIoTICS architecture. A complete description of the SEMIoTICS architecture and components can be found in deliverable D2.5 - “SEMIoTICS high level architecture (final)”.

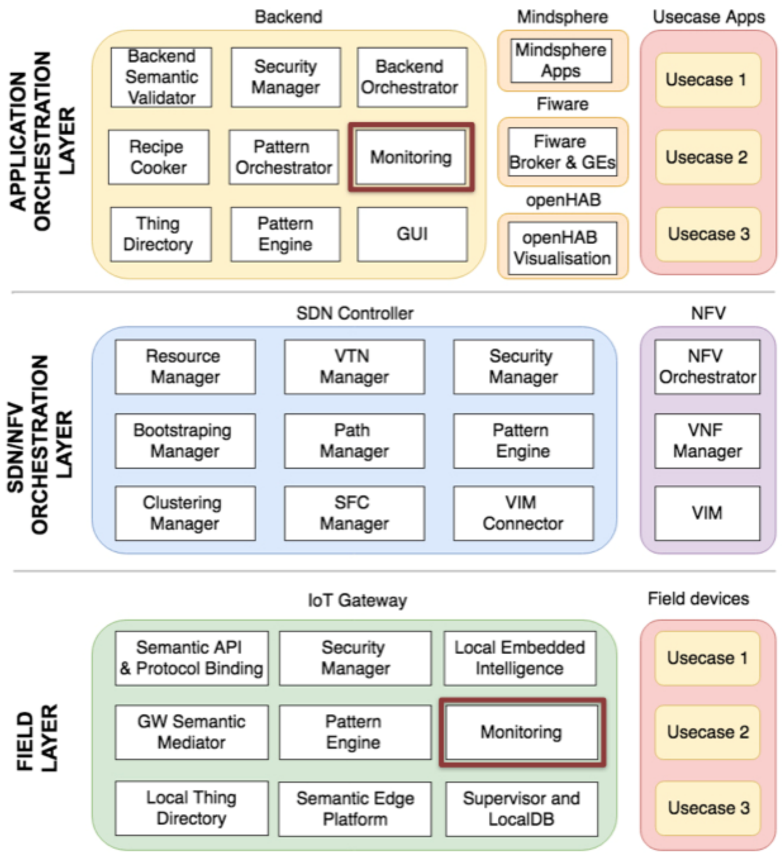


FIGURE 1 MONITORING COMPONENT IN SEMIOTICS

The UML component diagram in Figure 2 shows the relations between the MDP and the other components of the SEMIoTICS architecture. More specifically some of the components the MPD interacts are:

- Pattern Engine. Pattern Engine delegates monitoring tasks to the MPD by means of the Query interface. Monitoring Components notifies detected events by means of the Event Consumer interface. (see section 2.5)
- GUI. The Graphical User Interface may submit monitoring tasks to the MPD by means of the Query interface. The GUI receives detected events by means of the Event Consumer interface. (section 2.5)
- Cloud Platforms (e.g. FIWARE Context Broker or MindSphere). The MPD is able to consume events generated by specific cloud platform thanks to the availability of platform specific adapters. (section 2.2)
- SDN Controller. The Monitoring Controller consumes network level events generated by the SDN Controller and made available by the OpenDaylight (ODL) Cardinal plugin (or similar). (section 2.2.7)
- VIM: The Monitoring Controller consumes network level events generated by the OpenStack (Virtual Infrastructure Manager) and made available via the Compute API. (section 2.2.5)

• Field Devices. Monitoring Component aggregates events generated by Field Devices and made available via the Web of Things (WoT) interface. (section 2.2.8)

• Field Monitoring Component (i.e. instances of the Monitoring Component deployed in the field (e.g. within gateways). Monitoring Component consumes also events generated by Field Monitoring Components using the Query interface (section 2.5).

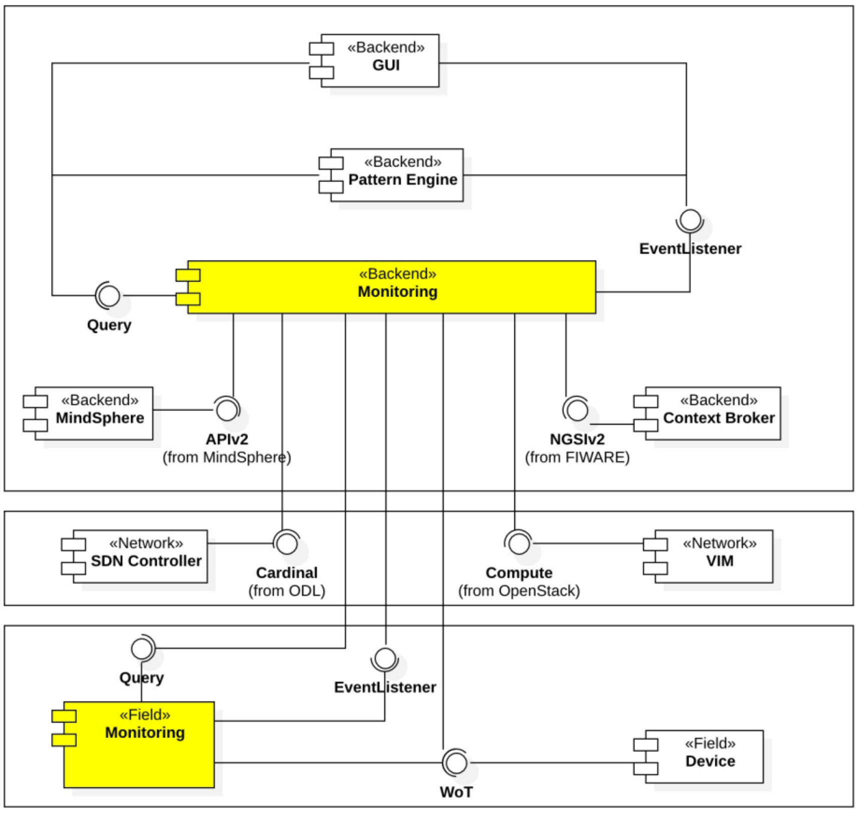


FIGURE 2 DEPENDENCIES OF THE MPD COMPONENT

1.3. Methodology and document structure

This deliverable D4.9 is the result of the joint effort by all contributing partners during the development of the project.

The work proceeded along four directions:

- The design of the architecture of the SEMIoTICS Monitoring Component. The result of this thread is presented in the section 2 - “Monitoring Management”
- The investigation and selection of the suitable approaches for prediction and diagnosis. The resulting selection is presented in section 3 - “Predictive Mechanisms” and 4 - “Diagnosis Mechanisms”.
- The scouting and experimentation of the technologies (e.g. software libraries, services, datasets) enabling the implementation of the SEMIoTICS Monitoring Component. The results of this thread of work informed the architecture presented in section 2.
- The implementation of a prototype of the MPD component. The results of this thread are reported in sections 2.6 - “Using the MPD Component” and 5 - “Implementation aspects”.

With respect to the previous version of the deliverable, i.e. D4.2 - "SEMIoTICS Monitoring, prediction and diagnosis mechanisms (draft)", the present document introduces the following main additions/changes:

- section 1.4 containing the answers to the design questions left open by the previous version of the deliverable (i.e. D4.2)
- section 2.6 presenting examples of queries for the MPD component. This section is intended to be a short tutorial of the MPD.
- section 2.7 presenting additional examples of the mapping of SPDI patterns into MPD Queries.
- section 4.2 extended with additional evaluation results.
- section 4.4 extended with conclusions about the possibility to use the Generative Adversarial Network for the task of anomaly detection.
- chapter 5 containing the description of the most relevant implementation aspects of the various modules of the MPD.

Bi-weekly meetings were run to keep aligned each partner about the work of the others, share results of experimentations and take design decisions. These meetings were devoted to discussing not only the design issues concerning Task 4.2 but also those faced by Task 4.3. This because, as already anticipated in the Description of Work (DoW) of the project and further explained in this document, there is a dependency between the Monitoring Component developed in Task 4.2 and the Embedded Intelligence and local Analytics mechanisms provided by Task 4.3 and described in deliverables D4.3 (1st Daft) and D4.10 (final).

1.4. Answers to open design questions from D4.2

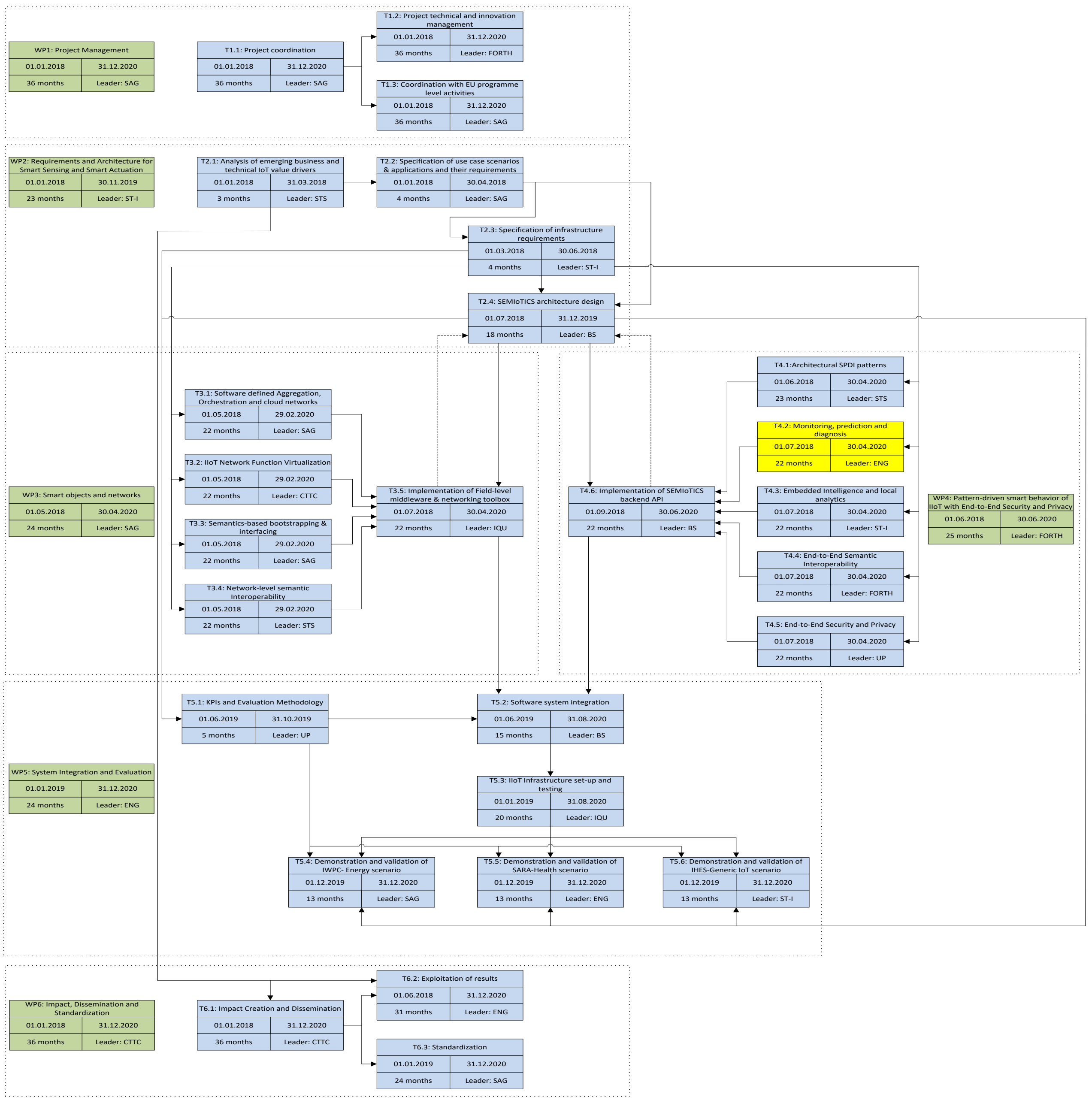
The deliverable D4.2, along with the overall design of the Monitoring Component, has presented a number of enabling technologies and approaches suitable for its implementation. At month 17 of the project (date of release of D4.2 deliverable) the following design questions were still open:

- Which is the most suitable CEP technology to be integrated into the SEMIoTICS Monitoring Component?
- Is CloudEvents format rich enough to carry the information needed by the SEMIoTICS Monitoring Component?
- Is the monitoring language presented in this deliverable rich enough to fulfill the monitoring requirement of the Pattern Orchestrator?
- Which are the signalers, among those described in section, more valuable for the SEMIoTICS platform and use cases that need to be delivered with the final version of the monitoring component?

The design questions listed above were answered through the development of the prototype of the MPD. The answers to the open design questions for the monitoring component sketch the delta between D4.2 and D4.9 presented herein, and can now be summarized as follows:

- Apache Flink was selected as CEP (section 5.6)
- the CloudEvent format was chosen and an MPD-specific payload defined (section 2.5)
- the monitoring language worked to fulfill the requirements of the Pattern Orchestrator (section 2.7)
- given the requirements from the project's use cases the following signalers are implemented in the current version of the MPD: Fiware Signaler, Network Signaler, WoT Signaler, Backend Orchestrator Signaler.

1.4. PERT chart of SEMIoTICS


WP1: Project Management | 01.01.2018 | 31.12.2020 | 36 months | Leader: SAG
T1.1: Project coordination | 01.01.2018 | 31.12.2020 | 36 months | Leader: SAG
T1.2: Project technical and innovation management | 01.01.2018 | 31.12.2020 | 36 months | Leader: FORTH
T1.3: Coordination with EU programme level activities | 01.01.2018 | 31.12.2020 | 36 months | Leader: SAG
WP2: Requirements and Architecture for Smart Sensing and Smart Actuation | 01.01.2018 | 30.11.2019 | 23 months | Leader: ST-I
T2.1: Analysis of emerging business and technical IoT value drivers | 01.01.2018 | 31.03.2018 | 3 months | Leader: STS
T2.2: Specification of use case scenarios & applications and their requirements | 01.01.2018 | 30.04.2018 | 4 months | Leader: SAG
T2.3: Specification of infrastructure requirements | 01.03.2018 | 30.06.2018 | 4 months | Leader: ST-I
T2.4: SEMIoTICS architecture design | 01.07.2018 | 31.12.2019 | 18 months | Leader: BS
WP3: Smart objects and networks | 01.05.2018 | 30.04.2020 | 24 months | Leader: SAG
T3.1: Software defined Aggregation, Orchestration and cloud networks | 01.05.2018 | 29.02.2020 | 22 months | Leader: SAG
T3.2: IIoT Network Function Virtualization | 01.05.2018 | 29.02.2020 | 22 months | Leader: CTTC
T3.3: Semantics-based bootstrapping & interfacing | 01.05.2018 | 29.02.2020 | 22 months | Leader: SAG
T3.4: Network-level semantic Interoperability | 01.05.2018 | 29.02.2020 | 22 months | Leader: STS
T3.5: Implementation of Field-level middleware & networking toolbox | 01.07.2018 | 30.04.2020 | 22 months | Leader: IQU
T4.6: Implementation of SEMIoTICS backend API | 01.09.2018 | 30.06.2020 | 22 months | Leader: BS
T4.1:Architectural SPDI patterns | 01.06.2018 | 30.04.2020 | 23 months | Leader: STS
T4.2: Monitoring, prediction and diagnosis | 01.07.2018 | 30.04.2020 | 22 months | Leader: ENG
T4.3: Embedded Intelligence and local analytics | 01.07.2018 | 30.04.2020 | 22 months | Leader: ST-I
T4.4: End-to-End Semantic Interoperability | 01.07.2018 | 30.04.2020 | 22 months | Leader: FORTH
T4.5: End-to-End Security and Privacy | 01.07.2018 | 30.04.2020 | 22 months | Leader: UP
WP4: Pattern-driven smart behavior of IIoT with End-to-End Security and Privacy | 01.06.2018 | 30.06.2020 | 25 months | Leader: FORTH
WP5: System Integration and Evaluation | 01.01.2019 | 31.12.2020 | 24 months | Leader: ENG
T5.1: KPIs and Evaluation Methodology | 01.06.2019 | 31.10.2019 | 5 months | Leader: UP
T5.2: Software system integration | 01.06.2019 | 31.08.2020 | 15 months | Leader: BS
T5.3: IIoT Infrastructure set-up and testing | 01.01.2019 | 31.08.2020 | 20 months | Leader: IQU
T5.4: Demonstration and validation of IWPC- Energy scenario | 01.12.2019 | 31.12.2020 | 13 months | Leader: SAG
T5.5: Demonstration and validation of SARA-Health scenario | 01.12.2019 | 31.12.2020 | 13 months | Leader: ENG
T5.6: Demonstration and validation of IHES-Generic IoT scenario | 01.12.2019 | 31.12.2020 | 13 months | Leader: ST-I
WP6: Impact, Dissemination and Standardization | 01.01.2018 | 31.12.2020 | 36 months | Leader: CTTC
T6.1: Impact Creation and Dissemination | 01.01.2018 | 31.12.2020 | 36 months | Leader: CTTC
T6.2: Exploitation of results | 01.06.2018 | 31.12.2020 | 31 months | Leader: ENG
T6.3: Standardization | 01.01.2019 | 31.12.2020 | 24 months | Leader: SAG

Please note that the PERT chart is kept on task level for better readability.

2. MONITORING MANAGEMENT

2.1. Monitoring Management layer architecture

This section presents the overall architecture of the monitoring management layer. The SEMIoTICS monitoring component has two key functional requirements:

- To generate specific messages in response to the reception of a set of messages generated by the components of an IoT application and matching some condition specified in the monitoring component by a client application (Monitoring requirement).
- To guarantee that the messages needed to decide whether to generate a message can be produced by an IoT application and received by the monitoring component (Observability property).

The first of the two technical requirements listed above directly stems from the general SEMIoTICS platform requirement R.GP.4 “Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern” (see deliverable D2.3 - “Requirements specification of SEMIoTICS framework”). In fact, the capacity of the monitoring component to detect and signal the occurrence of specific patterns of events allows the components responsible for the enforcement of SPDI patterns (i.e. Pattern Engine) to delegate to the monitoring component the monitoring task whilst retaining for itself the triggering of adaptation actions, id needed. Moreover, it also addresses the requirements

- R.UC2.10 - “The SEMIoTICS platform SHOULD allow the SARA components (e.g. SARA Hubs) to query and aggregate (e.g. to average) the values of a resource (e.g. current measured temperature) hosted by a group of field devices”
- R.UC2.12 - “The SEMIoTICS platform SHOULD allow SARA components to delegate to the platform the computation of complex functions over the data received by field devices. These computations may result either in the generation of higher-level observation events (e.g. significant Patient events abstracted form sensor data) towards the ACS or in sensors configuration parameters (including actuators command).”

The second requirement, i.e. the ability to guarantee the generation of the monitoring events needed to serve the monitoring tasks submitted by client applications, requires that the monitoring component is able to adapt to the changing conditions that may occur in all layers (field/network/backend) of the monitored infrastructure. To achieve this objective the monitoring component needs prediction and diagnosis capabilities. The Monitoring component uses causal inference to make predictions and diagnosis. These inference capabilities are enabled by the availability of a causal model learned from the observation of monitoring events generated either by the queries submitted client application or by queries generated by the monitoring component itself (self-monitoring queries).

Figure 3 presents the main required inputs and outputs of the SEMIoTICS monitoring component.

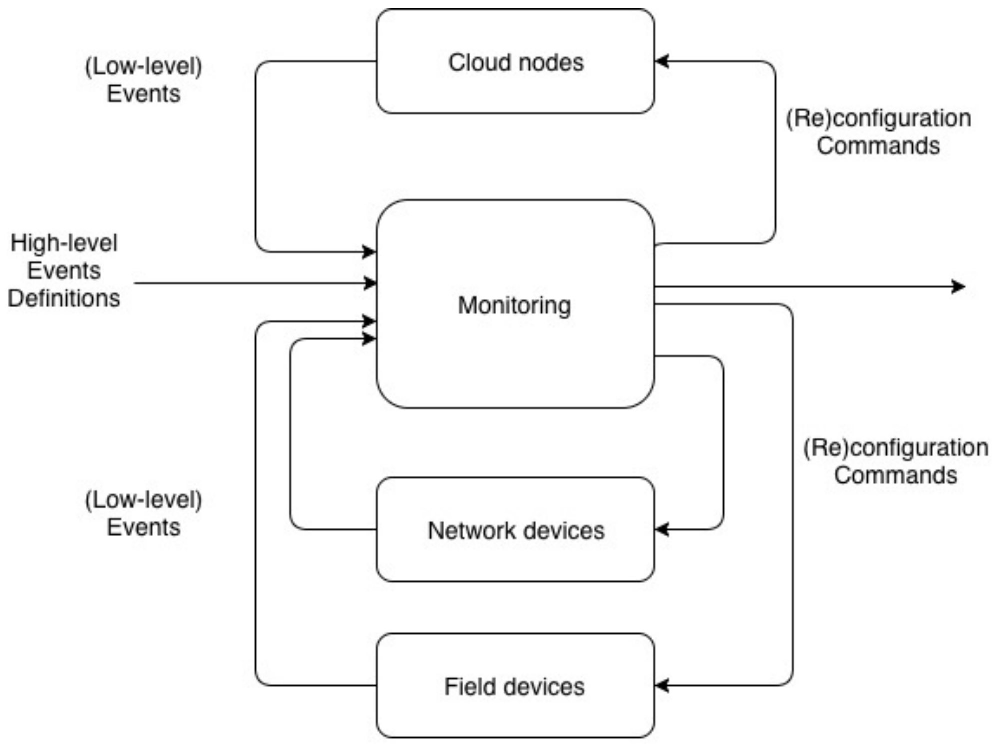


FIGURE 3 MAIN INPUT AND OUTPUT OF THE MONITORING COMPONENT

In more detail, the Monitoring component receives as input:

- **Low-level events**: the messages generated by the computational nodes belonging to the three layers identified by the SEMIoTICS architecture: field (e.g. sensors, gateways), network (e.g. routers) and cloud (e.g. FIWARE cloud services, MindSphere services). These low-level events are generated by the computational nodes by means of signaling mechanisms specific to the technology used to implement a computational node. The possibility of the monitoring component to process events from the cloud, network and field level directly address the requirement R9.4 - "The SEMIoTICS platform SHOULD allow SARA components to delegate to the platform the computation of complex functions over the data received by field devices. These computations may result either in the generation of higher-level observation events (e.g. significant Patient events abstracted form sensor data) towards the ACS or in sensors configuration parameters (including actuators command"
- **High-level events definitions**: the conditions stating whether a new event should be generated in response of the reception of a set of low-level events.

The monitoring component emits as outputs:

- **High-level events**: the messages generated by the monitoring component itself in response to the reception of a set of low-level events matching one of the events definitions.
- **Configuration commands**: messages requesting a specific configuration of the mechanisms used by the computational nodes to generate the low-level events. The ability to issue these commands allows the monitoring component to properly select and configure the signaling mechanisms needed for the monitoring purpose.

Given the above schema it is worth to noting that low-level events serve two purposes:

- to decide whether to emit a high-level event

- to decide whether to request a reconfiguration of the signaling mechanisms used at the different levels (Field, Network, Cloud)

Hence the Monitoring component can be decomposed in three main sub-components (Figure 4):

- **Business Events Monitor** responsible for matching low-level events against the conditions specified by the High-level events definitions. We term Business Events the high-level events generated by the Business Events Monitor and directed to a client application of the Monitoring component whilst we term Control Events refer to those high-level events directed to the other components within the Monitoring component.
- **Monitoring Controller** responsible for configuring, observing and, if needed, reconfiguring both the signaling mechanisms serving the *Business Events Monitor* and the *Causal Model Identifier*. The Monitoring Controller bases its decisions about configurations and observations on the (causal) model made available by the Causal Model Identifier component (described below). Whenever needed the Monitoring Controller can adapt the observation and, hence, fulfill the observability requirement thanks to the availability of this causal model. As an example, based on the Causal Model the Monitoring Controller might decide that, in order to guarantee that a type of high-level event is produced as requested by a client application, there is also the need to monitor additional type of events. This decision will result in the submission of extra monitoring tasks to the Business Events Monitor.
- **Causal Model Identifier** having the role to build the causal models. These models are created using as input both the (Re)configuration commands emitted by the Monitoring Controller and the events generated by the Business Event Monitor. The Causal models identified by this component are consumed by (i) the Monitoring Controller to configure observations (see above) and (ii) the Business Event Monitor to infer events not directly observable.

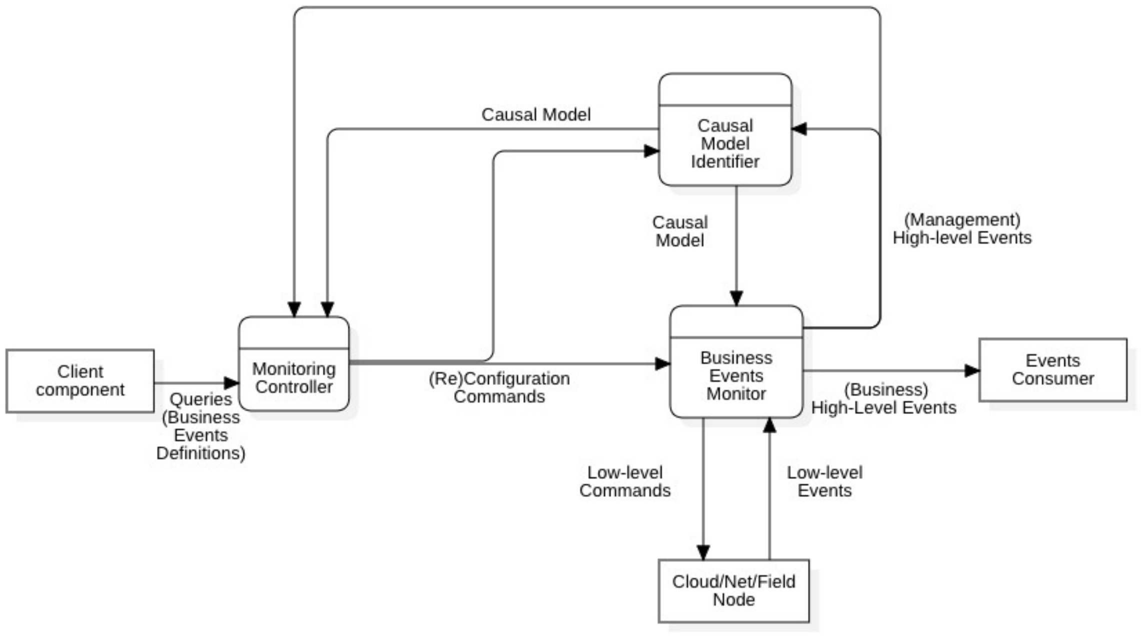


FIGURE 4 MAIN COMPONENTS AND DATA FLOW OF MONITORING

Given the context and aims described so far, the design of the SEMIoTICS MPD required the definition of:

- The technologies and languages made available by the different IoT platform to observe and control the IoT applications at the Cloud/Network/Field level. The options are presented in section 2.2.
- The technology for the processing of the low-level events once they are represented in a common format. The options that were considered are presented in section 2.3.

- The language used to define the high-level events in terms of the low-level events (i.e. the language to express in a machine readable format rules like "the occurrence of three consecutive 'failed login' events should produce the rise of an 'account violation attempt' event). This language is presented in section 2.4.
- The content of the messages representing the high-level events generated by the Business Event Processor and received by the client applications of the Monitoring component. This is presented in section 2.5
- The algorithms implemented by the Causal Model Identifier to identify the model needed by the Monitoring Controller. The algorithms are discussed in section 3.3.
- The prediction and diagnosis algorithms enabling the Monitoring Controller to decide how to configure (or reconfigure) the Business Event Monitor in order to comply with the requests received from its client applications. Possible approaches are presented in section 3.2 and section 3.3.

2.1.1. BUSINESS EVENT MONITOR ARCHITECTURE

The central components of the Business Event Monitor (Figure 5) are a Complex Event Processor (section 2.3 explores candidate technologies available for the implementation of this component) and a collection of Anomaly Detectors.

Anomaly Detectors are a collection of components specialized to detect anomalies in the flow of events generated by Cloud/Network/Field nodes (Chapter 4 presents a set of algorithms suitable for this purpose).

Driven by the configurations and queries received from client components (e.g. Monitoring Controller) the **Complex Event Processor** (CEP) processes the events received from the monitored nodes and produce the high-level events defined by the client applications of the SEMIoTICS Monitoring Component.

The design choice to use CEP technology to process the events received from the monitored nodes is rooted in the requirement R.P.4 - "A short data retention period MUST be enforced, and maintaining data for longer than necessary avoided". In fact, CEP technology allows detecting events patterns directly in the stream of events without the need to store the events in a database for subsequent processing.

It is worth mentioning here that the high-level events produced by the CEP can be divided in two broad categories:

- Business events: generated in response to requests coming from clients of the Monitoring Component
- Management events: generated in response to requests coming from other components belonging to the Monitoring component (e.g. the Monitoring Controller). These management events are needed to support the adaption of the Monitoring Component. Management events may concern both changes of state of the monitored nodes and measures of the performance of the Complex Event Processor.

Within the Business Event Monitor (Figure 5) an **Event Signaler** component is responsible for:

- Translating the configuration commands received by the Event Monitor into the command messages actually accepted by the nodes belonging to the monitored infrastructures.
- Translating the platform specific events generated by each node of the monitored infrastructure into the format accepted by the Complex Events Processor.

Altogether the collection of event signalers within the Event Monitor acts as a communication bus between the Event Monitor and the different platforms of IoT applications. An Event Signaler can reside either in the cloud layer (e.g. within the same VM where the other components of the Monitoring Component reside) or in the field layer (e.g. within a SEMIoTICS gateway). In latter case, the events generated by Events Signalers can also result from the aggregation of low-level events by means of algorithms executed locally within the field device. The deliverable D4.10 - "Embedded Intelligence and local Analytics (final)" of the SEMIoTICS project presents possible events aggregation algorithms suitable to be executed in a constrained resources field device that are generated thanks to dedicated statistical / machine learning analytics algorithms conveniently designed.

The **Event Predictor** component has the role to infer both future events and events not directly observable from the cloud/net/field nodes (e.g. because of the lack of a suitable sensor). Section 3 (prediction mechanisms) presents possible approaches for event prediction suitable to be implemented by the Event Inference Engine.

The Event Predictor uses to Causal Model learned by the Causal Model Identifier to infer events not directly observable through the Events Signalers. As an example, if causal model states that there is a causal dependency between a light being turned on and someone entering a room the reasoner could infer the event "someone entered room A12" upon the observation of the event "light switched on in room A12". The Event Predictor computes a likelihood for the inferred events. The likelihood value is conveyed to the Event Consumers by means of the likelihood attribute of an event object type (see section 2.4). Inferred events can be consumed either by external Events Consumers or by the Complex Event processor.

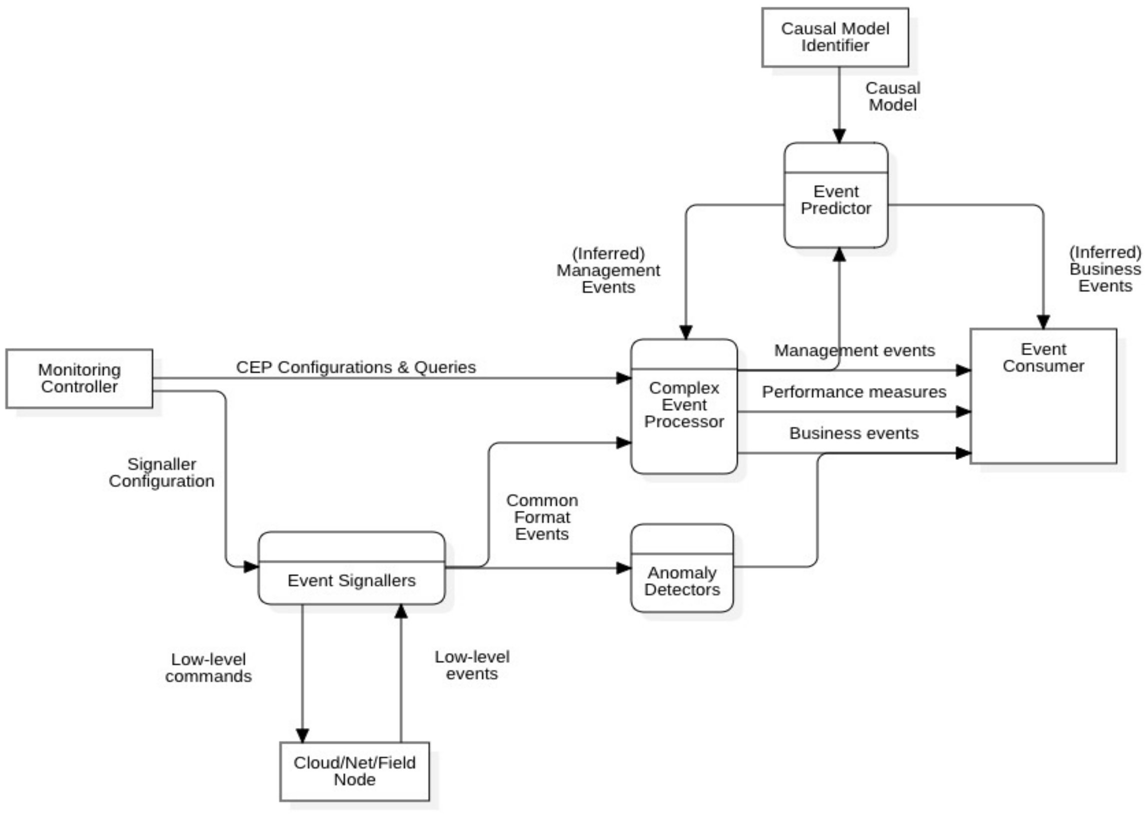


FIGURE 5 EVENTS MONITOR DATA FLOW

2.2. Cloud and IoT Platforms Monitoring Capabilities

This section describes the monitoring capabilities made available by third party Cloud and IoT platforms (e.g. AWS, MindSphere) (Figure 6). Each subsection presents capabilities for both the observing and controlling made available through APIs by a platform. The repertoire of platforms considered in this section is informed by the results presented in deliverable D2.1 - "Analysis of IoT Value Drivers" and by the technological choices envisaged by the main three use cases.

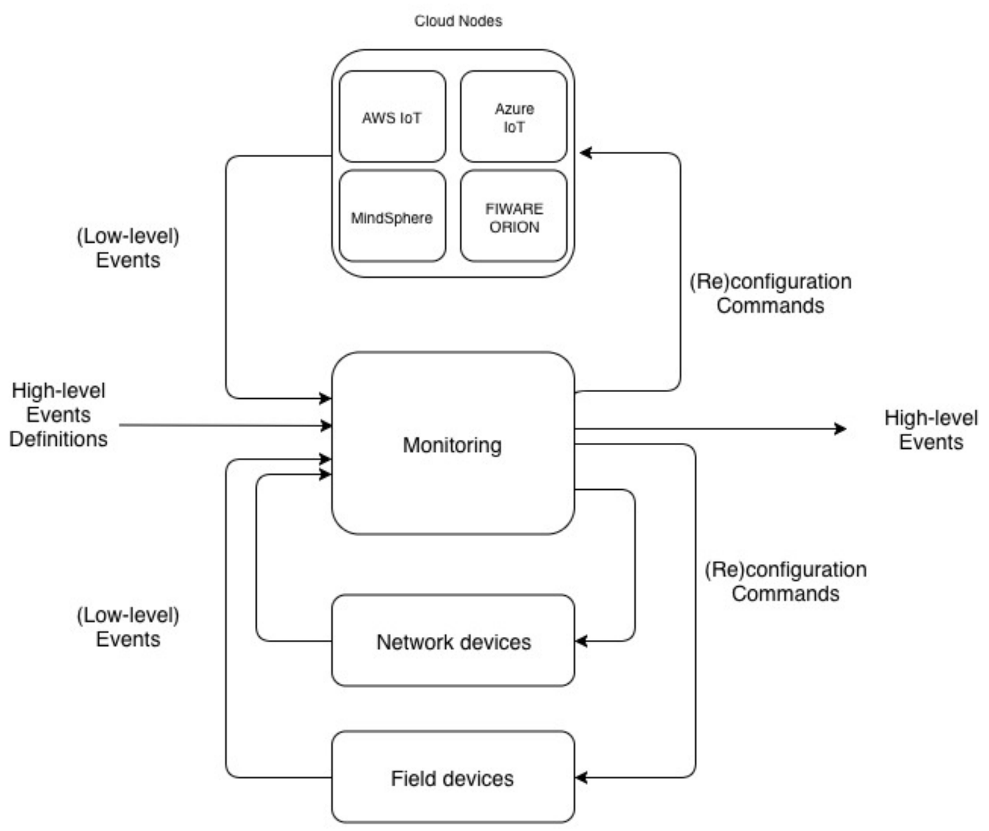


FIGURE 6 IOT PLATFORMS AND SEMIOTICS COMPONENTS

2.2.1. AWS IOT CORE

AWS IoT [1] is a collection of web-services offered by Amazon for managing bi-directional (secure) communications between internet-connected devices (sensors, actuators, embedded micro-controllers, smart appliances, etc.) and applications hosted on the AWS Cloud. Applications can collect telemetry for data from multiple devices, store and analyze this data, and provide users with remote control of the devices. More details of the AWS IoT services can be found in [2].

AWS provides various tools, both automated and manual, for monitoring. The primary tool is Amazon ‘CloudWatch’ that “collects monitoring and operational data in the form of logs, metrics, and events, providing you with a unified view of AWS resources, applications and services that run on AWS”[3]. As indicated in the Figure 7, Amazon CloudWatch supports the collection of metrics & logs from AWS resources, dashboard and automated alert/alarm based monitoring, automated responses to alarms, and real-time analytics.

[1] https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[2] https://docs.aws.amazon.com/iot/latest/apireference/iot-api.pdf

[3] https://aws.amazon.com/cloudwatch/

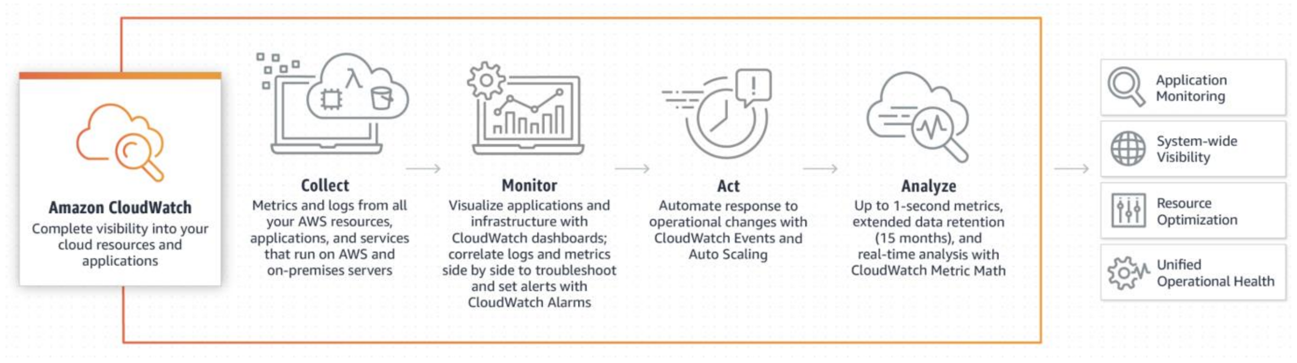

Amazon CloudWatch
Complete visibility into your cloud resources and applications
Collect
Metrics and logs from all your AWS resources, applications, and services that run on AWS and on-premises servers
Monitor
Visualize applications and infrastructure with CloudWatch dashboards; correlate logs and metrics side by side to troubleshoot and set alerts with CloudWatch Alarms
Act
Automate response to operational changes with CloudWatch Events and Auto Scaling
Analyze
Up to 1-second metrics, extended data retention (15 months), and real-time analysis with CloudWatch Metric Math
Application Monitoring
System-wide Visibility
Resource Optimization
Unified Operational Health

FIGURE 7 AWS CLOUDWATCH[4]

CloudWatch alarms can be defined to monitor specific metrics over a period of time and to perform one or more actions based on the value of the metric relative to a threshold. An 'action' is typically a notification sent to an Amazon Simple Notification Service[5] (Amazon SNS) topic. CloudWatch services can be variously accessed through a dedicated console[6], an AWS command line interface[7], or using the CloudWatch APIs [8].

The observable metrics for AWS IoT includes the following[9]:

- Rule & rule action metrics - e.g. the number of messages published on a topic to which a rule is listening, and the success or failure of the actions triggered by the rule;
- Message broker metrics - concerning the number and status of device & client connection requests, the messages published to different topics, and of subscriptions to these topics;
- Device metrics - relating to device control messages (e.g. requests to modify a device's state);
- Device defender metrics - concerning the satisfaction or violation of security constraints.

CloudWatch APIs can be accessed also using SDKs available for various programming languages[10].

2.2.2. AZURE IOT SUITE

The Azure IoT suite is Microsoft's evolving solution for cloud-based IoT management. The suite consists of a collection of services, including:

- Azure IoT Hub[11]: the core services for managing bi-directional (secure) communications between IoT devices and cloud applications;
- Azure IoT Edge[12]: allows for typically cloud-side (and often computationally intensive) processes to be deployed on edge devices - intended in particular for Azure's 'Machine Learning' and 'Cognitive' services (below);
- Azure Sphere: for secure communications with microcontroller (MCU) devices;

[4] https://aws.amazon.com/it/cloudwatch/?nc2=type_a
[5] https://aws.amazon.com/sns/
[6] https://console.aws.amazon.com/cloudwatch/
[7] https://aws.amazon.com/cli/
[8] https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/Welcome.html
[9] https://docs.aws.amazon.com/iot/latest/developerguide/metrics_dimensions.html
[10] https://aws.amazon.com/tools/?nc1=h_ls
[11] https://azure.microsoft.com/en-us/services/iot-hub/
[12] https://azure.microsoft.com/en-us/services/iot-edge/

- Azure Maps: various geo-spatial information services (maps, traffic, routing, etc.);
- Azure Time Series Insight: analytic functions for IoT time-series data;
- Azure Machine Learning (ML) Service: for training and deploying ML processes;
- Azure Cognitive Services: a range of services offering AI capabilities for knowledge, language, speech and vision processing and for searching the internet (see: https://azure.microsoft.com/en-us/services/cognitive-services/);
- Azure Digital Twins: for modelling and reasoning about the physical-spatial relationships between people, places and devices.

For monitoring purposes, the Azure IoT suite includes REST-based APIs with capabilities similar to those described for Amazon CloudWatch (above) - namely, for:

- Observing device metrics (e.g. sensor values);
- Controlling the routing of metric data and diagnostic logs;
- searching events in device activity logs;
- defining and managing rule-based alerts, and corresponding alert-triggered actions;

Details of the monitoring APIs can be found at https://docs.microsoft.com/en-us/rest/api/monitor/, while more general information on the Azure IoT suite is available from https://azure.microsoft.com/en-us/overview/iot/ . SDKs are available for a variety of languages and platforms[13].

Events generated by Azure IoT Hub can be also consumed via Azure Event Grid. Azure Event Grid allows to build applications with event-based architectures. Using Azure Event Grid, it is possible define filters to route specific events to different endpoints, multicast to multiple endpoints, and make sure that events are reliably delivered. Figure 8 shows some of the possible event sources and handlers for Azure Event Grid. WebHook can be used for handling events. The WebHook does not need to be hosted in Azure to handle events. Event Grid only supports HTTPS WebHook endpoints. Event Grid provides SDKs that enable to programmatically manage resources and post events[14].

[13] https://github.com/Azure/azure-iot-sdks

[14] https://docs.microsoft.com/en-us/azure/event-grid/sdk-overview

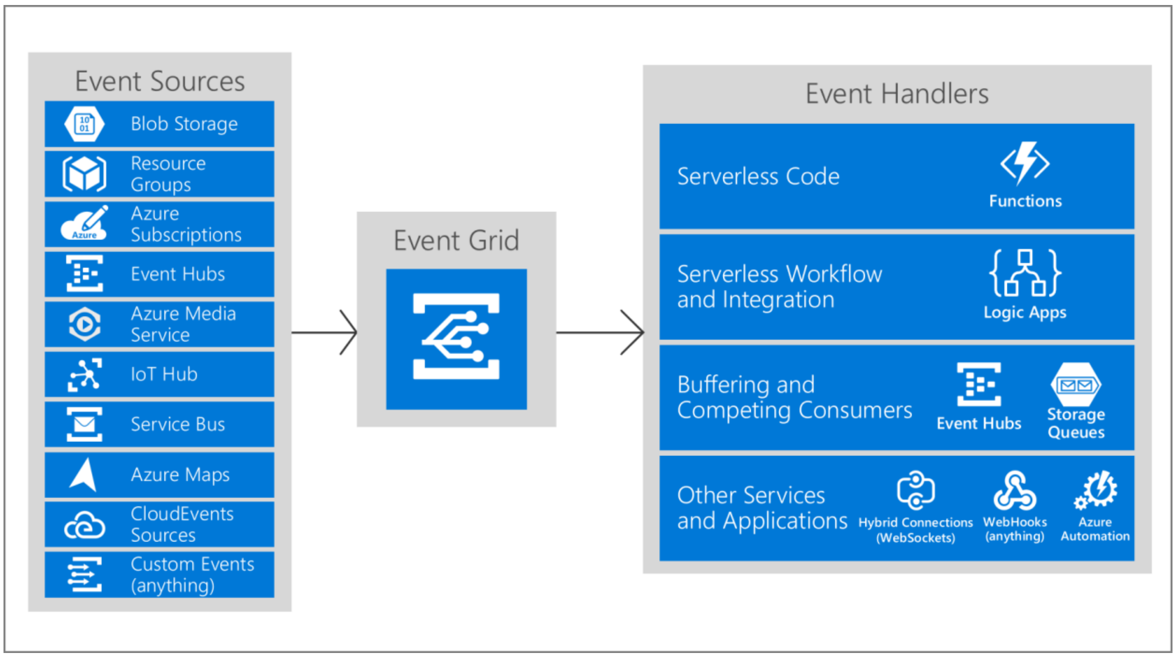


FIGURE 8 AZURE EVENT GRID SOURCES AND HANDLERS

2.2.3. MINDSPHERE

MindSphere[15] is Siemens' cloud-based, open IoT 'operating system' comprising (among other things) various device and enterprise system connectivity protocols and analytics tools in a dedicated development environment (built primarily over Siemens' open Platform-as-a-Service (PaaS) capabilities, but also including access to Amazon's AWS and Microsoft's Azure cloud platforms - see above). The development environment is a browser-based graphical 'work flow editor' (see Figure 9) called 'Visual Flow Editor', that allows users to manage the connections between IoT devices and cloud applications, to create rules and key performance indicators (KPIs), and to define actions (such as email notifications) in case of rule violations (i.e. exceeding KPI thresholds)

[15] https://new.siemens.com/global/en/products/software/mindsphere.html

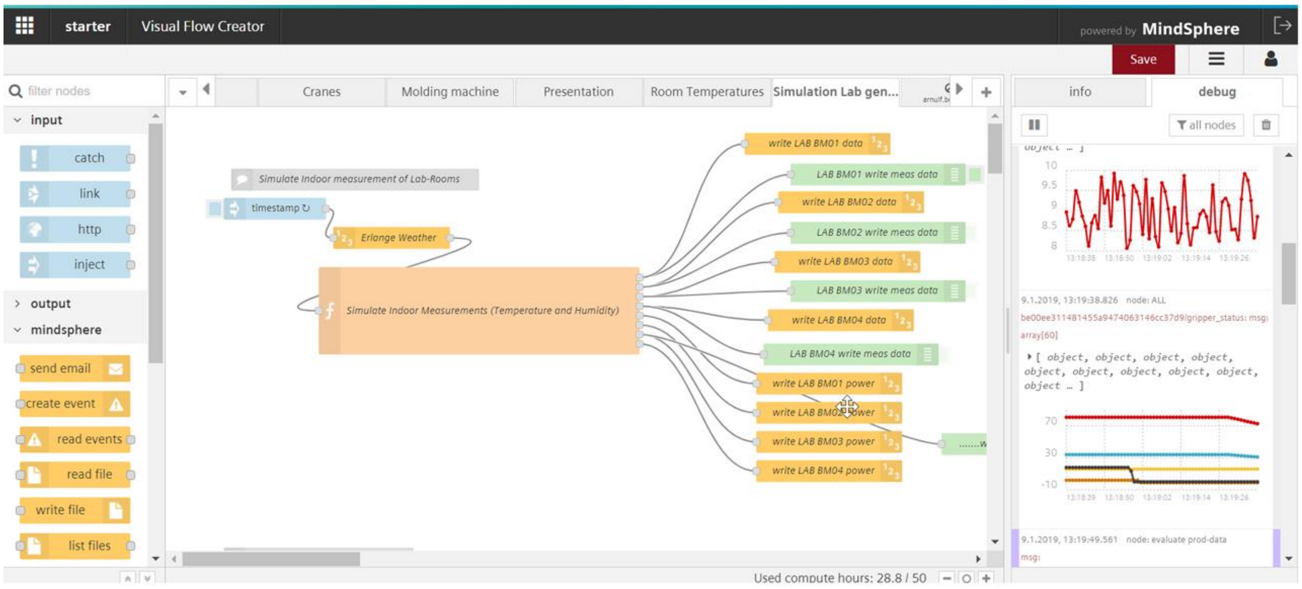


FIGURE 9 MINDSPHERE VISUAL FLOW EDITOR[16]

The Visual Flow Editor is constructed over the MindSphere's services[17], which include APIs for connecting to IoT devices, monitoring events and sending notifications, and for building and executing dataflow[18].

A more limited client-side Java SDK[19] is also offered, which includes IoT device and event management and analytics, but without the dataflow capabilities. The events-related Java clients include (REST-based) APIs for:

- Event Management[20] - supporting the creation, querying and modification of events.
- Event Analytics[21] - which supports the analysis of event logs to identify specific sequences of events, and/or count the number of events matching a specific pattern.

2.2.4. FIWARE ORION CONTEXT BROKER

The FIWARE Orion Context Broker[22] is the reference implementation for the core FIWARE 'Next Generation Service Interface' (NGSI) specification, which defines a REST API for managing the lifecycle (updates, queries, registrations & subscriptions) of "context" information. The term "context" here, originates in the domain of smart-cities[23] and essentially denotes an embedded device, or 'thing' in the IoT sense (e.g. an embedded temperature sensor supplies 'contextual' information, in this case about the temperature of the environment). The Orion

[16] https://www.dex.siemens.com/mindsphere/solution-packages/connect-and-monitor

[17] https://developer.mindsphere.io/apis/index.html

[18] https://developer.mindsphere.io/apis/advanced-dataflowengine/api-dataflowengine-overview.html

[19] https://developer.mindsphere.io/resources/mindsphere-sdk-java/index.html

[20] https://developer.mindsphere.io/resources/mindsphere-sdk-java/apidocs/MindSphere_EventManagement.html

[21] https://developer.mindsphere.io/resources/mindsphere-sdk-java/apidocs/MindSphere_EventAnalytics.html

[22] https://fiware-orion.readthedocs.io/en/master/index.html#welcome-to-orion-context-broker

[23] https://www.fiware.org/community/smart-cities/

Context Broker supports the registration of IoT devices (as 'context entities'), management of their state (through context updates) and push/pull notification of state changes (through context queries & subscriptions).

The FIWARE NGSI (version 2)[24] standard offers no monitoring capabilities beyond the basic ability to query device state and receive device state change notifications.

2.2.5. MONITORING OPENSTACK DEPLOYMENTS

As already described in D3.8 "Network Functions Virtualization for IoT (final)", SEMIoTICS uses OpenStack for the NFV orchestration. As a result, we could integrate monitoring capabilities in SEMIoTICS to ensure availability of resources in this level. In other words, SEMIoTICS can measure that each Virtual Machine (VM) has enough computational resources to keep functioning. Particularly, we can measure the CPU, memory, and disk consumption for each VM using several approaches.

First, there are a number of ways to collect this information from OpenStack. For one, it is possible to rely on the command line interface of the administrator to obtain average values of the computational resources25. However, since this information needs to be transformed in low-level events, it is better to obtain it in a machine-readable format. This can be done by relying on OpenStack APIs, or by installing a software component in the host.

OpenStack has a diverse API through the Compute API[26], which allows users to query diverse metrics for VMs. Thus, it is possible to obtain information on resource consumption for all hypervisors, for one particular hypervisor-level, or even for one particular server. We plan to collect the most fine-grained information, through the OpenStack API, to assess whether there is a potential risk for the availability of the platform[27].

OpenStack deployments generate notifications whenever a significant change of state occurs. Notification messages are published via the message queuing services supporting inter-process communication in OpenStack. RabbitMQ is a typical technology employed to implement message queuing services within OpenStack.

The notifications published by OpenStack can made available to the SEMIoTICS monitoring component implementing an Event Signaler that consumes the messages published by OpenStack, creates the corresponding messages in the SEMIoTICS event format and forward them to the Complex Event Processor (see section 2.1.1). This pattern is followed, for example, by StackTach a tool that can be used for debugging and performance monitoring of OpenStack deployments[28].

2.2.6. LINUX-BASED COMPUTATIONAL RESOURCE MONITORING

In addition to the computational resource monitoring tools provided by Nova in OpenStack, there are also monitoring tools available for in-host to ensure availability of computational resources. These tools can be used within OpenStack VMs, e.g., if the API from NOVA, or the intercepting the events from the RabbitMQ component in OpenStack, is not feasible for any technical reason. Additionally, these approaches have a significant advantage over using only OpenStack because these approaches could also be deployed at the field or backend level to enhance monitoring capabilities.

There are two main approaches for the monitoring of in-host events related to computational resource use. The first is the set of tools called BEATS[29]. BEATS are single-purpose elements that send data to a monitoring

[24] http://fiware.github.io/specifications/ngsiv2/stable/

[25] https://docs.openstack.org/nova/pike/admin/common/nova-show-usage-statistics-for-hosts-instances.html

[26] https://developer.openstack.org/api-ref/compute/

[27] The server diagnostics API can potentially deliver CPU, memory, and networking usage information. The diagnostics API can be found at: https://developer.openstack.org/api-ref/compute/?expanded=show-server-diagnostics-detail

[28] https://stacktach.readthedocs.io/en/latest/intro.html

[29] https://www.elastic.co/products/beats

infrastructure based on ElasticSearch[30]. Specifically, for the case of computational resource monitoring, there is a metric BEAT module measuring CPU, networking and storage information[31]. The second is to monitor the Linux kernel functions to assess the performance of the operating system. This can be done using tools such as SystemTap[32] or by using or implementing tools based on extended Berkley Packet Filter (eBPF) to monitor kernel events[33]. BEATS and the Linux kernel monitoring have advantages and challenges. Particularly, the metrics BEATS component is available for a wide range of operating systems[34], but provides coarse-grained data. On the contrary, monitoring the OS kernel provides specific monitoring capabilities to profile the OS, but it is only available for particular versions of the Linux kernel, and it may require specific versions of the OS.

2.2.7. MONITORING THE NETWORK LAYER

The SEMIoTICS SDN Controller (SSC) represents the centralized intelligence, as a function, in network that possesses the view on mappings of Virtual Tenant Networks, Application Services (formulated as connectivity patterns) to underlying physical topologies, as well as on the device capabilities and resources [11].

The SDN controller used in SEMIoTICS is OpenDaylight (ODL) that provides the user more programmatic control over the infrastructure: managing OpenFlow (OF) capable switches. OF is a communications protocol that empowers a network switch or router to access the forwarding plane over the network.

Using ODL is possible not only to control the resources, but also to monitor and to set some rules. Three components of the high-level architecture of OF[35] are relevant, for the monitoring:

- Statistics Manager is responsible for collecting statistics and status from attached OpenFlow switches and storing them into the operational data store for applications' use.
- Topology Manager is responsible for discovering the OpenFlow topology using Link Layer Discovery Protocol (LLDP) and putting them into the operational data store for applications' use.
- Forwarding Rules Manager is on the "top level" of OpenFlow module, it exposes the OF functionality to controller apps and it provides the app-level API. Its main entity is that manages the OpenFlow switch inventory and the configuration (programming) of flows in switches. It also reconciles user configuration with network state discovered by the OpenFlow plugin.

Cardinal[36] is a plugin that allows ODL to be a monitoring service. Cardinal enables ODL and the underlying SDN to be monitored remotely by the deployed Network Management Systems (NMS) or Analytics suite. NMS is a viable approach to provide the system that monitors and controls remote (and managed) devices located throughout the network, using for example Simple Network Management Protocol (SNMP, the basic protocol).

Cardinal, support SNMP requests, as REST GET, to enable SDN Applications to retrieve ODL diagnostics data.

2.2.8. MONITORING FIELD DEVICES

For the monitoring of field devices, the SEMIoTICS monitoring component will rely on the mechanisms provided by the Web of Things (WoT) architecture[37].

[30] ElasticSearch is available as an open source component here: https://www.elastic.co/products/elasticsearch

[31] https://www.elastic.co/guide/en/beats/metricbeat/current/metricbeat-module-system.html

[32] https://sourceware.org/systemtap/

[33] https://github.com/iovisor/bcc

[34] https://www.elastic.co/support/matrix

[35] https://docs.opendaylight.org/projects/openflowplugin/en/latest/users/architecture.html

[36] https://docs.opendaylight.org/en/stable-fluorine/developer-guide/cardinal_-opendaylight-monitoring-as-a-service.html

[37] https://www.w3.org/TR/2019/CR-wot-architecture-20190516/

A Thing Description is a central building block in the Web of Things (WoT). It describes the metadata and interfaces of Things, where a Thing is an abstraction of a physical or virtual entity that provides interactions to, and participates in, the Web of Things[38].

A Property[39] is a variable of a Web Thing and it represents the internal state of a WoT. The clients can subscribe to Properties to receive a notification message when specific conditions are met (e.g. one or more value changes) and this condition has to be set a priori. In the same way, it is possible to monitor the field devices by subscribing to an event; i.e. monitoring an interval of values.

2.3. Fusion of cross-layer monitoring data

This section will explore the candidate platform enabling the cross-layer data fusion. Each subsection describes a specific platform. Each description will emphasize the pro and cons of its integration within the architecture presented in section 2.1.

2.3.1. PROTON

PROTON[40] (Proactive Technology Online) is an open source complex event processing engine developed also as part of FIWARE. PROTON allows to detect patterns of raw events from various types of data sources (e.g. RESTful services). The PROTON API allows to define custom adapters[41].

Complex events can be determined and processed using a data flow programming paradigm.

Events are processed through event processing networks. The PROTON API allows adding additional custom operators.

2.3.2. APACHE FLINK CEP

Apache Flink[42] is a distributed processing engine for stateful computations over event data streams (both unbounded - i.e. potentially unending - and bounded). Applications are parallelized into possibly thousands of tasks that are distributed and concurrently executed in a cluster - such that an application can leverage virtually unlimited amounts of CPUs, main memory, disk and network IO. Flink requires compute resources in order to execute applications and integrates with common cluster resource managers such as Hadoop YARN, Apache Mesos and Kubernetes. All communications to submit or control applications are via REST calls.

Applications per se are written in Java. The code below is a short ‘hello world’ example[43] that receives a stream of Wikipedia edit events and counts the number of bytes that each user edits within a given (5 second) window of time.

```
public class WikipediaAnalysis {

  public static void main(String[] args) throws Exception {

    StreamExecutionEnvironment see = StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<WikipediaEditEvent> edits = see.addSource(new WikipediaEditsSource());
```

[38] https://www.w3.org/TR/wot-thing-description/#introduction

[39] http://model.webofthings.io/#terminology

[40] https://github.com/ishkin/Proton

[41] https://github.com/ishkin/Proton/tree/master/IBM%20Proactive%20Technology%20Online/ProtonJ2SE/src/com/ibm/hrl/proton/adapters

[42] https://flink.apache.org/flink-architecture.html

[43] https://ci.apache.org/projects/flink/flink-docs-release-1.8/tutorials/datastream_api.html

```
        KeyedStream<WikipediaEditEvent, String> keyedEdits = edits
          .keyBy(new KeySelector<WikipediaEditEvent, String>() {
            @Override
            public String getKey(WikipediaEditEvent event) {
              return event.getUser();
            }
          });

        DataStream<Tuple2<String, Long>> result = keyedEdits
          .timeWindow(Time.seconds(5))
          .fold(new Tuple2<>("", 0L), new FoldFunction<WikipediaEditEvent, Tuple2<String, Long>>() {
            @Override
            public Tuple2<String, Long> fold(Tuple2<String, Long> acc, WikipediaEditEvent event) {
              acc.f0 = event.getUser();
              acc.f1 += event.getByteDiff();
              return acc;
            }
          });

        result.print();

        see.execute();
      }
    }
```

LISTING 1. A JAVA CODE SNIPPET SHOWING THE USE OF THE FLINK CEP LIBRARY

Flink supports complex event processing though the FlinkCEP[44] library. FlinkCEP provides APIs to describe patterns of events and to specify actions to undertake what a sequence of events matches a pattern.

2.4. Events Object Model

This section introduces the object model for the messages produced by the monitoring component (MPD) and consumed either by the MPD internal components or clients of the MPD.

The Monitoring Components produces two types of events:

- Base Event used to represent in a common format the events produced by the various IoT cloud platforms (e.g. Azure, MindSphere) and the various layers (e.g. Network, Filed) of the SEMIoTICS infrastructure. The events of this type are produced by the Event Signalers.
- High Level Event produced by the Complex Event Processor whenever a sequence of events (of type Common Format Event) matches one of the queries submitted to the Monitoring Component

Figure 10 shows the object model for both types of events.

[44] https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html

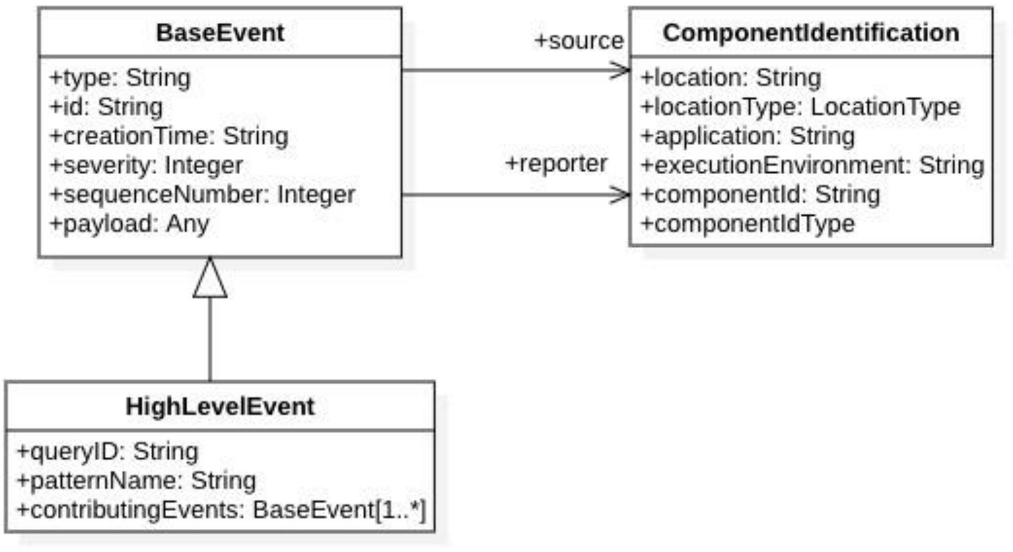


FIGURE 10 SEMIOTICS EVENT OBJECT MODEL

The BaseEvent class describes the events generated by event signalers and defines the following attributes:

TABLE 2. ATTRIBUTES OF THE BASEEVENT CLASS

Attribute	Type	Description
type	String	The type of the event
id	String	The primary identifier for the event.
source	ComponentIdentification	The component in which the original event/action takes place.
reporter	ComponentIdentification	The component that generated the event.
creationTime	String	The date-time when the event generated.
severity	Integer	The perceived severity of the status the event is describing with respect to the application that reports the event.
sequenceNumber	Integer	A source-defined number that allows to identify the order in which events have been generated.
payload	Any	The observed or computed data that correspond to the event.

The ComponentIdentification class describes a component generating or reporting an event:

TABLE 3. ATTRIBUTES OF THE COMPONENTIDENTIFICATION CLASS

Attribute	Type	Description
location	String	Specifies the physical address that correspond to the location of a component.
locationType	LocationType	Specifies the format and meaning of the values in the location property. E.g IPv4, IPv6, hostname, devicename
application	String	The name of the application
executionEnvironment	String	The immediate environment that an application is running in. E.g. MindSphere
componentId	String	Specifies the logical identity of a component
componentIdType	String	Specifies the format and meaning of the values in the compoenentId property.

The HighLevelEvent class describes the events generated by the complex event processor and defines the following attributes:

TABLE 4. ATTRIBUTES OF THE HIGHLEVELEVENT CLASS

Attribute	Type	Description
queryID	String	The identifier of the query that caused the creation of the event
patternName	String	The name of the matched pattern
contributingEvents	BaseEvent[1..n]	Sequence of events matching the pattern

For the representation of both event types (Base and HighLevel) the MPD uses as basis the format being defined by CloudEvents[45]. CloudEvents is an ongoing initiative aimed to define a vendor-neutral specification for defining the format of event in order to ease event declaration and delivery across services, event routers and tracing systems. Current version of the specification is 0.2[46] and a set of SDKs is available for various programming languages (e.g. Java, Python). The current version of CloudEvents specifications define the following attributes[47]:

- type: the type of the event
- specversion: the version of the CloudEvents specification which the event uses.
- source: the event producer.
- id: identifier of the event

[45] https://cloudevents.io

[46] https://github.com/cloudevents/spec

[47] https://github.com/cloudevents/spec/blob/v0.2/spec.md

- time: timestamp of when the event happened
- schemaurl: a link to the schema that the data attribute adheres to
- contenttype: content type of the data attribute value
- data: the event payload

The following table shows the mapping between the information defined by the SEMIoTICS Events Object Model and the CloudEvents format:

TABLE 5. MAPPING BETWEEN THE SEMIOTICS EVENTS AND CLOUDEVENTS FORMAT

CloudEvents	SEMIoTICS
type	type
source	source.location
id	id
time	creationTime

In order to be able to represent all the source and reporter information defined by the Events Object Model the Monitoring Component defines CloudEvent extensions[48].

The values of the attributes of the HighLevelEvent class are transported as payload of a CloudEvent i.e. by means of the data attribute.

2.5. Query Object Model

This section presents the object model of the queries accepted by the Monitoring component and utilized by its client applications to define the high-level events that should be produced as result of the monitoring activity.

[48] https://github.com/cloudevents/spec/blob/master/primer.md#cloudevent-attribute-extensions

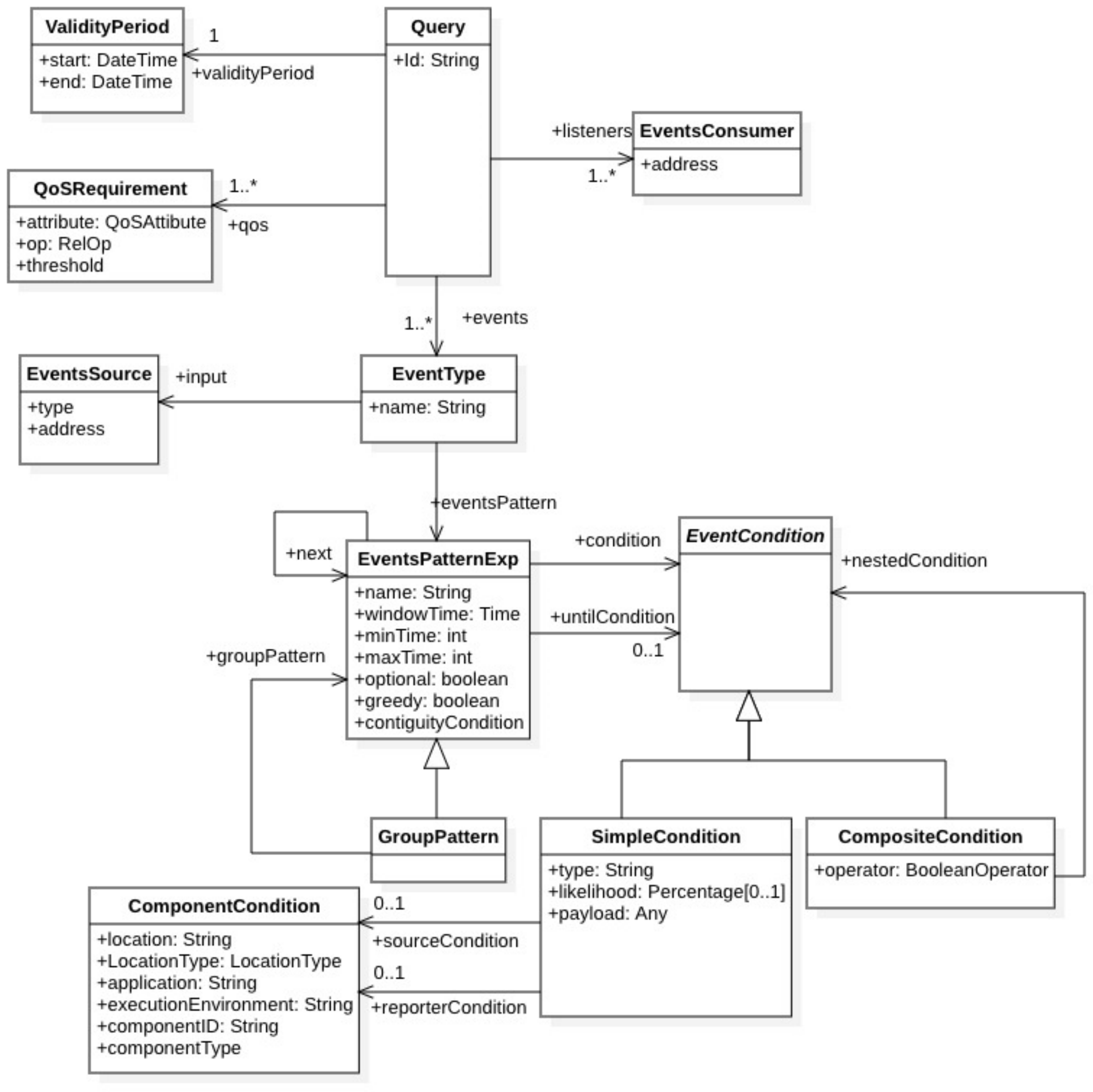


FIGURE 11 QUERIES OBJECT MODEL

A Query object specifies which are the high-level events to be generated and the consumers of those events. A query has a validity period and a set of QoS requirements (e.g. Availability>0).

Attribute	Type	Description
id	String	identifier of the query
validityPeriod	ValidityPeriod	Period of validity of the query

Attribute	Type	Description
qos	QoSExp[1..n]	QoS requirements that should be fulfilled by the execution of the query
eventsPatterns	EventsPatternExp[1..n]	The patterns that have to be matched by the query
listeners	Service[1..n]	Endpoints that should be notified whenever a pattern is matched

A QoS object describes a QoS requirement:

Attribute	Type	Description
attribute	QoSAttribute	The QoS attribute (e.g. Availability)
op	RelOp	relational operator (e.g. <, = , >=)
threshold	Real	threshold value for the attribute

An EventType object describes which is the pattern of events that has to be detected, the source of the events and the name of the type of the generated event.

Attribute	Type	Description
name	String	Name of the type of the event to be generated when the pattern is matched
eventsPattern	EventsPatternExp	the pattern to be matched
input	EventsSource	The source of events to monitor

An EventsPatternExp describes a pattern of events as a list of conditions (simple or composite) that have to be matched, in the order specified in the list, by the events generated by an event source. The conditions in the list are combined according to a contiguity condition. Possible types of contiguity are:

- strict: next matching event is the immediate next of the matched event
- relaxed: interleaving events are allowed between the matched event and the next matching event
- non-deterministic relaxed: interleaving matches are allowed between the matched event and the next matching event
- not-next: the matched event should not be immediately followed by the rest of the pattern
- not-followed-by: the matched event should not be followed by the rest of the pattern

Attribute	Type	Description
name	String	name of the pattern

Attribute	Type	Description
condition	EventCondition	The condition that has to be satisfied by an event
windowTime	Time	The maximum time gap between the first and the last matching event.
minTime	int	Minimum number of events that have to match the condition.
maxTime	int	Maximum number of events that have to match the condition
optional	boolean	This attribute specifies whether the condition if optional for the matching of overall pattern.
greedy	boolean	Specifies whether the condition has to be matched only by a pattern of by as many events as possible.
next	EventsPatternExp	the next condition defining the overall pattern
contiguityCondition	ContiguityCondition	Specify the continuity condition between the event matching the condition and the event matching the next condition in the pattern.
untilCondition	EventCondition	The condition specifying when to stop accepting events in the pattern

An EventCondition object describes a condition that should be matched by an event. There are two possible types of event conditions: SimpleCondition and CompositeCondition.

A SimpleCondition describe the constraints on the properties of an event (see 2.4).

Attribute	Type	Description
type	String	Name of the type of the event
likelihood	Percentage	A percentage representing the likelihood of the event. The value is less than 100% if the event was not directly observed and inferred using the other observations.
payload	Any	The event payload
sourceCondition	ComponentCondition	The conditions on the source of the event
reporterCondition	ComponentCondition	The conditions on the component reporting the event

A CompositeCondition describe a logical combination of EventConditions.

Attribute	Type	Description
operator	LogicalOperator	The combining operator (AND, OR)
nestedCondition	EventCondition[1..n]	The conditions being combined through the operator

The following expression (in a naive/pseudo syntax) illustrates the use of the object model (above):

```
NOTIFY TO http://127.0.0.1/listener
FROM ts=http://127.0.0.1/sensors/temperature/0
EVENTS
    'Cold' IF ts.value<18 && ts.liklihood > 0.5
    'Warm' OTHERWISE
STARTING FROM 01/11/2018 UNTIL 30/03/2019
WITH Availability > 0.5
```

Informally, this expression conveys a request that:

- the monitoring component should notify the client http://127.0.0.1/listener either an event of type 'Cold' if the value of the temperature sensor http://127.0.0.1/sensors/temperature/0 is below 18 with a likelihood of at least 0.5 or an event of type 'Warm' in all the other cases
- the monitoring task should occur from November 1st, 2018 until March 30th, 2019 with an uptime of at least 50%

2.6. Using the MPD Component

This section presents examples of queries accepted by the SEMIoTICS Monitoring, Prediction and Diagnosis (MPD) component.

The MPD component exposes a REST API to its client application. Client applications can submit queries using the `POST /semiotics/api/mdp/queries` command. The MDP listen on port 8080 by default and accepts `application/json` payloads.

2.6.1. A SIMPLE EVENT MAPPING QUERY

The following Query 1 presents one of the simplest form of query that can be submitted to the MPD. Query 1 specifies that the MPD should send an event of type FallEvent to the endpoints `"http://localhost:8181/sara/falldetector"` and `"http://localhost:8080/semiotics/api/mdp/storage/observations"` whenever the MPD receives an event of type `IMUEvent` from the endpoint `"http://localhost:9090/smartphone"` implementing the NGSIv2 (`fiware`) API. The query also specifies that the monitoring task should be active from 15:30 to 16:30 (GMT+1) on March 30, 2020.

```
{    "id":"Q2100",
    "events":[
      {"name":"FallEvent",
        "input":{"fiware":"http://localhost:9090/smartphone"},
        "eventsPattern":{
              "name":"SingleEventPattern",
              "condition":{"type":"IMUEvent"}
              }
          }
      ],
      "listeners":["http://localhost:8181/sara/falldetector",
                  "http://localhost:8080/semiotics/api/mdp/storage/observations"],
      "validityPeriod":{"from":"2020-03-31T10:15:30+01:00", "to":"2007-03-31T10:16:30+01:00" }
}
```

QUERY 1: A TYPE MAPPING QUERY

In general an MPD query should specify at least four key elements: the identifier of the query (`id`), the list of patterns of events that should be detected by the monitoring (`events`), the list of endpoints that should be notified whenever a pattern of events is matched (`listeners`) and, the validity period of the query (`validityPeriod`).

The *events* field is the central element among the fields mentioned above: it allows a client application to specify the type of events that should be emitted by the MPD and the conditions ruling their emission. In particular each object appearing within the *events* list is supposed to have three fields:

- **name**: is the name of the type that will be given to the events emitted by the monitoring component in response to the matching condition defined by the *eventsPattern* field. In the case of Query 1 an event of type `FallEvent` will be emitted whenever an event of type `IMUEvent` is received from the endpoint `"http://localhost:9090/smartphone"`.
- **input**: is an object having a single attribute that specifies the source to monitor for the events referenced by the *eventsPattern* field. In particular, the name of the attribute indicates the API that should be used to interact with that event source whilst the value of the field indicates the url of the event source. In the Query 1 it is specified that the events in the *eventsPattern* field are expected to originate from the source which implements the FIWARE API (i.e. NGSIv2). Other possible API names are `"wot"` for Web of Things event sources, `"restconf"` for event sources implementing the RESTCONF protocol, `"aws"` for Amazon Web Services.
- **eventsPattern**: it is an object defining the condition that should be matched by the MPD against the stream of events generated by the source indicated by the *input* field. An MPD condition is the specification of a set of properties that should be matched by a sequence of events in a stream of events generated by a monitored event source. An object defining a patterns of events contains at least of two fields:
    - **name** is the name of the of events pattern. In the case of Query 1 the name is `SingleEventPattern`.
    - **condition** is the filter condition over the stream of events. In the case of Query 1 the condition is that the events should have the type `IMUEvent.`

Another key element in a MPD query is the *listeners* field: it is a list of strings each one representing a url that should be notified by the MPD whenever the one of the patterns specified by means of the *events* field is matched. In particular the MPD uses a POST operation to notify a client about the occurrence of a matching in a stream of events. In the case of Query 1 the MDP would call the operation `POST "http://localhost:8181/sara/falldetector"` and `POST "http://localhost:8080/semiotics/api/mdp/storage/observations"` with payload:

```
{    "id":"3f802a24-b7f9-4745-b7ee-bfa7274c5c63",
     "source":"/eu.semiotics.monitoring.cep.flink",
     "specversion":"1.0",
     "type":"FallEvent",
     "time":"2020-03-30T15:45:01.459088+01:00",
    "extensionsFormats":[],
    "data":{"sequenceNumber":0,
        "queryId":"Q2100",
        "eventsPatternName":"SingleEventPattern",
        "contributingEvents":[{
            "data":{"sequenceNumber":0,
                    "likelihood":0.0,
                    "sv":"0" },
            "id":"6234b9d5-00d1-4657-9e56-c0daf24944f8",
            "source":"http://localhost:9090/smartphone",
            "specversion":"1.0",
            "type":"IMUEvent",
            "time":"2020-03-30T15:44:58.938719+01:00",
            "extensionsFormats":[]}]
    }
}
```

The semantics of the remaining fields in a MPD query object is pretty straightforward:

- **id**: is a string defined by the client to identify the query
- **validityPeriod**: is an object indicating the validity period of the query by means of a couple of attributes each indicating the date and time when to start and stop the evaluation of the query. Date and time is encoded using an ISO 8601 string.

2.6.2. PUTTING SIMPLE CONDITIONS ON PAYLOADS

A condition within a pattern expression can concern also the payload of the events.

Query 2 presents a query requesting to emit an event of type "Cooking" towards the endpoint "http://localhost:8181/sara/activitydetector" whenever the event source "http://localhost:9090/smarthome/owen" emits an event of type "temperature" having a numeric payload greater than 40.

```
{ "id":"Q2100",
    "events":[
        {"name":"Cooking",
        "input":{"FIWARE":"http://localhost:9090/smarthome/owen"},
        "eventsPattern":{
                "name":"PayloadFilter",
                "condition":{"type":"temperature","v":">40"}
            }
        }
    ],
    "listeners":["http://localhost:8181/sara/activitydetector"],
    "validityPeriod":{"from":"2020-03-31T10:15:30+01:00",
                      "to":"2007-03-31T10:16:30+01:00" }
}
```

QUERY 2: PAYLOAD FILTERING QUERY EXAMPLE

In general conditions over payloads having numeric, string or boolean types can be expressed within a condition object using the *v*, *bv*, and *sv* attributes respectively. In particular the value of the

- *v* attribute is a string starting with the indication of a relational operator followed by the decimal representation of a numeric constant. The condition is satisfied if the numeric payload carried by the event satisfies the expression.
- *bv* attribute is either the “true” or "false" string. The condition is satisfied if the boolean payload carried by the event matches that indicated by the value of the attribute.
- *s* attribute is a string representing a regular expression pattern. The condition is satisfied if the string payload carried by the event matches the regular expression indicated a value of the attribute.

2.6.3. QOS-AWARE QUERIES

The MPD query language supports late binding of event sources, i.e. to delegate to the MPD the selection the selection of the actual source to use for the monitoring task defined by a query. This feature allows a client component to exploit redundancies that could be present within the system to achieve QoS requirements that might exists on the monitoring task (e.g. continuity of the monitoring to support dynamic risk assessment in offshore environments).

Query 3 instructs the MPD to send to `“http://localhost:8181/environmentmonitor”` an event of type `“HighTemperature”` whenever one of the sources of type `“temperatureSensor”` located either within the region `“Room1”` or the region `“Room2”` emits an event of type `“sensorValue”` having a payload greater than `40`. Moreover, the query also requests that the monitoring task must be available 90% of the time period indicated by the *validityPeriod* attribute. This requirement is specified by means of the *qos* attribute of the query.

In this specific case the MPD, and in particular its Controller component, will:

1. Use the service `“http://localhost/location”` implementing the `“fiware”` API to retrieve the list of all available sources of type `“temperatureSensor”` located within the regions `“Room1”` and `“Room2”`.
2. Select from the list of candidate sources from step 1, a specific source allowing to meet the *qos* requirements expressed within the query
3. Derive a set of ancillary queries suitable to monitor the availability of the selected source. The ancillary queries may concern not only the selected source (e.g. monitoring its “heartbeat”) but also other sources (e.g. battery) that are known to cause the unavailability of the selected source. The MPD Controller will use the causal model provided by the Causal Model Identifier component to identify the additional sources that should be monitored.
4. Forward both the primary and ancillary queries to the CEP component.
5. In the case one of the ancillary queries signals the actual or predicted unavailability of the selected source, the MPD controller selects form the candidate lists from step 1, an alternative source allowing to meet the QoS requirements. Once selected the alternative source the Controller instructs the proper signaler to serve the query using the new source in place of the old one.

```
{  "id":4100,
    "events":[
        {"name":"HighAvailabilityTemperature",
        "eventsPattern":{
                "name”:"HighTemperature",
                "condition":{"type":"sensorValue",
                            "payload":">40",
                            “source":{
                                “registry":"http://localhost/location",
                                        "api":"fiware",
                                        "type":"temperatureSensor",
                                        "location":{
                                            "regions":["Room1","Room2"]}
                                    }
                            }
                        }
                }
    ],
    "qos":{
        "availability":">0.9"
    },
    "listeners":["http://localhost:8181/environmentmonitor"],
    "validityPeriod":{  "from":"2020-03-30T10:15:30+01:00",
                        "to":"2020-03-30T10:16:30+01:00" }
}
```

QUERY 3: EXAMPLE OF QOS-AWARE QUERY

In general a QoS-aware query is a query having a QoS requirement and at least one source of events specified by means of a source condition.

QoS requirements are expressed by means of the *qos* attribute. The value of the *qos* attribute is an object having one attribute for each qos requirement where the name of the attribute indicates the quality attribute and the value a string representing the constraint over its value (currently only values having the form *relational operator + threshold value* are supported).

A source condition is expressed by means of the *source* attribute appearing within event condition objects. When specifying a source condition the value of a source attribute may specify:

- *registry*: the url of the resource maintaining an association between events sources and locations (either regions or addresses or geo locations)
- *api*: the api implemented by the registry and to be used to query the registry
- *type*: a string encoding the identifier of the type of event source (e.g. temperatureSensor). Only sources having this type are matched.
- *location*: an object specifying a constraint about the physical location of the source. A location object specifies one of the following alternatives an (optional) list of region names (*regions*), an (optional) *Address*, an (optional) geo reference..
- *organisations*: a sequence of strings each identifying an organizations managing events sources. Only sources owned by one of the organizations are matched.

2.6.4. EVENT PATTERN QUERIES

The main use case for the MPD monitoring component is the detection of specific sequence of events (events pattern) within a stream of events generated by an event source. Query 2 is an example of this class of queries.

Query 4 instructs the MPD component to send an event of type `"UnauthorizedAccessAttempt"` to the REST service listening at `"http://localhost:8181/sara/securityalerts"` whenever the NGSI (`"fiware"`) broker located at `"http://localhost:9090/smartphone"` emits three failed login events (`"/login/failure"`) in a row.

```
{   "id":3100,
    "events":[
        {"name":"UnauthorizedAccessAttempt",
        "input":{"fiware":"http://localhost:9090/smartphone"},
        "eventsPattern":{
                "name":"FirstFailedLogin",
                "condition":{"type":"/login/failure"},
                "next":{
                    "name":"SecondFailedLogin",
                    "condition":{"type":"/login/failure"},
                    "next":{
                        "name":"ThirdFailedLogin",
                        "condition":{"type":"/login/failure"}
                    }
                }
            }
        }
    ],
    "listeners":["http://localhost:8181/sara/securityalerts"],
    "validityPeriod":{  "from":"2020-03-30T10:15:30+01:00",
                        "to":"2020-03-30T10:16:30+01:00" }
}
```

QUERY 4: EVENTS PATTERN QUERY EXAMPLE

A sequence of events is expressed via the nesting of events pattern expressions specified in the "`eventsPattern`" attribute of an event type specification. In the case of Query 2 the nesting (highlighted) is realized via the "`next`" attribute: the expression named `FirstFailedLogin` contains another expression named `SecondFailedLogin` which in turn contains a third expression named `ThirdFailedLogin`. In an events pattern specification the types indicated in the `condition` field specifies the types of the events (`"/login/failure"`) to be detected within the event stream whilst the nesting the order in which they should occur to represent the occurrence of that pattern.

A nesting specified via the "`next`" attribute indicate to the MDP that a pattern is matched if, and only if, the events indicated strictly follow each other without any other interleaving type of event. As an example, in the case of Query 2 the sequence of events `["/login/failure","/light/on","/login/failure","/login/failure"]` would not be recognized as matching the "`ThreeFailedLogin`" pattern because of the occurrence of the "`/light/on`" event between the first and second occurrence of "`/login/failure`".

The "not interleaving" event (strict sequence) constraint can be relaxed by using the "`followedBy`" attribute in place of "`next`".

Based on the capabilities of the CEP Engine (i.e. the Flink CEP) selected for the implementation of the MPD, the MPD Query language provides following attributes to express constraints concerning the occurrence of events within a pattern:

- `next`: the pattern is matched if the events occurs without other interleaving events.
- `followedBy`: the pattern is matched even if the events matching the pattern occur interleaved with other events.

- `followedByAny`: the pattern is matched if the pattern specified by the nesting element is followed any pattern specified by the nested element.
- `notNext`: the pattern is matched if the prefix specified by the nesting element is not immediately followed by the pattern specified by the nested element
- `notFollowedBy`: the pattern is matched if the prefix specified by the nesting element is not followed by the pattern specified by the nested element (this is the relaxed form of `notNext`)

2.6.5. EVENTS PATTERNS WITH TEMPORAL CONSTRAINTS

An MPD client can indicate a temporal constraint concerning the occurrence of the events matching a pattern of events.

Query 5 instructs the MPD component to send an event of type `"UnauthorizedAccessAttempt"` to the REST service listening at `"http://localhost:8181/sara/securityalerts"` whenever the NGSI (`"fiware"`) broker located at `"http://localhost:9090/smartphone"` emits three failed login events (`"/login/failure"`) within a time window of five minutes.

```
{ "id":3100,
   "events":[
        {"name":"ThreeFailedLogin",
        "input":{"fiware":"http://localhost:9090/smartphone"},
        "eventsPattern":{
                "name":"FirstFailedLogin",
                "condition":{"type":"/login/failure"},
                "next":{
                    "name":"SecondFailedLogin",
                    "condition":{"type":"/login/failure"},
                    "next":{
                        "name":"ThirdFailedLogin",
                        "condition":{"type":"/login/failure"}
                    }
                },
                "within":"PT5M"
            }
        }
    ],
    "listeners":["http://localhost:8181/sara/securityalerts"],
    "validityPeriod":{  "from":"2020-03-30T10:15:30+01:00",
                        "to":"2020-03-30T10:16:30+01:00" }
}
```

QUERY 5: EXAMPLE OF QUERY WITH TEMPORAL CONSTRAINT

In general a time constraint concerning pattern of events is specified via the *within* attribute. The value of the *within* attribute is an ISO-8601 string specifying the duration of the temporal window for the occurrence of the pattern: a pattern having a within attribute is matched if the events indicated by the patterns occurs within the time widow specified by the attribute *within*.

2.6.6. CROSS-PLATFORM QUERIES

MPD allow expressing queries against stream of events generated by multiple sources managed by different platforms (cross-platform queries). This is possible using the *source* attribute in the context of events filtering conditions.

Query 6 is an example of cross-platform query requesting to emit an `"AwsConfirmedHighTemperature"` event whenever an event of type `"temperature"`, received from the Fiware Context Broker located at `"https://orion.lab.fiware.org:1026/urn:smartsantander:testbed:357"`, is followed by an event of type `"temperature"` emitted by a source `"https://abc123defghijk.iot.eu-west-3.amazonaws.com/sensor1"` managed by AWS platform.

```
{ "id":4100,
  "events":[
        {"name":"AwsConfirmedHighTemperature",
         "eventsPattern":{
                "name":"FiwareTemperatureEvent",
                "condition":{
                    "type":"temperature",
                    "payload":">40",

"source":{"fiware":"https://orion.lab.fiware.org:1026/urn:smartsantander:testbed:357"}},
                "next":{
                    "name":"AwsTemperatureEvent",
                    "condition":{
                        "type":"temperature",
                        "payload":">40",
                        "source":{"aws":"https://abc123defghijk.iot.eu-west-3.amazonaws.com/sensor1"}
                    }
                }
            }
        }
    ],
    "listeners":["http://localhost:8181/environmentalmonitor"],
    "validityPeriod":{  "from":"2020-03-30T10:15:30+01:00", "to":"2020-03-30T10:16:30+01:00" }
}
```

QUERY 6: CROSS-PLATFORM QUERY EXAMPLE

In general a pattern expression with a *source* attribute (e.g. `"FiwareTemperatureEvent"`) is matched only by events generated by the source specified via value of that attribute. The value of a *source* attribute is an object having a single attribute of which the name indicates the API implemented by the source (e.g. `"fiware"`) and the value (a string) the endpoint of the source (e.g. `"https://orion.lab.fiware.org:1026/urn:smartsantander:testbed:357"`). Hence, cross-platform queries can be formulated by specifying different source values with different api specifications within the various pattern expression in a query.

2.6.7. CROSS-LAYER QUERIES

Cross-Layer queries can be formulated using the same approach used to express cross-platform queries.

Queries 7 is an example of cross layer query requesting to emit an event of type `"InopportuneUseOfOwen"` whenever an event of type `"HighExternalTemperature"` form the Fiware cloud source `"fiware":"https://orion.lab.fiware.org:1026/urn:smartsantander:testbed:357"` is followed by an event of type `"OwenInUse"` from the `"https://homegateway/owen"` source implementing the `"wot"` API (i.e. a Field layer API).

```
{ "id":4100,
    "events":[
        {"name":"InopportuneUseOfOwen",
         "eventsPattern":{
             "name":"HighExternalTemperature",
             "condition":{
                 "type":"temperature",
                 "payload":">40",

"source":{"fiware":"https://orion.lab.fiware.org:1026/urn:smartsantander:testbed:357"}},
             "next":{
                 "name":"OwenInUse",
                 "condition":{
                     "type":"temperature",
                     "payload":">100",
                     "source":{"wot":"https://homegateway/owen"}}
                }
            }
        }
    ],
    "listeners":["http://localhost:8181/activitymonitor"],
    "validityPeriod":{  "from":"2020-03-30T10:15:30+01:00",
                        "to":"2020-03-30T10:16:30+01:00" }
}
```

QUERY 7: CROSS-LAYER QUERY EXAMPLE

2.6.8. QUERY OVER UNCERTAIN EVENTS

MPD Queries may concern not only events captured from external platforms and devices by means of the event signalers, but also events generated by the MPD event predictor that, from this point of view, can be seen as a special kind of signaler. The MPD event predictor uses the causal model inferred from the event streams generated by signalers to predict future events that are likely to occur next. The events generated by the MPD event predictor are characterized by a likelihood measure. The MPD Query language allows filtering events based on their associated likelihood.

Query 8 presents an example of query involving an event generated by the MPD event predictor. In particular the query requests to notify an event of type `"CookingWhileCaregiverNotAtHome"` whenever the prediction of the emission, with probability of 70%, of an event of type `"leaveHome"` from the source `"http://homegateway:9090/notifications/caregiver"` is followed by the emission of an event of type `"switchOn"` from the source `"http://homegateway:9090/owen"`.

```
{ "id":3100,
   "events":[
        {"name":"CookingWhileCaregiverNotAtHome",
        "eventsPattern":{
                "name":"CaregiverNotAtHome",
                "condition":{"type":"leaveHome",
                            "source":{"wot":"http://homegateway:9090/notifications/caregiver"},
                            "likelihood":70.0
                },
                "next":{
                    "name":"OwenTurnedOn",
                    "condition":{"type":"switchOn",
                                "source":{"wot":"http://homegateway:9090/owen"}
                    }
                }
            }
        }
    ],
    "listeners":["http://callcentre:8080/alerts"]
}
```

QUERY 8: QUERY OVER UNCERTAIN EVENTS EXAMPLE

In general events generated by the MPD can be filtered by specifying the *likelihood* attribute within the condition object. The numeric value of the *likelihood* attribute defines the threshold value for the matching of the event (i.e. only events having a *likelihood* greater than the threshold matches the query).

2.6.9. PREDICTION QUERIES

Events pattern may concern not only the specification of the temporal ordering of the events but also the causal relations inferred by the MPD Causal Model Identifier.

Query 9 presents an example of query involving an event predicted by the MPD event predictor. In particular the query requests to notify an event of type `"CookingWhileCaregiverNotAtHome"` whenever an event of type `"leaveHome"` received from the source `"http://homegateway:9090/notifications/caregiver"` allows the MPD to predict, with a likelihood greater than 70%, that an event of type `"switchOn"` will be emitted by the source `"http://homegateway:9090/owen"`.

```
{ "id":3100,
  "events":[
      {"name":"PatientIsGoingToCook",
       "eventsPattern":{
           "name":"CaregiverNotAtHome",
            "condition":{"type":"leaveHome",
                         "source":{"wot":"http://homegateway:9090/notifications/caregiver"}},
            "predicts":{
                "name":"OwenTurnedOn",
                "condition":{"type":"switchOn",
                                "source":{"wot":"http://homegateway:9090/owen"},
                                "likelihood":70.0 }
                    }
                }
            }
        ],
        "listeners":["http://callcentre:8080/alerts"]
}
```

QUERY 9: PREDICTION QUERY EXAMPLE

In general an MPD prediction query is an event pattern query realized via the *predicts* attribute.

2.6.10. DIAGNOSTIC QUERIES

The causal relations inferred by the MPD Causal Model Identifier can also be used to enable queries concerning the causes of events captured by signalers.

Query 10 presents an example of diagnostic query. The query requests to notify an event of type `"RemoveObstacleRequest"` to the endpoint `"http://homegateway:8080/humantasks/notifications"` whenever the causal model identified by the MPD let to infer that an event of type `"advance"` from `"http://sara/robotics/navigation"` is immediately followed by an event of type `"stuck"` from `"http://robotic_assistant"` because, with likelihood 70%, an (undetected) event of type `"touchOn"` from source `"http://robotic_assistant/sensors/touch"`.

```
            { "id":3100,
                "events":[
                    {"name":"RemoveObstacleRequest",
                    "eventsPattern":{
                        "name":"MoveRequest",
                        "condition":{"type":"advance",
                                    "v":">0",
                                    "source":{"fiware":"http://sara/robotics/navigation"}
                            },
                        "next":{
                            "name":"RobotIsStuck",
                            "condition":{"type":"stuck",
                                        "source":{"wot":"http://robotic_assistant"}
                            },
                            "causedBy":{
                                "name":"ObstacleTouched",
                                "condition":{"type":"touchOn",
                                            "source":{"wot":"http://robotic_assistant/sensors/touch"},
                                            "likelihood":70.0
                                            }
                                }
                            }
                        }
                    }
                ],
                "listeners":["http://homegateway:8080/humantasks/notifications"]
            }
```

QUERY 10: DIAGNOSTIC QUERY EXAMPLE

In general an MPD diagnostic query is a query specifying an events pattern having its last element specified via a *causedBy* attribute.

2.7. Translation of SPDI patterns into monitoring policies

SEMIoTICS follows a pattern-driven approach in managing IoT/IIoT deployments, with SPDI properties' verification, as detailed in deliverable D4.1 and its follow-up D4.8. There are 4 verification means that can be used for said SPDI properties: testing, certificate, pattern-based and monitoring. The latter, in specific, is of particular interest in the context of Task 4.2.

In more detail, the Pattern Orchestrator (PO) component receives instantiated Recipes (i.e. definitions of IoT orchestrations) from the Recipe Cooker (RC) component and transforms them to a machine readable format (Drools), then transmitting them to the Pattern Engines at the various layers. In the context of this translation, the PO will be also responsible to match the SPDI properties required to the monitoring capabilities of the specific components selected to instantiate the designed workflows.

A confirmed property is a property that is verified at runtime, through a specific means as defined in the Verification. Verification is a class that describes the way a Property of a Placeholder is verified. The verification process can be done through monitoring, testing, a certificate or via a pattern. Regarding the first option, this means that the existence of a monitoring service allows the verification of the SPDI property of a placeholder activity. Such a monitoring service is materialized by the monitoring components of SEMIoTICS in the three layers. These components are fed with monitoring policies from the PO, as described below, undertaking the verification of the corresponding properties. For example, justify that a service or a device is available at specific time windows if the desirable property is a specific target for availability. The Mean of verification in this case is an interface to the corresponding monitoring component through which the verification can take place.

Figure 12 depicts the classes of Property and Verification in a fragment of the pattern model class diagram. As we can see, class Property has a 1:1 relationship with class Verification and the included VerificationType enumerator shows the four verification processes.

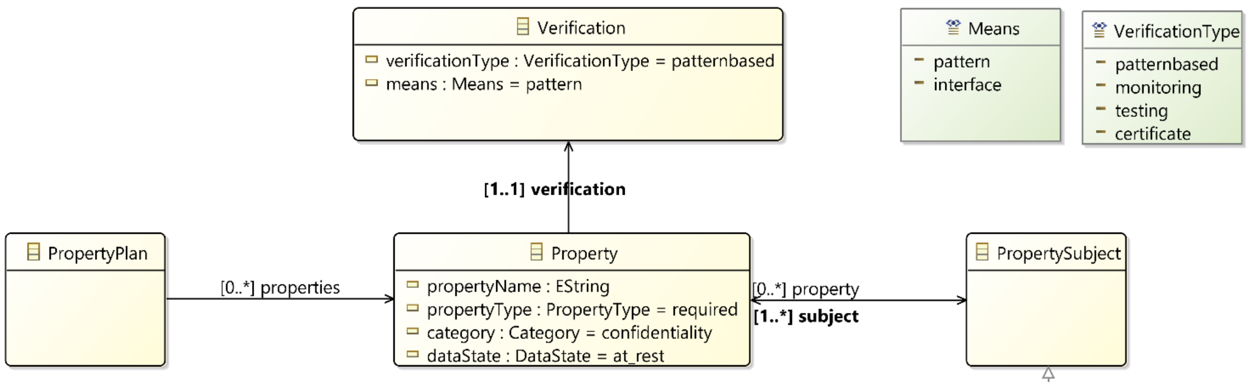
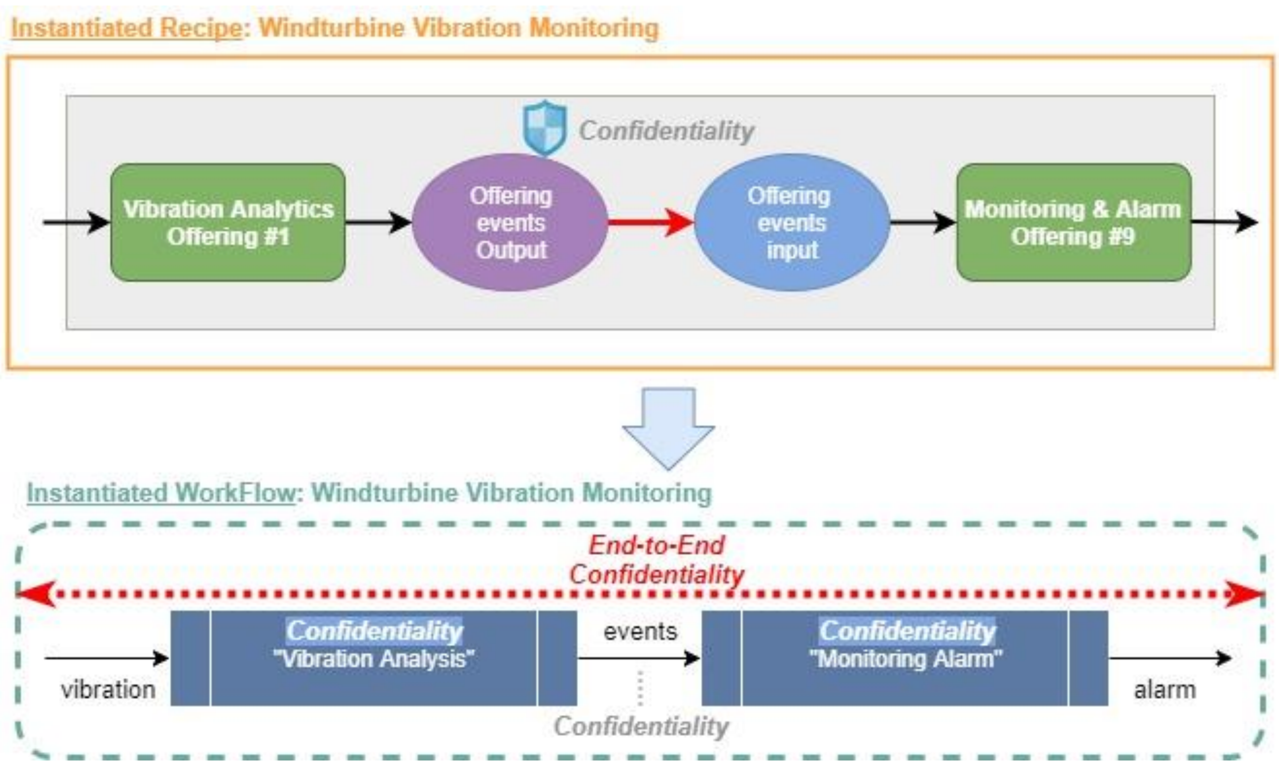


FIGURE 12 PATTERN MODEL CLASS DIAGRAM – PROPERTY VERIFICATION

Following the example that is described in deliverable D4.8 – "SEMIoTICS SPDI Patterns (final)" (see section 6.1), we have the recipe appearing in Figure 13.


Instantiated Recipe: Windturbine Vibration Monitoring
Confidentiality
Vibration Analytics Offering #1
Offering events Output
Offering events input
Monitoring & Alarm Offering #9
Instantiated WorkFlow: Windturbine Vibration Monitoring
End-to-End Confidentiality
Confidentiality "Vibration Analysis"
events
Confidentiality "Monitoring Alarm"
vibration
alarm
Confidentiality
FIGURE 13 INSTANTIATED RECIPE (TOP) AND WORKFLOW (BOTTOM)

The RC translates the above instantiated Recipe to a Pattern Language-compatible representation (as defined in D4.1) before transmitting it to the PO, providing a representation as follows:

1. ORCH "Seq2"
2. Placeholder (Placeholder1, (Vibration Analysis Activity, Vibration Analysis Description))
3. Placeholder (Placeholder2, (Monitoring Alarm Activity, Monitoring Alarm Description))
4. Sequence (Seq, Placeholder1, Placeholder2)
5. Link (Link1, Vibration Analysis, Monitoring Alarm)
6. Property (AP_1, Placeholder1, required, (certificate, interface), confidentiality, in_processing)
7. Property (AP_2, Link1, required, (pattern, "PSPpattern"), confidentiality, in_transit)
8. Property (AP_3, Placeholder2, required, (monitoring, interface), confidentiality, at_rest)
9. Property (OP, "Seq", required, (pattern-based, "PR1"), confidentiality, end_to_end)
10. Pattern rule: (PR1: AP_1, AP_2, AP_3 → OP)

According to the representation of the recipe in question above, all three components of the Sequence "Seq" have a Confidentiality Property. However, each of these properties has a different way to be verified. The existence of a certificate verifies that the property holds for Placeholder1. A pattern takes that role for Link1. Finally, the way to verify that Confidentiality Property holds for Placeholder2 is monitoring. That makes the PO to create the corresponding policy and communicate it to the monitoring component of the corresponding layer.

Taking under consideration all of the SPDI-related metadata available in the involved components' Thing Descriptions (described in deliverable D3.9 – "Bootstrapping and interfacing SEMIoTICS field level devices (final", section 3.4), PO creates policies in the form presented below. As showcased in Query Object Model a policy example could be:

```
{ "id":"Q2121",
    "events":[
        {"name":"Confidentiality",
        "input":{"FIWARE":"http://localhost:9090/sensors/MonitoringAlarm"},
        "eventsPattern":{
                "name":"ConfidentialityEvent",
                "condition":{"placeholder2.confidentiality":"true"}
            }
        }
    ],
    "listeners":["http://localhost:8181/patternEngineField"],
    "validityPeriod":{"from":"2020-03-30T10:15:30+01:00",
                        "to":"2020-03-30T10:16:30+01:00" }
}
```

The above policy checks that the Monitoring Alarm sensor has enabled confidentiality. Any status updates regarding the monitoring of the specific property defined above will have to be transmitted to the Pattern Engine responsible for monitoring the specific component (e.g., Field Pattern Engine if the component is at field layer). The Pattern Engine can then use these incoming monitoring events to reason about the status of the Confidentiality property, which is the focus in this example.

Leveraging these concepts, the implementation of the network signaler that is presented in subsection 5.3 highlights the interactions between monitoring and pattern components in practice.

3. PREDICTIVE MECHANISMS

Modern IoT/IIoT infrastructures typically include thousands of IoT devices which generate a massive amount of data. The capability of data processing and analysis both in near real-time and off-line fashion allows for the discovery of information that has significant impact on the whole infrastructure in terms of security, system's health status, policy guarantees etc., and thus it is crucial to combine edge/fog computing with big data and cloud computing in an efficient manner. Edge/fog computing provides fast near real-time analytics while the plethora of storage and computing resources in the cloud/back-end system can be exploited to carry out computationally intensive tasks. In edge/fog computing scheme data-in-motion (streaming data, time series data etc.) are collected from IoT sources and they are integrated towards highly sophisticated analytics processes that deliver timely decision-making or short-term prediction. Long-term prediction and decision making can be performed at the cloud since data gathered from the lower IoT infrastructure level and stored for computationally intensive and data-hangry tasks. Below, we provide a brief overview of edge/fog-level predictive mechanisms and cloud/back-end oriented predictive algorithms.

3.1. Regression techniques for the prediction at Edge/fog level

In the current section, a general description of the edge/fog-level predictive algorithms is provided. We assume that we collect continuous valued data used within a set of supervised/unsupervised learning framework to predict future outcomes which is a task referred as regression. The simplest regression form can be considered when there is a linear relationship between two variables for example between the quantity measured by an IoT device/sensor and time, and thus we want to estimate a trend of the data points by formulating a model based on existing data. Linear regression is used to fit a straight line usually computed based on linear least squares method. However, in some cases data collected in an edge/fog computing environment may be highly correlated or have collinearity, which can lead the model towards being more susceptible to overfitting. One possible way to alleviate this issue is to use kernel ridge regression [41]. It is a technique that combines the ridge regression model[49] with the kernel trick[50], in order to learn a linear function in the space induced by the kernel and the data.

Support Vector Regression [3] is another method similar to the kernel ridge regression in that it uses the kernel trick, but has some basic differences such as an ε-insensitive loss function (instead of a squared loss function as in linear regression) is used. Support vector regression uses a subset of the training data for inference, since the cost function for building the model ignores any training data not close to the model prediction. Based on various simulations in the literature, support vector regression has been observed that its training procedure is longer as compared to kernel ridge regression, but it is faster at providing predictions due to learning a sparse model. This can be very important in practical applications when we are interested in real-time predictions.

In addition, random forests [6] constitute an ensemble model belonging to the decision-tree class of machine learning algorithms and can also be used for regression by fitting a number of decision trees to various subsamples of the dataset that are then averaged to improve accuracy and reduce overfitting (a process known as bagging). Random forests are also able to use the same model for both regression and classification tasks and they have the ability to learn features that are most important from a set of features from the training set.

Another technique that can be applied for regression/prediction purposes is the Gaussian processes method which computes the outputs probabilistically assuming a Gaussian distribution for approximating a set of functions (processes) in a high-dimensional space [28]. It is assumed that there exists a mapping of independent to dependent variables that cannot be sufficiently captured by a single Gaussian process. Gaussian processes method also uses lazy learning, which delays generalization about training data until after a query has been made which finds a local approximation for each query.

[49] Ridge Regression Model : linear least squares using l2-norm regularization.

[50] Kernel Trick : use of a nonlinear function to transform the data into a higher dimensional space that computations and data modeling can be efficiently performed.

All of the above-mentioned approaches and techniques has been also considered as part of task 4.3 activities. The main outcomes of this task has been reported in D4.10 with a full characterization of the algorithms used in SEMIoTICS at edge device level for the local analytics for specific scenarios events detection.

3.2. Deep Neural Network for prediction at Cloud Level

Cloud-based resources can be exploited towards long-term predictive mechanisms based on historical data saved in the cloud. For that reason, “data-hungry” methods such as deep learning can be applied. In specific, a deep neural network is the core part of the deep learning scheme since has multiple hidden layers between the input and output layers in order to model complicated nonlinear data relationships. Using a large amount of training data (which usually are labeled data) the parameters in a deep neural network can be efficiently and accurately trained to extract complex features from a large amount of data. Recurrent neural networks (RNNs) [16] have been developed to tackle the issue of inter-dependencies between successive samples/data with various length. The input to an RNN consists of both the current sample and the previous observed sample, and thus the output of an RNN at time step t-1 affects the output at time step t. Each neuron is equipped with a feedback loop that returns the current output as an input for the next step, where this structure models each neuron’s internal memory that keeps the information of the computations from the previous input. To train the network, an extension of the backpropagation algorithm, called backpropagation-through-time, is used, where its core concept is a technique called unrolling the RNN, such that we come up with a feed-forward network over time spans. Since the focus is given on cloud-based resources, deeper RNN [26] architectures can be applied to enhance the predictive performance.

Long-short-term-memory networks (LSTMs) [16] is an extension of RNNs. LSTM uses the concept of gates for its units, each computing a value between 0 and 1 based on their input. In addition to a feedback loop to store the information, each neuron in LSTM (also called a memory cell) has a multiplicative forget gate, read gate, and write gate. These gates are introduced to control the access to memory cells and to prevent them from perturbation by irrelevant inputs. An important difference between LSTMs and RNNs is that LSTM units utilize forget gates to actively control the cell states and ensure they do not degrade. The gates can use sigmoid or *tanh()* as their activation function. In fact, these activation functions cause the problem of vanishing gradient during backpropagation in the training phase of other models using them. By learning what data to remember in LSTMs, stored computations in the memory cells are not distorted over time. Backpropagation-through-time is a common method for training the network to minimize the error. When data is characterized by a long dependency in time, LSTM models perform better than RNN models.

3.3. Causal Networks

As stated in section 2.1 one of the objectives of the SEMIoTICS MPD is to generate the high-level business events as defined by client applications and in spite of the dynamicity of the underlying computing infrastructure.

To achieve this objective there is the need of means to predict the effects of, intentional or unintentional, changes on the computing infrastructure used to generate the raw events that are the basis for the events aggregation process performed by the MPD (see section 2.3).

Causal prediction mechanisms are algorithms that rely on causal structures to predict the effects of changes/interventions on a system.

Causal networks are a typical example of causal structure used for that purpose. A Causal Network is a directed acyclic graph in which nodes represent domain variables, edges represents causal relationship between variables (i.e. a change on the “source” variable cause a change in the “sink” variable) [33]. In a Causal Network each node has associated probabilities: prior probability for nodes without incoming edges and conditional probability for nodes with one or more incoming edges [10].

The SEMIoTICS Monitoring Component constructs causal models using prior knowledge of IoT application (e.g., which components are connected and thus, causally affecting which other components). A possible source for this initial knowledge are the SPDI patterns: in the example presented in section 2.6 the pattern ORCH Seq2 allows to derive that there is a causal relation between the “alarm” events generated by the “Monitoring Alarm” process and the events generated by the “Vibration Analysis” process.

Subsequently, since the structure and the parameters of a causal network can be learned from data [29], these models are refined using causal discovery (causal learning) algorithms (e.g., [36] [33]).

During the last years, several algorithms have been proposed to recover causal network from observational data (e.g. [5], [13], [18])

TETRAD[51] [27] is a suite of tools that make efficient causal modeling and discovery (CMD) algorithms from Big Data available on a variety of platforms and environments. The suite uses a common set of CMD algorithms implemented as a Java library. The TETRAD codebase is publically available and it is released under the GNU GPL v. 2 license.

More recently [17] proposed the use of Causal Generative Neural Networks (CGNNs) to learn causal models from observational data.

[1] addresses the challenges related to fast causal inference over event streams and real-time prediction of effects from events.

[38] presents a new method called Event Triggered Causality (ETC) that can determine causal relationships between observed events within time series data from very different sensors.

Because of their focus on causal inference over discrete event streams and Big Data this last two works are particularly relevant from the point of view of the development of the SEMIoTICS Monitoring Component (see section 2.1).

[51] http://www.phil.cmu.edu/tetrad/

4. DIAGNOSIS MECHANISMS

4.1. Computational resource abuse

In the scope of this task, we will use the data sources described in Sections 2.2.5 and/or 2.2.6 to detect possible computational resource abuse scenarios for different components deployed in SEMIoTICS.

First of all, monitoring and processing information about computational resources use should detect naive programming errors such as the case where Spotify was allegedly sending several hundreds of Gigabytes of unrequested data to their users due to a programming mistake[52]. Second, attackers may be inclined to abuse the computational resources from machines to make a financial profit. For example, in February 2018 Google's DoubleClick service was abused by attackers to distribute advertising abusing the computational resources of visitors from visitors obtaining the advertisement to execute crypto-currency mining[53].

As a result, we will explore how to detect when applications are using more computational resources than expected. To this end, we will evaluate the feasibility of an approach based on coarse-grained data, e.g., using information of server-wide CPU and memory consumption, versus monitoring more fine-grained sources of information, e.g., system calls monitoring.

4.2. IoT botnet attack detection based on sparse representation

The technology of IoT has emerged during the last years as a milestone in advancing the concept of Internet networking towards connecting data, users and "things" (usually dubbed as IoT devices, too) in a seamless fashion. IoT technology is based on three pillars: highly heterogeneous and distributed IoT devices data are captured through a gateway and are immediately accessible to a wide range of applications via a secure networking infrastructure. The type of IoT applications span from smart homes, smart cities and wearables to energy management, predictive maintenance, automotive driving, etc. However, the rapidly growing use and realization of IoT-based technology comes at the cost of resolving significant business and technical impediments as reflected in dynamicity, scalability and heterogeneity and end-to-end security/privacy. More specific, a dynamically adaptive behavior is followed at the IoT infrastructure, at the IoT applications and at the IoT devices, and thus there is a great need for promoting a (semi)-automatic behavior within all IoT layers. This gives rise to the pursuit of high scalability properties from the network layers as well as from the IoT infrastructure. In addition, enhanced heterogeneous behavior as a result of the extensive use and interconnection of a large volume of diverse IoT devices should be addressed through the concept of efficient semantic interoperability within IoT applications and platforms. End-to-end security is also a very crucial issue since IoT devices, IoT applications and their enabling platforms could be vulnerable to security attacks.

As a result, it is very crucial to propose a diagnosis mechanism for instant IoT botnet attack detection and the minimization of their impacts by immediate isolation of compromised IoT devices located at the edge of the IoT infrastructure. Due to limited computational capabilities which govern the edge IoT devices, we are strongly interested in providing an algorithmic procedure which uses as small as possible amount of training and testing data towards implementing an accurate IoT botnet attack detector. Next we describe a novel diagnosis technique proposed by FORTH partners under the SEMIoTICS framework [39], where the fundamental assumption is that there is no prior knowledge of malicious IoT network traffic data during the training procedure. The novelty is twofold. Firstly, a reconstruction error thresholding rule based on sparse representation is employed for IoT botnet attack detection assuming that only a very limited amount of both training and testing data is used to deal with low computational constraints as well as with fast reaction. Secondly, a greedy sparse recovery algorithm, dubbed as orthogonal matching pursuit [37], is adopted since it involves tuning of only two hyper-parameters, i.e. the thresholding constant and the sparse representation level.

[52] https://www.telegraph.co.uk/technology/2016/11/11/spotify-bug-killing-hard-drives-with-gigabytes-of-junk-data-user/

[53] https://blog.trendmicro.com/trendlabs-security-intelligence/malvertising-campaign-abuses-googles-doubleclick-to-deliver-cryptocurrency-miners/

Let us assume that statistical features are extracted from IoT traffic data. Usually, the features correspond to statistical metrics reflecting the IoT traffic flow characteristics. Assuming that that IoT network consists of *S* IoT devices, Figure 14 depicts the concatenation process of each features' matrix, where each column corresponds to a feature vector. As it is shown in Figure 14, the focus is given on IoT botnet attack detection at the gateway, where the IoT traffic data is collected and further analyzed in order to detect any malicious behavior originating from a compromised IoT device. As a result, the use of real-data for performance evaluation is of paramount importance. Here, we use the N-BaIoT dataset which corresponds to real IoT traffic data gathered from nine commercial IoT devices and can be found in [54].

The N-BaIoT dataset contains the features extracted from raw IoT network traffic data. More specific, whenever a packet arrives, a behavioral snapshot of the protocols and hosts that transmitted each packet is obtained. Each snapshot corresponds to the packet's contextual information as reflected in 115 statistical features, i.e., the arrival of each packet invokes the extraction of 23 statistical features from five time windows (100ms, 500ms, 1.5sec, 10sec and 1min), and then five 23-dimensional vectors from each window are concatenated into a single 115-dimensional vector (we will use the term *instance* hereafter). During the performance evaluation we use malicious instances obtained during a BASHLITE botnet attack. More specific, we use data based on three categories of BASHLITE attack: (I) Scan: scanning the network for vulnerable devices, (II) Junk: sending spam data, and (III) COMBO: sending spam data and opening a connection to a specified IP address and port. The interested reader is referred to [25] for more details on feature extraction. For the sake of clarity, it is important to notice that the uploaded N-BaIoT dataset1 includes a different amount of benign instances (see Table I) other than mentioned in [25]. As a result, during the performance evaluation we use the benign data captured from eight IoT devices as mentioned in the third column in Table I.

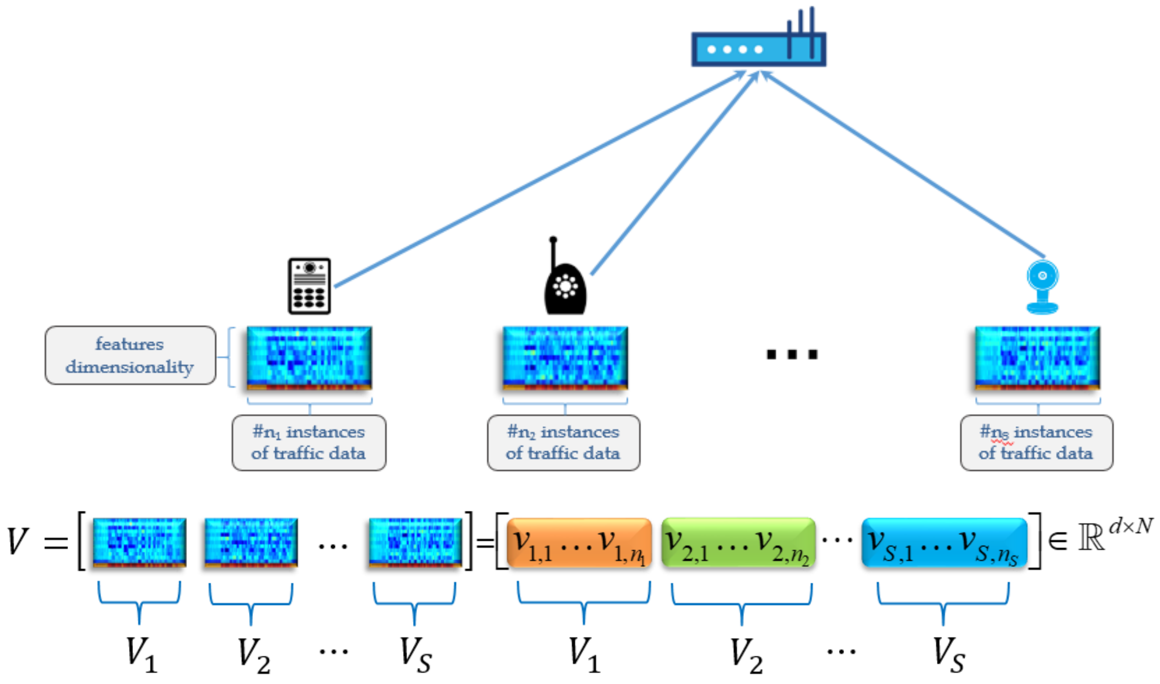

features dimensionality
$\#n_1$ instances of traffic data
$\#n_2$ instances of traffic data
$\#n_S$ instances of traffic data
$V = [\;\cdots\;] = [v_{1,1}\cdots v_{1,n_1}\; v_{2,1}\cdots v_{2,n_2}\cdots v_{S,1}\cdots v_{S,n_S}] \in \mathbb{R}^{d\times N}$
V_1 V_2 $\cdots$ V_S V_1 V_2 $\cdots$ V_S

FIGURE 14 STRUCTURE OF FEATURES EXTRACTED FROM RAW IOT TRAFFIC DAT

[54] http://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT

Device model	Number of benign instances mentioned in [82]	Number of uploaded benign instances
Danmini	49,548	40,395
Ennio	39,100	34,692
Ecobee	13,113	13,111
Philips B120N/10	175,240	160,137
Provision PT-737E	62,154	55,169
Provision PT-838	98,514	91,555
SimpleHome XCS7-1002-WHT	46,585	42,784
SimpleHome XCS7-1003-WHT	19,528	–
Samsung SNH 1011 N	52,150	46,817

TABLE I – COMMERCIAL IOT DEVICES USED TO CAPTURE THE BENIGN INSTANCES. THE THIRD COLUMN CONTAINS THE ACTUAL NUMBER OF UPLOADED BENIGN INSTANCES

Now, let us assume that *S* is the total number of IoT devices in the IoT network, and for each IoT device, a matrix V_i is constructed based on the benign instances extracted from the *i*-th IoT device as

$\boldsymbol{V}_i = [\boldsymbol{v}_{i,1}, \boldsymbol{v}_{i,2}, \cdots, \boldsymbol{v}_{i,n_i}] \in \mathbb{R}^{d\times n_i}, i = 1, \cdots, S$, where the column vector $\boldsymbol{v}_{i,j}$ denotes the j-th d-dimensional instance of the IoT device, and n_i is the number of benign training instances for the *i*-th IoT device. The total number of benign instances is $N = n_1 + n_2 + \cdots + n_S$.

The ultimate goal of the described diagnosis mechanism is to detect whether the observed IoT network traffic data corresponds to benign or malicious behaviour given an observed instance $\boldsymbol{x}_t \in \mathbb{R}^{d\times 1}$. Let us assume that $\boldsymbol{x}_t$ is an instance extracted from the *i*-th IoT device. We are interested in deducing if it is benign, emitted from a “healthy” IoT device, or not. The instance $\boldsymbol{x}_t$ can be expressed as a linear combination of the benign training instances associated with the *i*-th IoT device as seen in Figure 15, where the vector $\boldsymbol{c}_i = \{c_{i,j}\}_{j=1}^{n_i}$ is the vector containing the representation coefficients of $\boldsymbol{x}_t$ in the terms of the columns of $\boldsymbol{V}_i$. The overall data matrix $\boldsymbol{V}$ contains the instances corresponding to the benign data captured from all IoT devices and can be defined as the concatenation of all benign data matrices $\boldsymbol{V}_i, i = 1, \cdots, S$.

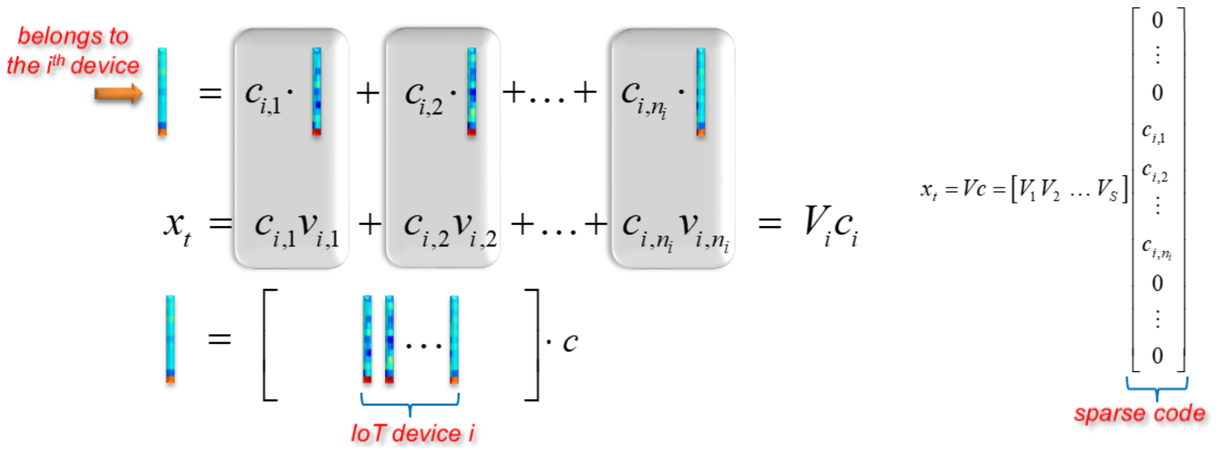

belongs to the i^th device
$= c_{i,1}\cdot + c_{i,2}\cdot + \ldots + c_{i,n_i}\cdot$
$x_t = c_{i,1}v_{i,1} + c_{i,2}v_{i,2} + \ldots + c_{i,n_i}v_{i,n_i} = V_i c_i$
$= [\ \ldots\ ]\cdot c$
IoT device i
$x_t = Vc = [V_1 V_2 \ldots V_S][0 \ldots 0\ c_{i,1}\ c_{i,2} \ldots c_{i,n_i}\ 0 \ldots 0]^T$
sparse code

FIGURE 15 SPARSE REPRESENTATION OF AN OBSERVED IOT TRAFFIC INSTANCE

As it is obvious from the right part of Figure 15, x_t can be sparsely expressed in terms of the overall benign training data matrix $\boldsymbol{V}$, namely $\boldsymbol{x}_t = \boldsymbol{V}\boldsymbol{c}$ with $\boldsymbol{c} = [0, \cdots, 0, c_{i,1}, c_{i,2}, \cdots, c_{i,n_i}, 0, \cdots, 0]^T \in \mathbb{R}^{N\times 1}$ denoting the coefficients vector called sparse code, whose elements are all zero except those associated with the *i*-th IoT device. Given the overall data matrix $\boldsymbol{V}$ and the test instance $\boldsymbol{x}_t$, the following optimization problem can be solved through the orthogonal matching pursuit (OMP) algorithm in order to obtain an estimate of the sparse code $\boldsymbol{c}$

$$\hat{\mathbf{c}} = \arg\min_{\mathbf{c}} \left\| \mathbf{x}_t - \mathbf{V}\mathbf{c} \right\|_2, \ s.t. \ \left\| \mathbf{c} \right\|_0 \leq \tau ,$$

where $\|\cdot\|_2$ denotes the L2-norm, $\|\cdot\|_0$ is the L0-(pseudo)norm which is defined as the number of non-zero elements of a given vector and τ denotes the sparsity level of the estimated sparse code $\hat{\boldsymbol{c}}$. Given $\boldsymbol{x}_t$ and $\boldsymbol{V}$, the sparse code $\boldsymbol{c}$ can be estimated via the orthogonal matching pursuit (OMP) algorithm which is an iterative low-computational constrained method.

The basic assumption is that if the test instance $\boldsymbol{x}_t$ (captured, e.g., from the *i*-th IoT device) corresponds to benign traffic behavior, we expect the reconstruction error $\left\| \boldsymbol{x}_t - \boldsymbol{V}\hat{\boldsymbol{c}} \right\|_2$ to achieve a low value since the indices of the non-zero entries of $\hat{\boldsymbol{c}}$ will correspond to those columns of $\boldsymbol{V}$ associated with the *i*-th IoT device. On the contrary, we expect a high reconstruction error $\left\| \boldsymbol{x}_t - \boldsymbol{V}\hat{\boldsymbol{c}} \right\|_2$ when $\boldsymbol{x}_t$ corresponds to unseen malicious traffic behavior as the estimated sparse code $\hat{\boldsymbol{c}}$cannot be sparsely expressed in terms of $\boldsymbol{V}$, since malicious IoT traffic information is not included in the overall matrix $\boldsymbol{V}$. As a result, the botnet attack detection rule can be written as

$$\text{detection}\left(\mathbf{x}_t\right) = \begin{cases} \text{benign } \mathbf{x}_t, & \text{if } \left\| \mathbf{x}_t - \mathbf{V}\hat{\mathbf{c}} \right\|_2 < \theta \\ \text{malicious } \mathbf{x}_t, & \text{if } \left\| \mathbf{x}_t - \mathbf{V}\hat{\mathbf{c}} \right\|_2 \geq \theta \end{cases} ,$$

where θ is the decision threshold. The decision threshold is estimated given the overall data matrix $\boldsymbol{V}$ containing only benign instances collected from all IoT devices.

It is very important to find the best combination of hyper-parameters τ and θ based only on benign instances collected from all IoT devices. Here, we adopt the concept of proxy outliers to compensate for lacking malicious instances during the threshold's estimation and OMP hyper-parameter tuning. The basic assumption is that if the sparse codes are computed only on benign instances, some of the reconstruction errors might attain large values. As a result, choosing the maximum reconstruction error as the threshold θ to identify botnet attacks could lead in accepting most of the malicious instances as benign. The concept of quartiles comes at the rescue to remove a small amount of proxy outliers (corresponding to large reconstruction errors) present in the benign instances. It is adapted to sparse codes estimation in order to tighten the threshold of the reconstruction error. First, the sparse codes of all benign training instances are computed, and then the reconstruction error of each training instance is estimated. Given the reconstruction errors of all training benign instances, the lower quartile (Q_1), the upper quartile (Q_3) and the interquartile range ($IQR = Q_3 - Q_1$) is computed. An instance $\boldsymbol{x}_t$ is qualified as an outlier of the benign class, if

$$\text{recon.error}\left(\mathbf{x}_t\right) < Q_1 - \rho \cdot IQR \ \ \textbf{OR} \ \ \text{recon.error}\left(\mathbf{x}_t\right) > Q_3 + \rho \cdot IQR ,$$

where ρ is the rejection rate reflecting the percentage of benign instances that are within the non-extreme limits. Based on the previous formula, the extreme values of reconstruction error that represents spurious training instances can be removed and a threshold θ is set as the maximum of the remaining reconstruction errors. The best value of ρ can be found through cross-validation to remove a small fraction of the benign training instances. More details on the main steps of the proposed approach towards estimating the decision threshold based only on benign training instances and tuning the hyper-parameters can be found in our published work in [39].

As it is mentioned above, the main goal of the sparse representation approach is the efficient and fast IoT botnet attack detection. Towards this direction, we examine a real-life scenario using only one test instance $\boldsymbol{x}_t \in \mathbb{R}^{115\times1}$ in order to detect the IoT network traffic behaviour as fast as possible in a reliable manner. Let us consider that $\boldsymbol{x}_t$ can be decomposed into five subvectors of the form $\boldsymbol{x}_t^1, \cdots, \boldsymbol{x}_t^5$ with each subvector $\boldsymbol{x}_t^w \in \mathbb{R}^{23\times 1}$ reflecting the 23 statistical features from five time windows, 100ms (w = 1), 500ms (w = 2), 1.5sec (w = 3), 10sec (w = 4) and 1min (w = 5), respectively. Now, the sparse optimization problem is solved for each subvector $\boldsymbol{x}_t^w$ with$w = 1, \dots, W$, as follows

$$\hat{\mathbf{c}}^w = \arg\min_{\mathbf{c}^w} \left\| \mathbf{x}_t^w - \mathbf{V}^w \mathbf{c}^w \right\|_2, \ s.t. \ \left\| \mathbf{c}^w \right\|_0 \leq \tau^w ,$$

where $\boldsymbol{V}^w \in \mathbb{R}^{23\times N}$ corresponds to the overall benign data matrix of the w-th time window, and we end up with a set of five sparse codes $\hat{\boldsymbol{c}}^1, \dots, \hat{\boldsymbol{c}}^5$. Next, five reconstruction errors of the form $\left\| \boldsymbol{x}_t^w - \boldsymbol{V}^w \hat{\boldsymbol{c}}^w \right\|_2$ (for $w = 1, \dots, W$) are computed leading to five decision functions of the form $detection\left(\boldsymbol{V}^w \hat{\boldsymbol{c}}^w\right)$. The final decision about the existence or not of a botnet attack detection is given via a majority voting scheme which acts as an ensemble learning type of algorithm. It is obvious that a different decision threshold θ^w is separately computed for each matrix $\boldsymbol{V}^w$.

In this section, the IoT botnet attack detection performance of the proposed sparse representation (SR) method based on majority voting is compared against a single hidden layer autoencoder (AE), where the N-BaIoT dataset (see Table I) was used during the evaluation process. For each IoT device we randomly select 100, 300 and 500 benign instances from the first half of each dataset to estimate the decision threshold and perform the tuning of the hyper-parameters following a 3-fold (CV = 3) cross-validation process, where τ is varied from $T = \{5,10,15,20,30\}$ and ρ is varied from $P = \{0.01,0.5,1,2,3\}$. For the AE hyper-parameters tuning we followed a similar strategy based on an AE reconstruction error-oriented decision threshold estimation and hyper-parameters tuning, where the number of epochs is fixed and equal to 50, while the number of nodes in the hidden layer is varied from $\{20,30,40,50,60\}$. As a result, both SR and AE have one hyper-parameter, the sparsity level and the number of nodes, respectively.

Here, an off-the-shelf AE implementation (http://www.mathworks.com/help/nnet/ref/trainautoencoder.html) was used, where '*KerneScale*' parameter was set to '*auto*' and '*Standardize*' to '*true*', while the rest of the parameters were kept to default values.

The evaluation results on IoT botnet attack detection are reported in the form of a confusion matrix as shown in Table II, where TP indicates the quantity of malicious instances correctly detected, TN shows the quantity of benign instances correctly detected, FN indicates the quantity of malicious instances incorrectly detected, and FP denotes the quantity of benign instances incorrectly detected. Here, we calculated the following metrics based on the confusion matrix in order to assess the performance of the proposed framework: (I) Positive Predictive Value (PPV) which indicates the proportion of correctly detected malicious instances in the total instances detected as malicious, (II) Sensitivity (detection rate) which shows the proportion of correctly detected malicious instances in the total number of actual malicious instances, (III) F1-score corresponding to the harmonic mean of PPV and sensitivity, (IV) Accuracy (ACC) denoting the fraction of correctly detected instances in total detected instances.

Detected / Actual	Malicious	Benign
Malicious	True Positive (TP)	False Negative (FN)
Benign	False Positive (FP)	True Negative (TN)

TABLE II – CONFUSION MATRIX CORRESPONDING TO THE EVALUATION RESULTS ON IOT BOTNET ATTACK DETECTION

To evaluate the performance of the two methods, we performed five Monte Carlo runs. During each Monte Carlo run we followed a leave-one-out-device-out cross validation (LOOCV) strategy, where benign instances from *S-1* IoT devices were used for tuning and threshold estimation, while the current (under testing) IoT device's benign and malicious instances were used for testing/evaluation purposes. This procedure was repeated *S* times and the total average performance metrics over all IoT devices and all Monte Carlo runs are reported. This evaluation is IoT device independent and shows the generalization capabilities as the IoT device which is being tested is not included in the tuning procedure.

During the evaluation process, we used 100, 300 and 500 left-out benign instances (see LOOCV description in the previous paragraph), respectively, for testing as well as 200 malicious instances randomly selected from each IoT device's COMBO malicious dataset (1600 malicious testing instances in total). In the case of Junk and Scan botnet attack we used 200, 600 and 1000 randomly selected instances from each IoT device's malicious Junk and Scan dataset (1600, 4800 and 8000 malicious testing instances in total during each evaluation scenario). It is important to notice that we used the malicious instances obtained from the eight IoT devices used during the tuning process (see Table I). Figure 16 shows the results corresponding to the Scan botnet attack, Figure 17 depicts the performance in the case of Junk botnet attack and Figure 18 corresponds to the COMBO botnet attack results. In all figures, the subscripts in the legend names indicate the number of benign instances per IoT device used during the hyper-parameters tuning and the decision threshold estimation process. The vertical black lines indicate the error bars since each experimental scenario is performed five (Monte Carlo runs) by *S* = *8* (total number of IoT devices) times.

It is obvious that the proposed SR method achieves superior performance in light of Sensitivity, F1-score and ACC, while the AE technique achieves slightly better results in terms of PPV. That means that SR is robust in accurately detecting both malicious and normal behavior in the IoT network (the Sensitivity, F1-score and ACC error bars corresponding to AE are wider as compared to the SR method's error bars). Besides, the time complexity between SR and AE is comparable and low (due to space limitation, a more thorough computation cost investigation will be provided in a future publication), and thus SR can be applied for accurate and fast IoT botnet attack detection.

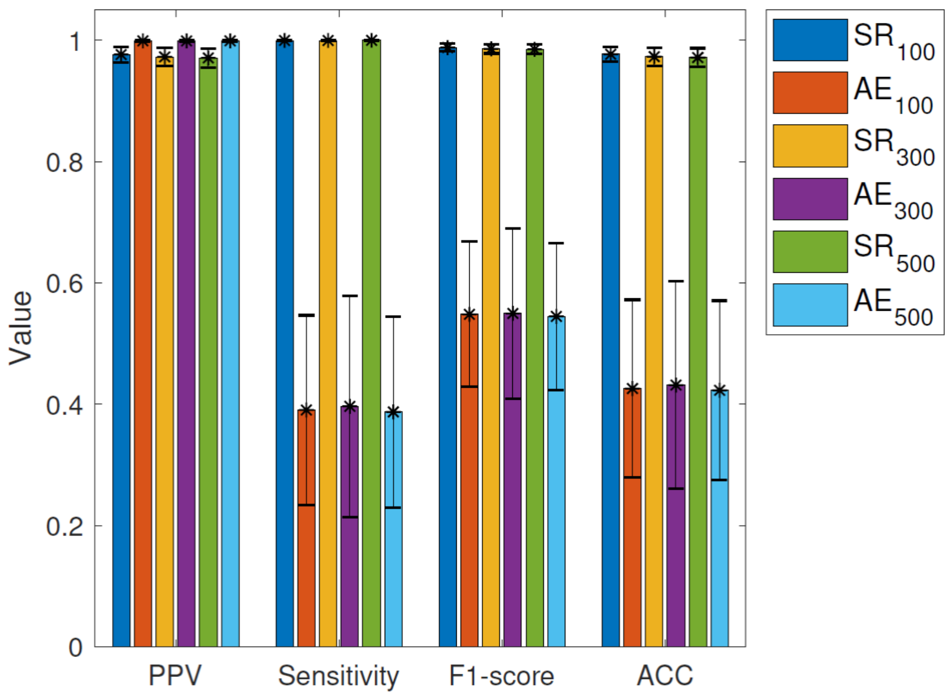


FIGURE 16 PERFORMANCE EVALUATION RESULTS FOR SCAN BOTNET ATTACK

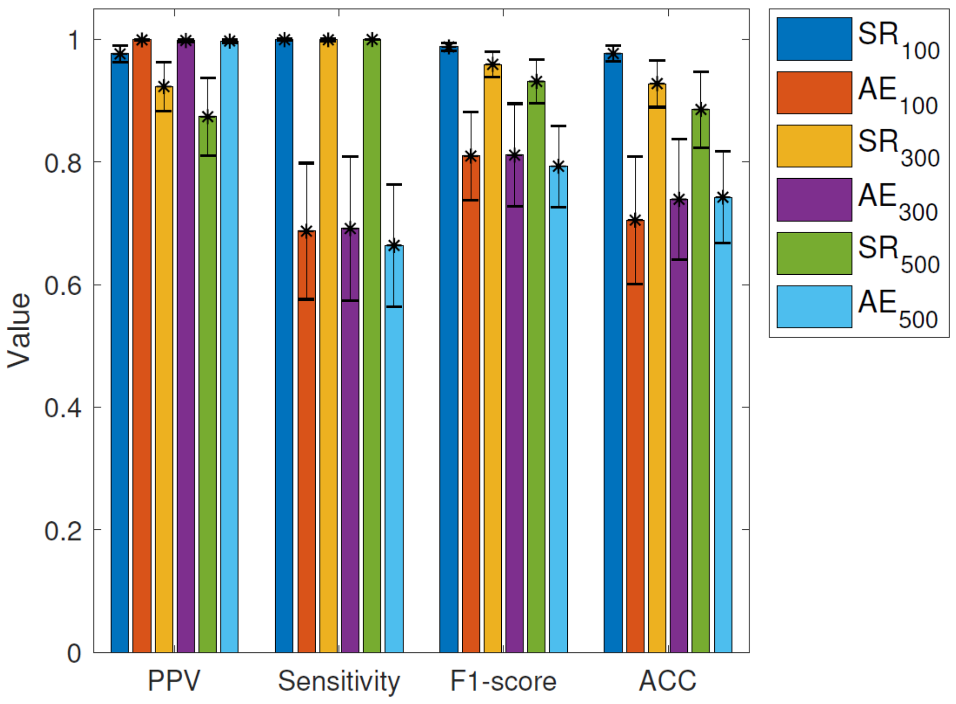


FIGURE 17 PERFORMANCE EVALUATION RESULTS FOR COMBO BOTNET ATTACK

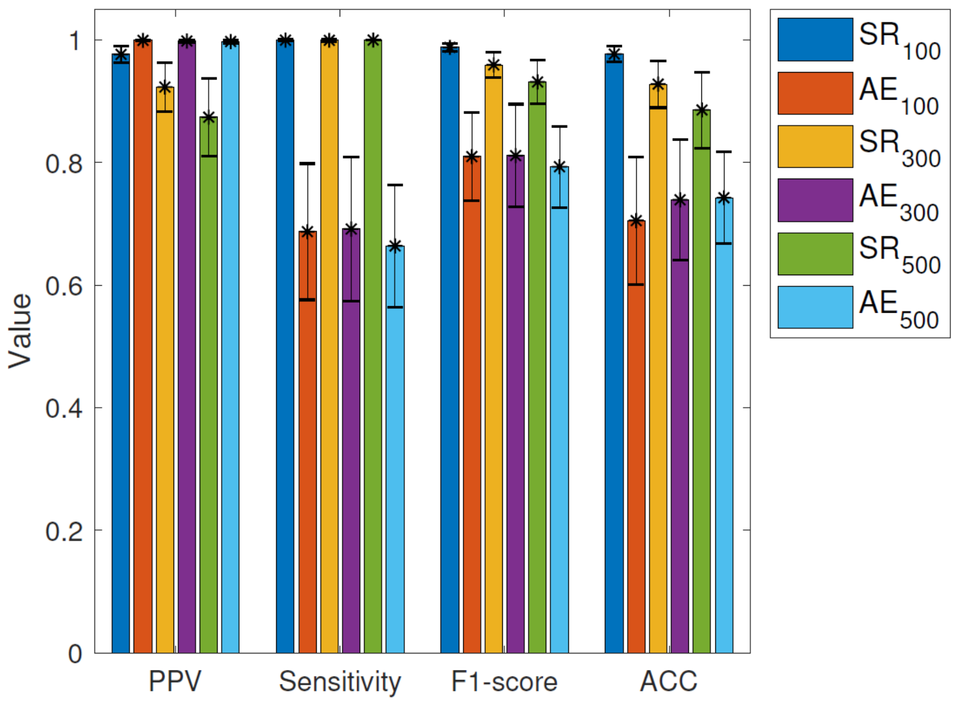


FIGURE 18 PERFORMANCE EVALUATION RESULTS FOR COMBO BOTNET ATTACK

4.3. Anomaly detection using Long Short-Term Memory Recurrent Networks

In many tasks, prediction is dependent on past samples such that, in addition to classifying individual samples, we also need to analyze the sequences of inputs. In such applications, a feed-forward neural network is not applicable since it assumes no dependency between input and output layers. Recurrent neural networks (RNNs) have been developed to address this issue in sequential (e.g., speech or text) or time-series problems (sensor data) with various length.

RNN is a deep learning architecture of an artificial neural network where connections between units form a directed circle. Thus, it can be seen as multiple copies of the same network each passing a message to a successor, giving RNN the ability to connect previous information to the current task.

The input to an RNN consists of both the current sample and the previous observed sample. In other words, the output of an RNN at time step t−1 affects the output at time step t. Each neuron is equipped with a feedback loop that returns the current output as an input for the next step. This structure can be expressed in such way that each neuron in an RNN has an internal memory that keeps the information of the computations from the previous input. To train the network, an extension of the backpropagation algorithm, called Backpropagation Through Time (BPTT) [42], is used. Due to the existence of cycles on the neurons, we cannot use the standard backpropagation here that is used in conventional neural networks, since it works based on error derivation with respect to the weight in their upper layer, while we do not have a stacked layer model in RNNs. The core of BPTT algorithm is a technique called unrolling the RNN, such that we come up with a feed-forward network over time spans. Figure 19 depicts the structure of an RNN and unrolled concept.

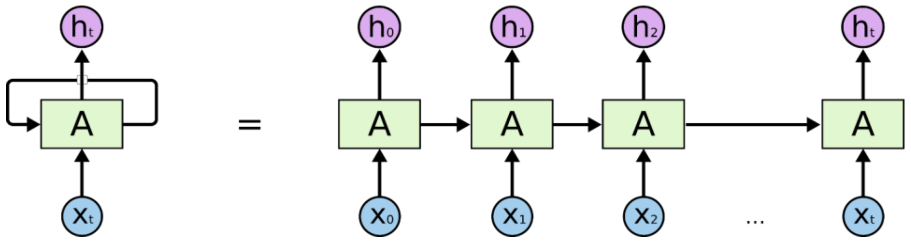


FIGURE 19 RNN UNROLLED- A TAKES AN INPUT XT AND OUTPUTS A VALUE HT [51]

But what if what if we need to "remember" information further back forming a longer dependency, see Figure 19. In theory, RNNs are capable of handling such "long-term dependencies" but in practice, practice they are not capable of modeling such type of dependencies. The problem was explored in depth by Hockreiter [19] and Bengio [4] and a solution was introduced by the former in 1997 that was able of overcoming this problem; the long-short-term-memory (LSTM) network.

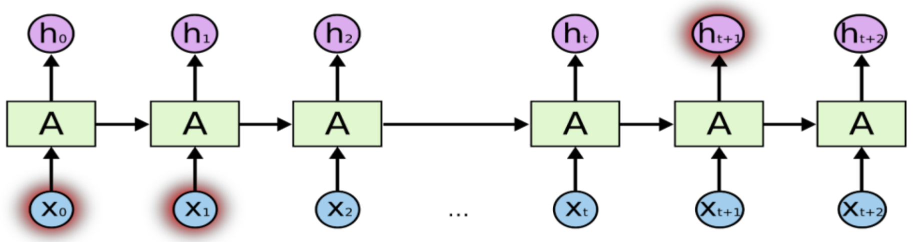
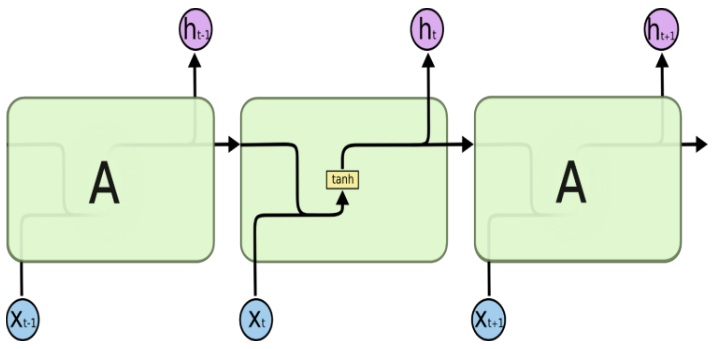


FIGURE 20 LONG DEPENDENCY PROBLEM [51]

LSTM uses the concept of gates for its units, each computing a value between 0 and 1 based on their input. In addition to a feedback loop to store the information, each neuron in LSTM (also called a memory cell) has a multiplicative forget gate, read gate, and write gate. These gates are introduced to control the access to memory cells and to prevent them from perturbation by irrelevant inputs. When the forget gate is active, the neuron writes its data into itself. When the forget gate is turned off by sending a 0, the neuron forgets its last content. When the write gate is set to 1, other connected neurons can write to that neuron. If the read gate is set to 1, the connected neurons can read the content of the neuron. Figure 20 depicts this structure. An important difference of LSTMs compared to RNNs is that LSTM units utilize forget gates to actively control the cell states and ensure they do not degrade. The gates can use sigmoid or *tanh()* as their activation function. In fact, these activation functions cause the problem of vanishing gradient during backpropagation in the training phase of other models using them. By learning what data to remember in LSTMs, stored computations in the memory cells are not distorted over time. BPTT is a common method for training the network to minimize the error. The architectural difference of RNN and LSTM can be depicted in Figure 21.



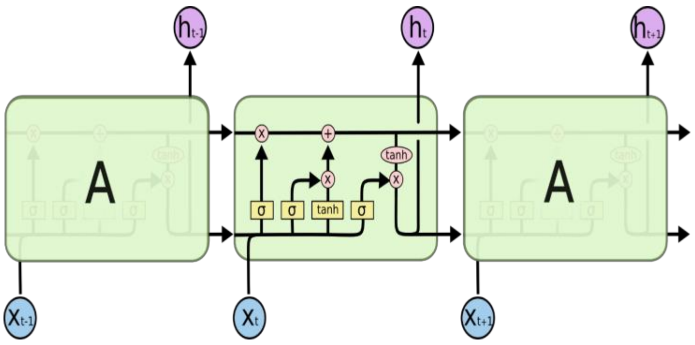


FIGURE 21 RNN VS LSTM [9]

The application of LSTM for intrusion detection is proposed by Ralf C. Staudemeyer [34], where they model the KDD Cup 99 challenge dataset as time series data to train a LSTM network in a supervised manner; outperforming all other algorithms [32] used in the challenge. They also found that because LSTM can look back in time and correlate with past information they excel when training to identify high frequency attacks (e.g., DoS attacks and network probes) as these traffics generate a high volume of successive connections. Another study that supports and outperform the above is this of Jihyun Kim et al [20] that also trained upon the KDD Cup 1999 dataset but improve its model performance by fine-tuning its hyperparameters (e.g., learning rate, number of hidden layers etc.). A different application of the LSTM using the RMSprop optimizer trained the model on the more modern CIDDS-001 dataset and achieved reasonable results performing better than traditional support vector machine (SVM), multilayer perceptron (MLP), and Naïve Bayes techniques for a multi-classification problem [2]. Finally, LSTM have also been proved to be highly effective when employed in an unsupervised manner with the authors of this study [14] combining it with Support Vector Data Description (SVDD) and One Class Support Vector Machines (OC-SVM) algorithms obtaining great results against conventional methods over real and simulated datasets.

Implementing the LSTM architecture for intrusion detection can utilizing Keras[55] a high-level neural networks API, written in Python and capable of running on top of Google's TensorFlow[56]. Also, for expediting the training process we could use a workstation running CUDA framework for GPU acceleration [7][35]. Finally, we should use libraries that enable pre-processing, include evaluation metrics and provide visualization means, such as Skikit-learn, Pandas, Numpy and matplotlib. In terms of the model's configuration and inspired from the aforementioned studies, we should consider training the model with various hyper parameter values (e.g., learning rates of 0.01,0.1, 0.5), using different optimizers (e.g., Adam [21], rmsporop, SGD [31]) and loss functions depending on the task (i.e., binary cross entropy, categorical cross entropy [8]). Finally, considering we are employing a supervised approach we need data to train such model, thus we should examine specific datasets such as CIDDS-001 [40] and NSL-KDD [30].

To summarize, given the work already done in the field we believe that LSTM is a very promising approach for intrusion detection in the scope of SEMIoTICS.

4.4. Anomaly detection based on Generative Adversarial Networks

First, we provide a brief overview of the described generative adversarial networks (GANs) methodology for anomaly detection (in the context of the previous deliverable 4.2). In specific, diagnosis mechanisms (e.g. time-series anomaly detection [23], attack detection [43]) can be performed using the concept of GANs [15], which constitutes an algorithmic procedure based on two neural networks, namely the generative and discriminative networks, working together to produce synthetic and high-quality data. The former network (dubbed as the generator) is in charge of generating new data after it learns the data distribution from a training dataset. The latter network (termed as the discriminator) performs discrimination between real data (coming from training data) and

[55] https://keras.io

[56] https://www.tensorflow.org/

fake input data (coming from the generator). The generative network is optimized to produce input data that is deceiving for the discriminator (i.e., data that the discriminator cannot easily distinguish whether it is fake or real). In other words, the generative network is competing with an adversary discriminative network. The objective function in GANs is based on minimax games, such that one network tries to maximize the value function and the other network wants to minimize it. In each step of this imaginary game, the generator, willing to fool the discriminator, plays by producing a sample data from random noise. On the other hand, the discriminator receives several real data examples from the training set along with the samples from the generator. Its task is then to discriminate real and fake data. The discriminator is considered to perform satisfactorily if its classifications are correct. The generator also is performing well if its examples have fooled the discriminator. Both discriminator and generator parameters then are updated to be ready for the next round of the game. The discriminator's output helps the generator to optimize its generated data for the next round.

Second, it is important to clarify that the ultimate goal regarding the anomaly detection module (within the monitoring and diagnosis framework as described in the current deliverable) is to provide an efficient and robust algorithmic solution towards an IoT botnet attack detector. As a result, it is crucial to apply the botnet attack detection in terms of the lower levels of the SEMIoTICS infrastructure, i.e., at the edge, guaranteeing that the botnet attack detection will be accomplished as fast as possible in order to avoid further intrusion of the whole system until the upper levels, avoid receiving malicious/non-meaningful data to the cloud etc. Thus, we decide that a GAN-based solution should not be taken into consideration towards a further investigation/implementation since it has a deep neural network architecture, and as a result a lot of data is needed a-priori for training purposes. In addition, GANs are highly sensitive to the hyper-parameter selection as well as the convergence of a GAN's objective function suffers both from the presence of a zero real part of the Jacobian matrix and the eigenvalues have large imaginary parts, i.e., facing further instability issues [44].

4.5. Visualization for the Diagnosis

This section describes how Graphical User Interface (GUI) will be used to give meaningful insights into the platform. Visualization can be helpful when it comes to monitoring, as by giving insights, it can ease development/debugging of infrastructure behavior, as well as positively affect end-user experience.

A lot of different kinds of widgets can be used for the diagnosis. Here are few of them:

- Graphs showing statistics of the processed events
- Monitors of SPDI patterns abuses
- Lists of alarms filtered by the priority levels of issues [warning/minor/major/critical]
- Lists of reconfiguration commands for the particular component / part of the framework
- Computing resources monitors

The necessary monitoring graphs and meaningful dashboards will be created based on generic widgets which are to be delivered. Assuming that the components will give the API to the last portion of events, the layer of presentation can be based on these data. That means the additional storage of historical data is required for each of components.

There are some tools describe above that already give insights into to the cloud platform or to Kubernetes cluster itself such us AWS Cloudwatch, Azure IoT Suite, MindSphere tools or Kubernetes Web UI Dashboard[57]. The set of necessary widgets, and therefore set of suitable tools, is to be established based on:

- what information will be exposed by Monitoring Component
- what will be generally visualized within the framework
- what are use case specific requirements.

After gaining all requirements, at least one of the following approaches is to be selected:

[57] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

- GUI that communicates through the API with an external application
- GUI that loads the view itself from the external application
- GUI dedicated to a given backend application.

More information about GUI approach is to be found in deliverable D4.13.

5. Implementation aspects

This section contains the description of the most relevant implementation aspects of the various modules of the MPD.

5.1. Controller

The MPD controller is implemented in the Java programming language and implements three interfaces:

- *Query Processor* offering the operations that allows a client application to submit, check status and cancel a query. This interface is available to client applications as a REST API with endpoint "`/semiotics/api/mdp/queries`". Each query received by means of this interface is processed by the controller and produce a corresponding monitoring task. A monitoring task keeps, along with the original query, additional information derived by the controller and relevant for the processing of the query. Examples of this additional information include the ancillary queries serving the monitoring of the QoS expressed in the query and the current bindings for the variables present in the query.
- *Signaler Manager* offering the operations that allows a signaler to register and declare the kind of API it is able to manage (e.g. Fiware NGSIv2). The controller keeps an internal index of available signalers that the controller uses to decide whether a query is processable i.e. whether for each API indicated in a query a corresponding signaler is available.
- *Event Consumer* offering the operation that allow the controller itself to receive the events produced by the Complex Event Processor and the Events Predictor. The events received by means of this interface are used by the Controller to trigger the adaptation actions (e.g. re-binding of a sensor) by the Controller.

5.2. FIWARE Signaler

The FIWARE Signaler is implemented as a Java library that allows the MPD to interact with context brokers by providing the NGSIv2 API (e.g. the Orion Context Broker or the CIoE-IoT Gateway).

The FIWARE Signaler implements the *Signaler* interface offering the operations to read, write or subscribe attributes of NGSIv2 entities. Moreover, it allows the MPD controller to retrieve the possible bindings for a variable appearing in a Query (e.g. get all entities with type IMU from mobile device).

The FIWARE Signaler transforms the resource observation requests received via the signaler interface into proper NGSIv2 subscriptions and, for each observed resource, keeps the list of clients observing that resource. If the signaler receives an observation request for a resource already being observed the new observer is simply added to the list of observers of that resource without creating a new notification toward the context broker.

5.3. Network Signaler

The network signaler is responsible for monitoring the status of the network topology of the SEMIoTICS use cases. There are two categories of network monitoring elements related to the SEMIoTICS. The first involves the monitoring of network virtual functions where the network traffic is forwarded. In this case, Pattern Engine (PE) in the backend is capable to support the monitoring of the status of different network functions as retrieved by the MANO and the VIM as described extensively in D3.8. The second category includes the monitoring of the network elements such as hosts and switches as identified by the SEMIoTICS SDN Controller (SSC). More specifically, SSC can monitor the topology for addition or deletion of failures of nodes as well as links between the said nodes. The component that will mainly consume the events regarding the topology changes is the PE at the SDN layer of the SEMIoTICS architecture. Every change in the topology is inserted in PE at the SDN layer as a fact. Every fact that is inserted to the PE at the SDN layer is then propagated to the PE at the Application layer. In parallel the embedded monitoring mechanism of the PE at the SDN layer forwards the topology changes to the Monitoring component at the Application layer as depicted in Figure 22. In the following subsections, the proposed monitoring mechanism regarding the status of the network including the prototype implementation and its initial evaluation.

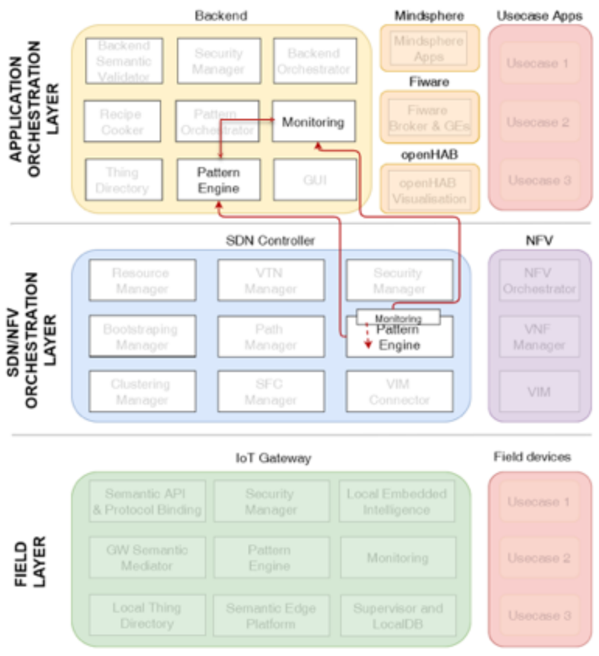


FIGURE 22 INTERACTION OF MONITORING COMPONENT WITH NETWORK MONITORING

As far as the SSC is concerned, the network topology is comprised of nodes and links between those nodes. The said nodes may be any network device such as a simple switch, a gateway or a host. Whenever a link failure occurs, the PE at the SDN layer must be informed of this, in the form of “fact” in order to reason whether any active SPDI or QoS property has been violated. For example, in UC1, where the QoS property of the link is monitored, the PE will alert that the said property does no longer hold upon failure of the said link.

In order for the PE to get the required facts, an interaction with the data of the SSC is necessary. Data in SSC is tree-based represented and SSC offers interfaces that can monitor/listen changes in the tree. In SSC, the use of *DataTreeChangeListener* interface is necessary to receive notifications about changes that occur in the data tree. The view of the change is cursor-based, which in some cases it can have lower overhead and provide a more flexible consumption of the change event.

In addition, the provided by the aforementioned interface methods are invoked every time when there is a data change event in the specific path of the tree. The said event is represented with the data before the change and data after the change therefore a simple comparison reveals the actual change in the data.

Using the above approach, the PE at the SSC registers two data-tree change listeners. The first listener is used to monitor the node changes as identified in the SSC network topology. The second listener is used to monitor the link changes as identified in the SSC network topology between the said nodes. This way, the PE becomes aware of the changes on the node and links, as soon as they occur in the SSC topology.

The created test network topology is used to evaluate the time that is required for the PE to become aware of topology changes. This is enabled by development of the necessary Drools rules that will print those changes to the SSC console. The topology consists of two hosts and two switches connected the way as shown in Figure 23. For simplicity purposes, the hosts are called piA and piB where the switches are the openflow:20 and openflow:12. In terms of pattern facts, the topology is consisted of 10 facts, including two switches, two hosts and 3 bidirectional links (6 in total).

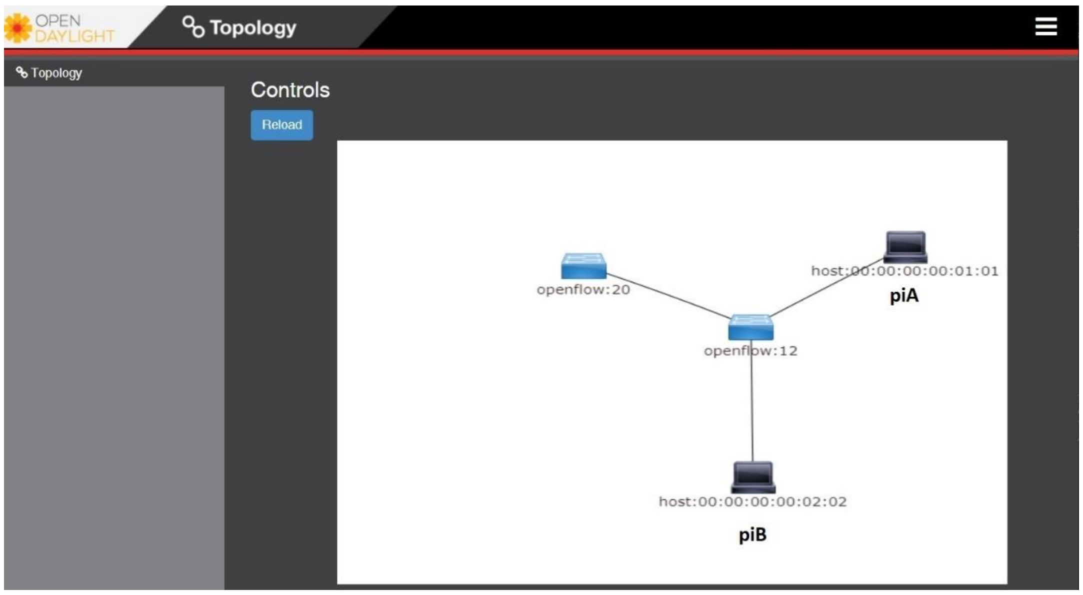
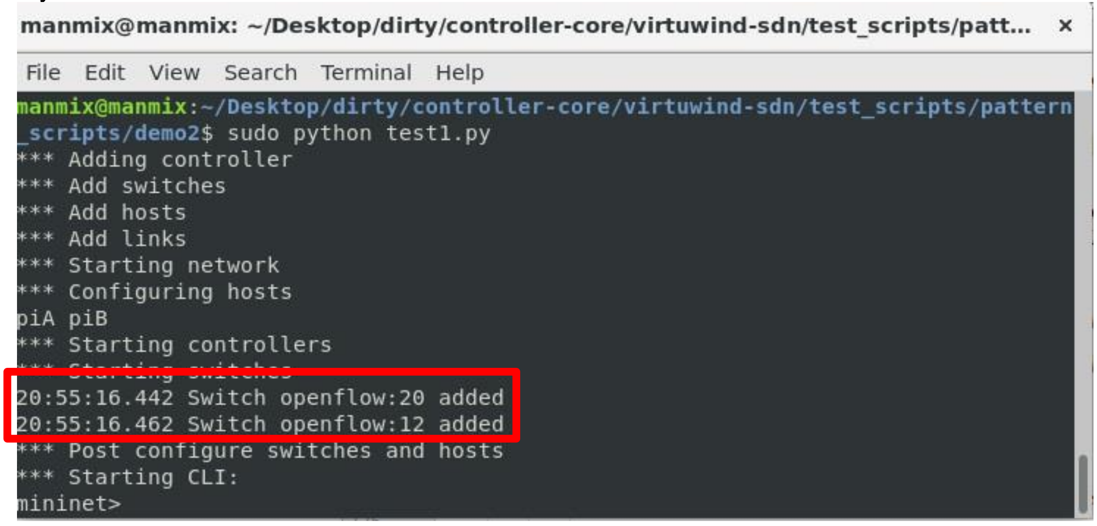


FIGURE 23 NETWORK TOPOLOGY

The topology is created using Mininet[58] (Figure 23) in order to provide us the means to accurately measure the time that is needed for the PE to collect the facts from the monitoring mechanism. The Mininet topology is intentionally run in the same machine with the SDN controller in order to have a common time reference.

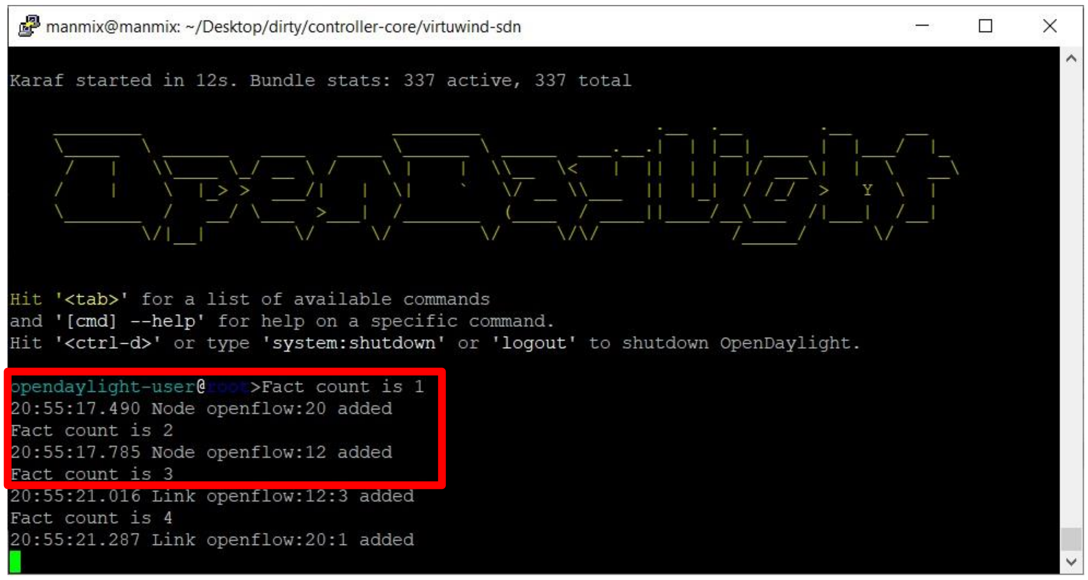

manmix@manmix: ~/Desktop/dirty/controller-core/virtuwind-sdn/test_scripts/patt...
File Edit View Search Terminal Help

```
manmix@manmix:~/Desktop/dirty/controller-core/virtuwind-sdn/test_scripts/pattern
_scripts/demo2$ sudo python test1.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
piA piB
*** Starting controllers
*** Starting switches
20:55:16.442 Switch openflow:20 added
20:55:16.462 Switch openflow:12 added
*** Post configure switches and hosts
*** Starting CLI:
mininet>
```

FIGURE 24 MININET SETUP

In the instantiation of the topology, the PE learns only about the two switches and the link between them (Figure 23). By observing closer the timestamps of Figure 23 and Figure 24, it can be observed that at 20:55:16.442 the openflow:20 is added and after 20ms the openflow:12 is also added. On the SDN side, it took approximately 1 sec for the fact to be added in the PE and fire the rule that prints it. The same happens for the openflow:12 after 298ms.

58 htttp://www.mininet.org



```
manmix@manmix: ~/Desktop/dirty/controller-core/virtuwind-sdn

Karaf started in 12s. Bundle stats: 337 active, 337 total

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>Fact count is 1
20:55:17.490 Node openflow:20 added
Fact count is 2
20:55:17.785 Node openflow:12 added
Fact count is 3
20:55:21.016 Link openflow:12:3 added
Fact count is 4
20:55:21.287 Link openflow:20:1 added
```

FIGURE 25 OUTPUT OF THE PE IN THE SSC CONSOLE FOR THE SWITCHES AND THEIR LINKS

However, the PE is not yet aware of the two hosts, since there is not any exchanged traffic between them, therefore the switches are unaware of their presence. This means that also the SSC is unaware of them and evidently the PE is unable to acquire any facts about them. In order to notify the switches for the existence of the hosts, a simple ping between the two hosts is enough to create interesting traffic. To measure accurately the time when the traffic is populating, a python script is used in the piA. The purpose of the script is to initiate a ping and after 5 seconds to disable the Ethernet interface with a purpose to simulate a link failure between piA and openflow:12, resulting in the topology shown in Figure 25.

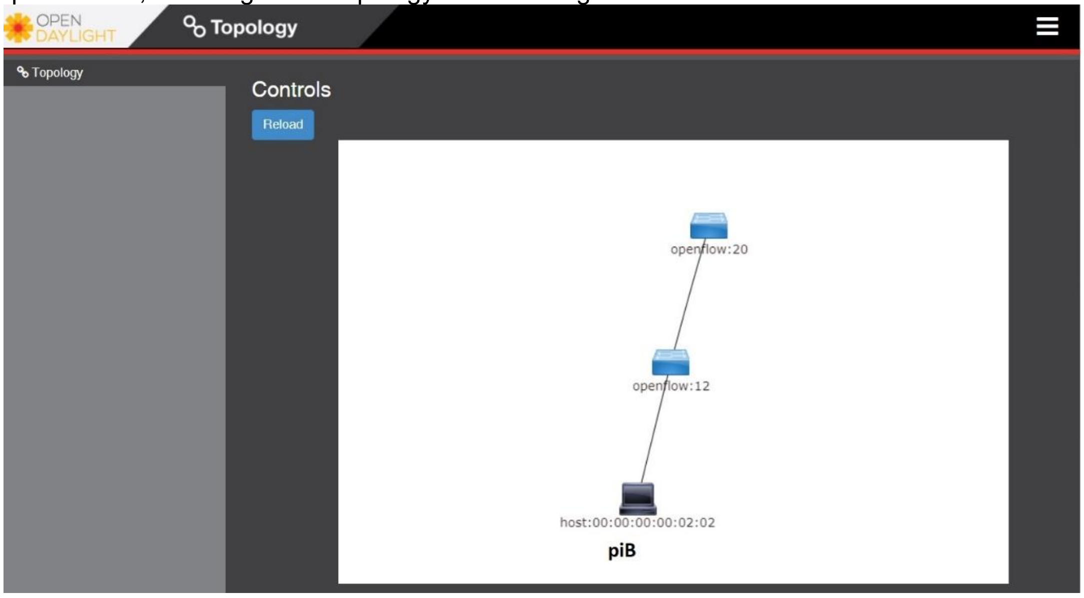
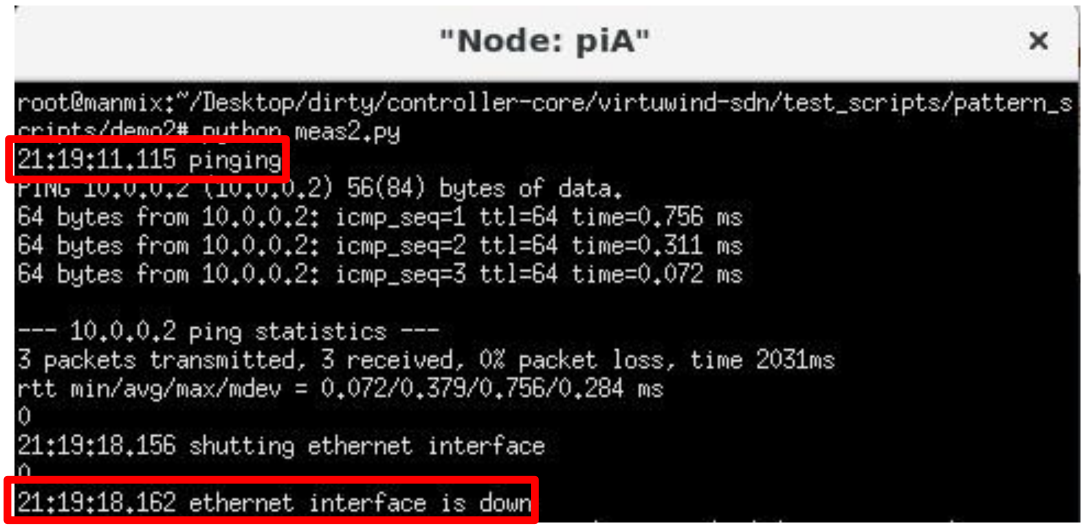


FIGURE 26 NEW TOPOLOGY AFTER THE LINK FAILURE

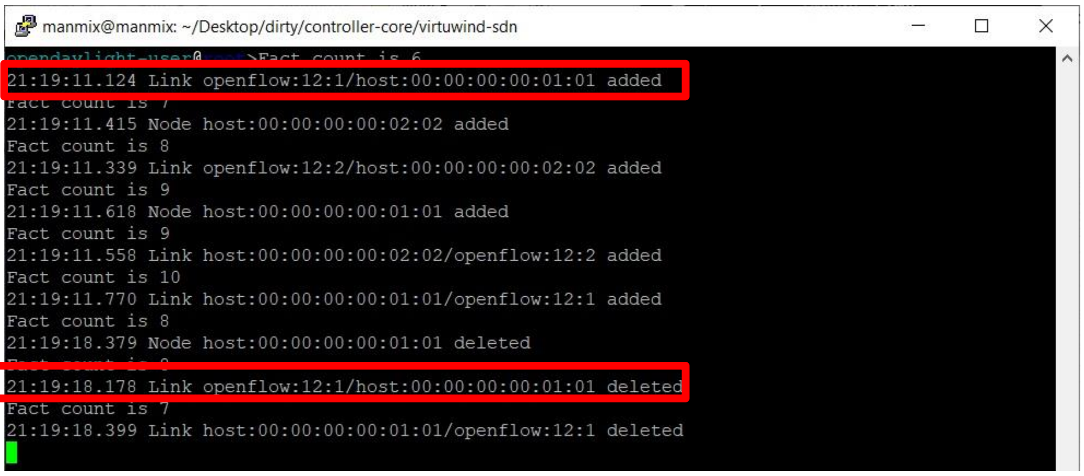
After comparing the output of piA from Figure 26, with the output of the PE from Figure 27, it can be observed that the traffic is initiated at 21:19:11.115 and the PE is made aware of the first changes just 9ms later at 21:19:11.124. All the required facts have been added to the PE by 21:19:11:770 where the fact count has reached number 10, which means that the total process took 655ms to complete. Regarding the link failure, it occurs at 21:19:18.162 at the piA side and the PE is made aware of the failure after 16ms.



```
"Node: piA"
root@manmix:~/Desktop/dirty/controller-core/virtuwind-sdn/test_scripts/pattern_s
cripts/demo2# python meas2.py
21:19:11.115 pinging
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.756 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.311 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.072 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2031ms
rtt min/avg/max/mdev = 0.072/0.379/0.756/0.284 ms
0
21:19:18.156 shutting ethernet interface
0
21:19:18.162 ethernet interface is down
```

FIGURE 27 POPULATION OF TRAFFIC AND LINK FAILURE SIMULATION



```
manmix@manmix: ~/Desktop/dirty/controller-core/virtuwind-sdn
opendaylight-user@root>Fact count is 6
21:19:11.124 Link openflow:12:1/host:00:00:00:00:01:01 added
Fact count is 7
21:19:11.415 Node host:00:00:00:00:02:02 added
Fact count is 8
21:19:11.339 Link openflow:12:2/host:00:00:00:00:02:02 added
Fact count is 9
21:19:11.618 Node host:00:00:00:00:01:01 added
Fact count is 9
21:19:11.558 Link host:00:00:00:00:02:02/openflow:12:2 added
Fact count is 10
21:19:11.770 Link host:00:00:00:00:01:01/openflow:12:1 added
Fact count is 8
21:19:18.379 Node host:00:00:00:00:01:01 deleted
Fact count is 8
21:19:18.178 Link openflow:12:1/host:00:00:00:00:01:01 deleted
Fact count is 7
21:19:18.399 Link host:00:00:00:00:01:01/openflow:12:1 deleted
```

FIGURE 28 OUTPUT OF THE PE IN THE SSC CONSOLE FOR THE HOSTS, THEIR LINKS AND THE LINK FAILURE

Taking the measurements from the tests presented in the previous section, it can be assumed that the monitoring mechanism implemented in the SSC for the links and nodes, is adequate for providing notifications to the PE in a reasonable timeframe (Figure 28). Specifically, the measurements reveal that the changes that occur in the network topology are made available to the PE within a maximum period of approximately 1 sec or even as low as 9ms. These timings are crucial to the adaptation techniques that are inherited by PE because the sooner a failure is made known, the better an adaptation action response will be. Finally, the proposed mechanism can be deployed in all network related aspects of the different use cases of SEMIoTICS to provide a runtime monitoring mechanism for enabling also the required runtime adaptations.

5.4. WoT Signaller

The WoT Signaler is implemented as a Java library that allows the MPD to interact with entities compliant with the WoT standard.

The WoT Signaler implements the *Signaler* interface offering the operations to read, write or subscribe properties of Web Things. Moreover, it allows the MPD controller to retrieve the possible bindings for source queries appearing in a Query (e.g. get all entities with type IMU from mobile device).

The WoT Signaler transforms the resource observation requests received via the signaler interface into proper WoT event subscriptions and, for each observed event, keeps the list of clients observing that event. If the signaler receives an observation request for an event already subscribed the new observer is simply added to the list of observers of that resource without creating a new subscription toward the Web Thing.

5.5. Backend Orchestrator Signaller

To monitor the events related to Backend Orchestrator a Backend Orchestrator Signaller had to be created. To achieve that, the component uses Backend Orchestrator and Monitoring API. The Backend Orchestrator exposes API to get the events that have occurred in the Kubernetes cluster. On the other hand, Monitoring API provides an endpoint to inject the low-level events to the Monitoring Component.

Backend Orchestrator Signaller sends the HTTP requests containing information about ongoing Backend Orchestrator's events to Monitoring Component via its API. The component is a stand-alone application running on the Backend Orchestrator. The component has been build according to the REST architecture and along with the documentation of Monitoring Component.

The main Backend Orchestrator Signaller functionality is to make HTTP requests to the given API with a constant frequency. After getting those events, the component should filter only events that are of interest. The final step is to send the aforementioned events to Monitoring Component as low-level events using Monitoring's API.

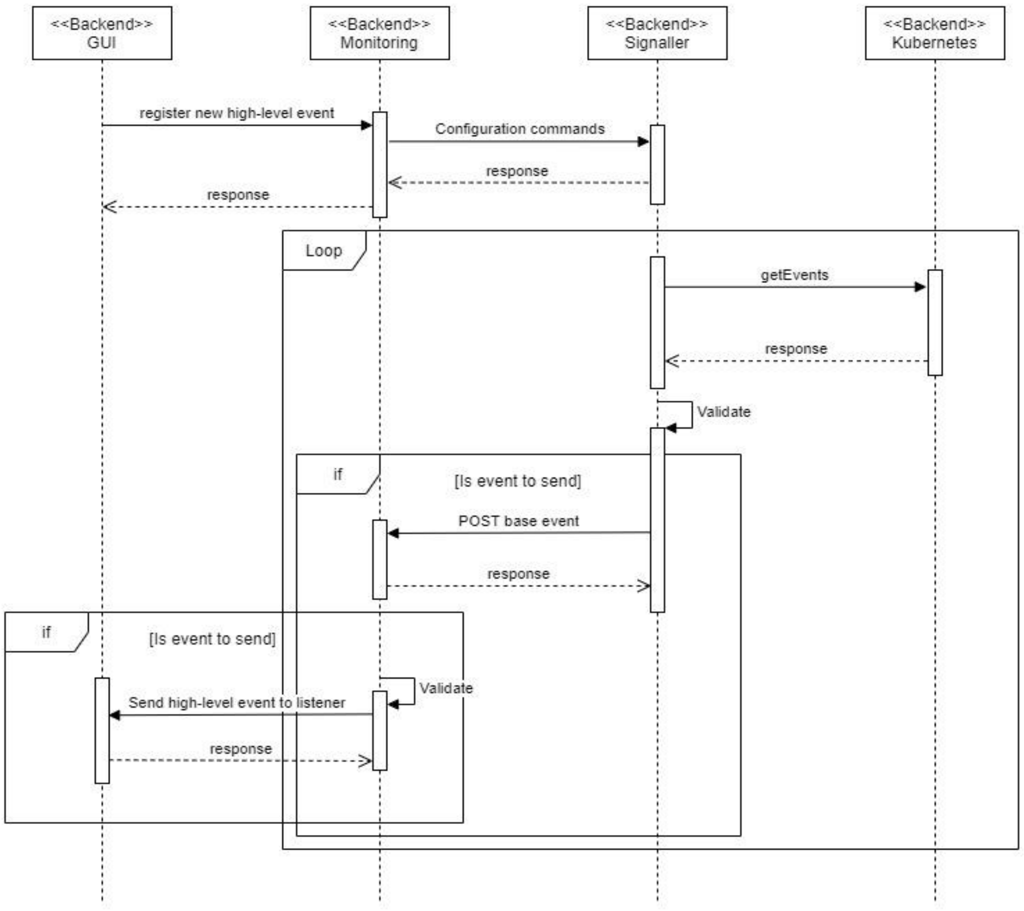


FIGURE 29 SEQUENCE DIAGRAM SHOWING THE FLOW BETWEEN COMPONENTS

The Kubernetes API allows getting many, various events. The events contain information about the state of Kubernetes's objects like Pods, Nodes. The filter functionality in Backend Orchestrator Signaller allows specifying many events that are to be filtered. Since the component is supposed to monitor the health and lifecycle of the pods, only events related to the pods are of interest. In the table below, the events related to pods' lifecycles are listed. The filter is generic, so it can be easily altered to monitor different events.

Event Name	**Severity**	**Description**
FailedToCreateContainer	Critical	Failed to create a container
FailedToStartContainer	Critical	Failed to start a container
PreemptContainer	Warning	Preempting other pods.
BackOffStartContainer	Warning	Back-off restarting failed the container.
ExceededGracePeriod	Warning	Container runtime did not stop the pod within the specified grace period.
FailedToKillPod	Warning	Failed to stop a pod.
FailedToCreatePodContainer	Moderate	Failed to create a pod container.
FailedToMakePodDataDirectories	Moderate	Failed to make pod data directories.
NetworkNotReady	Critical	The network is not ready.
FailedScheduling	Critical	Unable to schedule pod
FailedToPullImage	Critical	Failed to pull the image.
FailedToInspectImage	Warning	Failed to inspect the image.
ErrImageNeverPullPolicy	Warning	The image's NeverPull Policy is violated.
ImagePullBackOff	Critical	Container image pull failed, kubelet is backing off image pull
ImageInspectError	Warning	Unable to inspect image
ErrImagePull	Critical	Image pull error
ErrImageNeverPull	Critical	Required Image is absent on host and PullPolicy is NeverPullImage
RegistryUnavailable	Critical	Getting http error when pulling an image from the registry
InvalidImageName	Critical	Unable to parse the image name

The development process of Backend Orchestrator Signaller was as follows:

1. Analysis of Kubernetes events.
2. Implementation of HTTP request sender to get the cluster's low-level events using Kubernetes API.
3. Unit testing of the HTTP request sender.
4. Implementation of a job which would trigger HTTP request with a constant frequency.
5. Implementation of a filter method in order to filter the events of interest out of all the events.

6. Unit testing of the filter method.

7. Implementation of HTTP request sender, to send low-level events to Monitoring API.

8. Unit testing of HTTP request sender.

Detailed development will be described in the relevant Deliverable D4.13 Implementation of SEMIoTICS BackEnd API (Final Cycle) while integration and testing results will be described in deliverables D5.7 Software system integration (Cycle 2) and D5.8 IIoT Infrastructure set-up and testing (Cycle 2) respectively.

The component does not provide any API, though it uses Backend Orchestrator and Monitoring APIs. The endpoints which are used are:

- Backend Orchestrator: method *GET*, endpoint*:* `/api/v1/namespaces/semiotics/events`
- Monitoring: method GET, endpoint: `/semiotics/api/mdp/queries`

5.6. Complex Event Processor

The implementation of Complex Event Processor of the MPD component is based on Apache Flink Event Processor (see section 2.3.2) and its CEP library.

Apache Flink was chosen over PROTON (section 2.3.1) because:

- the PROTON project was no longer active at the date (June 2019) of the start of implementation activities for the MPD. The decision was to reject a technology becoming soon obsolete.
- Apache Flink is one of the technologies targeted by the FIWARE COSMOS Generic Enabler. The FIWARE COSMOS Generic Enabler is a set of tools that help achieving the tasks of Streaming and Batch processing over context data[59]. The decision was to select a technology aligned also with the technological choices made by FIWARE
- Apache Flink is also one of the technologies adopted within the ENG Digital Enabler[60] ecosystem and hence background experience about the Flink technology was more accessible to the development team of MPD component.

The MPD component is available in two flavor: Field and Backend. The Filed layer flavor is supposed to run on a computational node having reduced computational resources (e.g. a Raspberry Pi) whilst the Backed flavor is supposed to run on a cloud infrastructure having a plenty of computational resources. The interaction between the MPD and Flink varies depending on the flavor of the MPD:

- a Filed layer monitoring component runs Flink executes Flink within the same JVM
- a Backend layer monitoring component interacts with a Flink instance running on a different JVM

5.7. IoT Botnet Attack Detector

The IoT botnet attack detection algorithmic approach is thoroughly described in section 4.2. It constitutes a novel research idea under the SEMIoTICS framework (see publication [39]), and thus MATLAB scientific programming environment was used during the experimental evaluation.

5.8. Anomaly detector using LSTM

[59] https://fiware-cosmos-flink.readthedocs.io/en/latest/

[60] https://www.eng.it/en/our-platforms-solutions/digital-enabler

Implementing the LSTM architecture for intrusion detection, as defined in subsection 4.3, is accomplished by utilizing Keras[61] a high-level neural networks API, written in Python and capable of running on top of Google's TensorFlow[62].

For expediting the training process we a workstation running CUDA framework for GPU acceleration [7][35] can be used. Additionally, libraries to enable pre-processing, evaluation metrics and provide visualization means, such as Skikit-learn, Pandas, Numpy and matplotlib can be utilized. In terms of the hardware platform needed to deploy the framework, a typical system requires an NVIDIA graphic card with CUDA capabilities, such as NVIDIA'S 1070 GTX.

The model can be trained and tested via commonly used datasets for IDS benchmarking, such as NSL-KDD and CIDDS-001 [40]. Considering the model's configuration and inspired from the aforementioned studies, we can train and validate the model with various hyper parameter values (e.g., learning rates of 0.01, 0.1, 0.5), using different optimizers (e.g., Adam [21], rmsporop, SGD [31]) and loss functions (i.e., binary cross entropy, categorical cross entropy [8]).

Based on the produced validation results, especially considering the model's accuracy and false positive rate, the configuration that will produce the best trained classifier is selected and can be deployed in the anomaly detector.

5.9. Causal Model Identifier

The Causal Model Identifier (CMI) is a Java module responsible to identify causal relations existing between the variables monitored by the MPD.

The Causal Model Provider interface implemented by the CMI allows a client (e.g MPD Controller) application:

- to start the identification process of the causal relations existing between the state variables in a context
- to query the Causal Model

In the current implementation the CMI supports the discovery of causal relations existing in the context of a MPD query i.e. between the resources being referenced in a query being processed by the MPD Controller.

The CMI discovers the causal relations incrementally by processing the event streams produced by the Events Signalers. The CMI builds the Causal Model for a context incrementally by merging the information of two data structures it keeps updated with the events received by the event sources (e.g. Web Things, Fiware Context Broker) via the signalers: a Bayesian network and a network representing the temporal precedencies between the observed events.

5.10. Events Predictor

The Events Predictor predicts the next occurrence of events by processing the the event streams produced by both the Events Signalers and the Complex Events Processor.

In its present implementation the Events Predictor (EP), implemented in the Java programming language, runs as an independent thread of control within the JVM of MPD component.

The interface of the EP allows a client application (e.g. Complex Event Processor) to request the notification of the next occurrence of:

- events of a specific type whenever the likelihood of their predicted occurrence exceeds a given threshold
- the *k* events that are more likely to occur next.

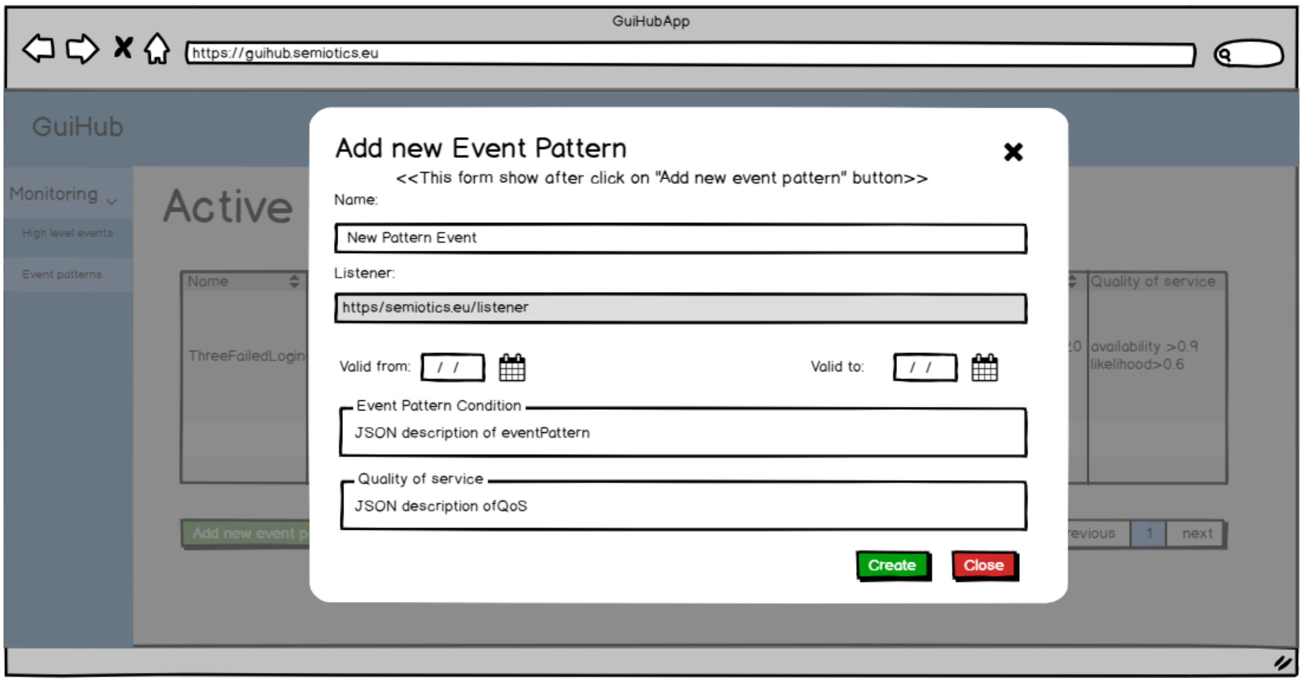
5.11. Graphical User Interface

SEMIoTICS GUI is a component that provides the visualization of the monitoring event and allows users to register the Event Pattern used by MPD to generate high-level events. It will be performing the role of Client

[61] https://keras.io

[62] https://www.tensorflow.org/

Component (see Figure 4) while utilizing the interface described in section 2.6. The Figure 30 below presents the form to be provided to the end-user to define the high-level event.



Add new Event Pattern

<<This form show after click on "Add new event pattern" button>>

Name: New Pattern Event

Listener: https/semiotics.eu/listener

Valid from: / /

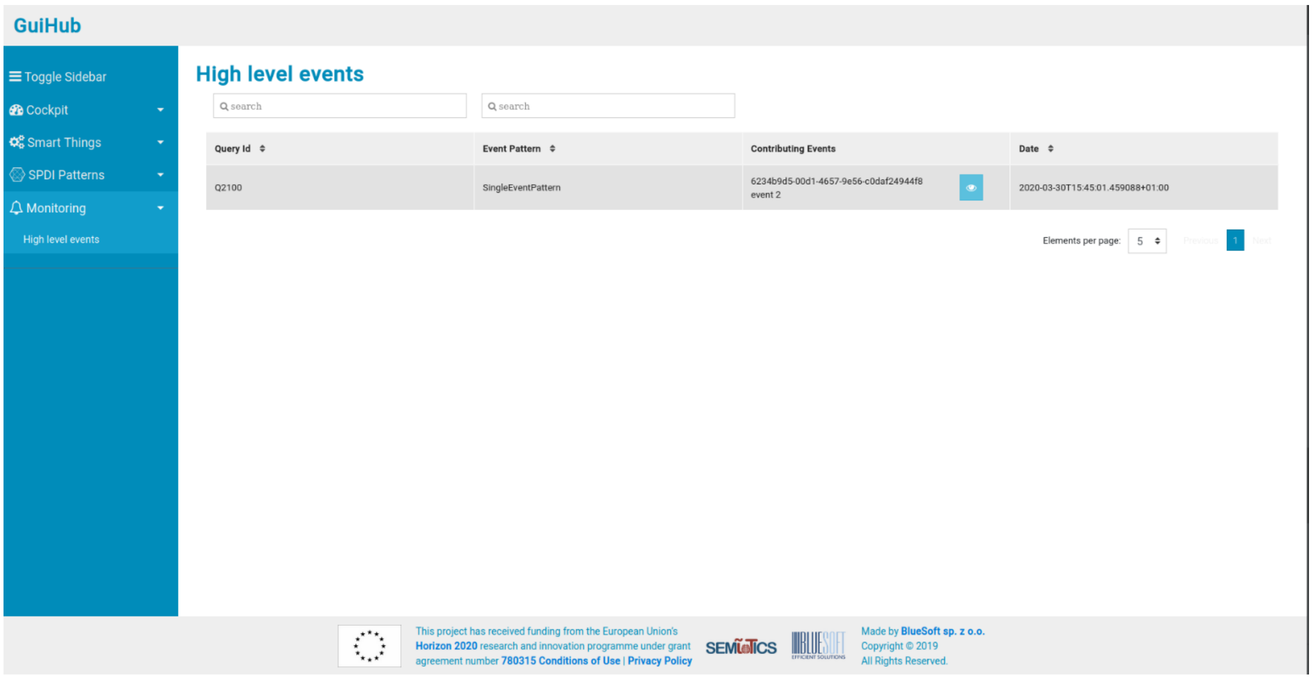
Valid to: / /

Event Pattern Condition: JSON description of eventPattern

Quality of service: JSON description ofQoS

FIGURE 30 ADD NEW EVENT PATTERN FORM

For visualization purposes, a new view in GUI showing the list of high-level events has been developed. It will display only the high-level events that have been generated by fulfilling event patterns with the listener as GUI. Figure 31 below presents this table.



High level events

Query Id	Event Pattern	Contributing Events	Date
Q2100	SingleEventPattern	6234b9d5-00d1-4657-9e56-c0daf24944f8 event 2	2020-03-30T15:45:01.459088+01:00

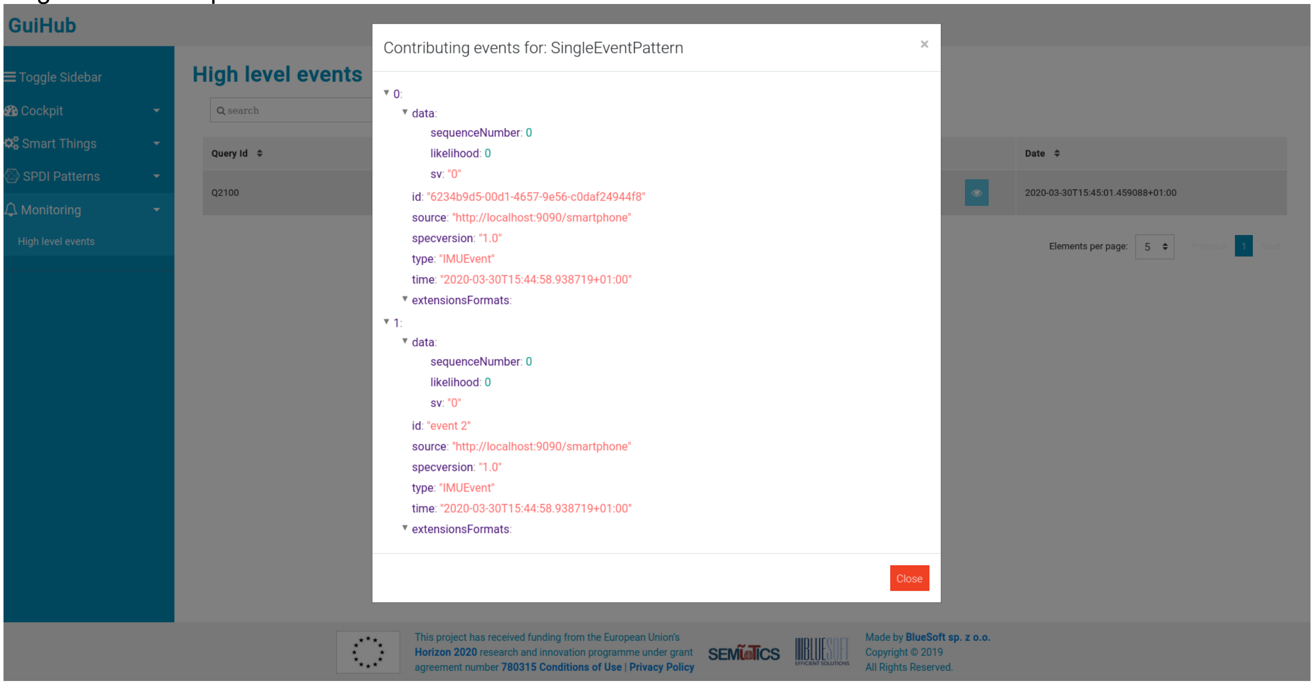
Elements per page: 5

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 31 HIGH-LEVEL EVENT VIEW

More information about contributing will be displayed after clicking eye icon in “Contributing Event column”. Figure 32 below present the view of contribution event.


GuiHub
Toggle Sidebar
Cockpit
Smart Things
SPDI Patterns
Monitoring
High level events
High level events
search
Query Id
Date
Q2100
2020-03-30T15:45:01.459088+01:00
Elements per page: 5
1
Close
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy
Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

Contributing events for: SingleEventPattern

```
▼ 0:
  ▼ data:
      sequenceNumber: 0
      likelihood: 0
      sv: "0"
    id: "6234b9d5-00d1-4657-9e56-c0daf24944f8"
    source: "http://localhost:9090/smartphone"
    specversion: "1.0"
    type: "IMUEvent"
    time: "2020-03-30T15:44:58.938719+01:00"
  ▼ extensionsFormats:
▼ 1:
  ▼ data:
      sequenceNumber: 0
      likelihood: 0
      sv: "0"
    id: "event 2"
    source: "http://localhost:9090/smartphone"
    specversion: "1.0"
    type: "IMUEvent"
    time: "2020-03-30T15:44:58.938719+01:00"
  ▼ extensionsFormats:
```

FIGURE 32 CONTRIBUTING EVENT VIEW

6. VALIDATION

This chapter summarizes the validation features of SEMIoTICS that are related with the monitoring, prediction and diagnosis and the various topics that are covered in this deliverable.

6.1. Related Project Objectives and Key Performance Indicators (KPIs)

The objectives of the related T4.2 (as per DoW) and their mapping to D4.2 content are summarized in the following table.

T4.2 Objectives	D4.9 Chapter
A monitor management layer, able to connect with different IoT platform and cloud monitors and smart object event captors.	2.1, 2.6.6, 2.6.7
The monitoring management layer will supports adaptation based on matching of dynamically evolving monitoring requirements with dynamically evolving available monitoring capabilities.	2.1.1, 2.6.3
Generic predictive and diagnostic mechanisms, utilising the information obtained from the monitoring mechanisms.	2.6.9, 2.6.10, 3, 4
Adaptation of existing methods for causal modelling, causal discovery, and causal inference in IoT large-scale applications for predictive modelling, anomaly detection, and diagnosis	3.3, 5.8, 5.9

The overall deliverable constitutes the contribution towards fulfilling the project’s requirements regarding:

- **SEMIOTIC’s Objective 3** (Development of dynamically and self-adaptable monitoring mechanisms supporting integrated and predictive monitoring of smart objects of all layers of the IoT implementation stack in a scalable manner.) and the relevant:
    - **KPI 3.1** (Delivery of a monitoring management layer for: (a) generating monitoring strategies for different checks and configurations of monitors available in the 3 targeted IoT platform, (b) fusing results of these 3 IoT platform monitors, and (c) performing predictive monitoring with an aimed accuracy of 80% on average)
    - **KPI 3.2** (Delivery of a generic monitoring language capable of defining platform agnostic monitoring conditions (as part of SPDI patterns), correlations of different IoT platform events that are necessary for this, and predictive monitoring checks.).

7. CONCLUSIONS

One of the specific objectives of the SEMIoTICS project is the development of dynamically and self-adaptable monitoring mechanisms supporting integrated and predictive monitoring of smart objects of all layers of the IoT implementation stack in a scalable manner.

This deliverable presented the design of the SEMIoTICS MPD Component along with a review of the algorithmic and technological options considered for the implementation of its key functionalities.

As described above, the concrete conditions monitored by the SEMIoTICS MPD are derived by the Pattern Orchestrator component starting from the recipes received by the Recipe Cooker.

The software architecture of the MPD includes a Controller sub-component for (i) checking the actual possibility to monitor such conditions across different IoT platforms and creating optimal monitoring strategies for this purpose, (ii) configuring automatically the monitors of IoT enabling platforms as required for different monitoring strategies.

Moreover, the software architecture of the MPD includes specific components (Event Signalers) having the role to translate the events generated by the different IoT/IIoT platforms (e.g. AWS, MindSphere) and software layers (e.g. network, field) into a common event format enabling the integration of monitoring results generated by the various IIoT platforms and layers of the IoT implementation stack. The IoT platforms considered in this deliverable are: Amazon Web Services, Microsoft Azure, Siemens MindSphere and FIWARE.

OpenStack Nova and BEATS tools can be used for the monitoring respectively of OpenStack instances and the Linux kernel. The common event format is based on the CloudEvents format.

Apache Flink and FIWARE Proton are the two options that were considered for the implementation of the Complex Event Processor responsible for the integration of events represented in the CloudEvents format. For the final implementation the Apache Flink options was chosen.

Continuous uninterrupted monitoring requires that the MPD has self-adapting capabilities. These capabilities require that the MPD is able to predict future states of the monitoring configurations and to identify (diagnose) the root causes of those state changes that inhibits the monitoring.

This deliverable identifies Causal Networks as an approach for reasoning about the discrete events accounting for the state changes of the monitoring infrastructure.

For what the continuous domain is concerned the Deep Neural Networks are identified as a suitable approach for reasoning at the cloud level. At the Field layer an approach based on regression techniques is indicated as more appropriate.

The deliverable also presents approaches to identify specific anomalies and attacks. Abuses of computational resources can be dealt by observing CPU, memory consumption and system calls. Anomalies can be detected using Long-Short-Term-Memory networks.

7.1. Implementation status and future work

The design, technologies and algorithmic approaches presented by this document represent the basis for the development of the final version of the Monitoring Component. The experience gained with the development and utilization of the first version of the Monitoring Component informed the final design of the SEMIoTICS Monitoring, prediction and diagnosis mechanisms.

The development process of Monitoring Component had the following milestones:

- **M7**: start of design activity and technology scouting.
- **M12**: start of development (coding) of the monitoring framework and components described in section 2.1
- **M24:** release of the first version of the API, integration with one CEP, adapters for FIWARE platform and WOT devices.

- **M28:** release of the second version including, in addition to the bug fixing and optimization of the functionalities of the previous version, the prediction and diagnosis mechanisms based on Causal Networks and the other approaches presented in chapters 3 and 4.

To the date of release of the present deliverable the second version of the MPD is being tested and its integration within the overall infrastructure is ongoing. The final results of this integration effort will be reported in the deliverable D4.13 - “Implementation of the SEMIoTICS BackEnd API”.

8. REFERENCES

1. Acharya, Saurav, "Causal modeling and prediction over event streams", Graduate College Dissertations and Theses, Paper 286, 2014.
2. Althubiti, S. A., Jones, E. M., & Roy, K. (2019). LSTM for Anomaly-Based Network IntrusionDetection. 2018 28th International Telecommunication Networks and Applications Conference, ITNAC 2018, 1–3. https://doi.org/10.1109/ATNAC.2018.8615300
3. D. Basak, S. Pal, and D. C. Patranabis, "Support vector regression," Neural Information Processing-Letters and Reviews, vol. 11, no. 10, pp.203–224, 2007.
4. Bengio (1994) 1994 Learning long-term dependencies with gradient descent is difficult. (n.d.). https://doi.org/10.1109/72.279181
5. Borchani, H., Chaouachi, M., and Ben Amor, N. (2007). Learning causal bayesian net- works from incomplete observational data and interventions. In Proceedings of the 9th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU '07, pages 17–29, Berlin, Heidelberg. Springer-Verlag
6. L. Breiman, "Random forests," in Machine Learning, vol.45, no.1, pp.5—32, 2001.
7. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., & Hanrahan, P. (2004). Brook for GPUs. ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04, 777. https://doi.org/10.1145/1186562.1015800
8. Loss Functions in Artificial Neural Networks | Isaac Changhau." [Online]. Available: https://isaacchanghau.github.io/2017/06/07/Loss-Functions-in-Artificial-Neural-Networks/
9. Understanding LSTM Networks -- colah's blog." [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.
10. G. Cooper, "Computational Complexity of probabilistic inference using Bayesian belief networks (research note)," Mach. Learn., vol. 42, no. 2–3,
11. SEMIoTICS Project, "Software defined programmability for IoT devices (first draft)", Deliverable D3.1, 2018
12. SEMIoTICS Project, "Network Functions Virtualization for IoT (first draft)", Deliverable D3.2, 2018
13. Ellis, B. and Wong, W. H. (2008). Learning causal Bayesian network structures from experimental data. Journal of the American Statistical Association, 103(482):778–789.
14. Ergen, T., Mirza, A. H., & Kozat, S. S. (2017). Unsupervised and Semi-supervised Anomaly Detection with LSTM Neural Networks. 1–12. Retrieved from http://arxiv.org/abs/1710.09207
15. I. Goodfellow et al., "Generative adversarial nets," in Proc. Adv. Neural Inf. Process. Syst., 2014, pp. 2672–2680.
16. I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning," MIT Press, 2016, http://www.deeplearningbook.org.
17. O. Goudet, D. Kalainathan, P. Caillou, I. Guyon, D. Lopez-Paz, and M. Sebag, "Causal Generative Neural Networks," 2017.
18. Heckerman, D. (1995). A Bayesian approach to learning causal networks. In Proceedings of the Eleventh conference on Uncertainty in artificial intelligence, UAI'95, pages 285–295, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
19. Hockreiter (1991) Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, TU Munich, 1991.
20. Kim, J. (n.d.). 1994 Learning long-term dependencies with gradient descent is difficult. https://doi.org/10.1109/72.279181

21. D. P. Kingma and J. L. Ba, “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION.”

22. Li, G. and Leong, T.-Y. (2009). Active learning for causal bayesian network structure with non-symmetrical entropy. In Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD ’09, pages 290–301, Berlin, Heidelberg. Springer-Verlag.

23. D. Li, D. Chen, L. Shi, B. Jin, J. Goh, and S.-K. Ng, “MAD-GAN: Multivariate Anomaly Detection for Time Series Data with Generative Adversarial Networks,” in arXiv:1901.04997, 2019.

24. Meganck, S., Leray, P., and Manderick, B. (2006). Learning causal bayesian networks from observations and experiments: a decision theoretic approach. In Proceedings of the Third international conference on Modeling Decisions for Artificial Intelligence, MDAI’06, pages 58–69, Berlin, Heidelberg. Springer-Verlag.

25. Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, D. Breitenbacher, A. Shabtai, and Y. Elovici, “N-BaIoT: Network-based detection of IoT botnet attacks using deep autoencoders,” in IEEE Pervasive Computing, vol. 17, no. 3, pp. 12–22, Jul-Sep 2018

26. R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, “How to Construct Deep Recurrent Neural Networks,” in International Conference on Learning Representations, April, 2014.

27. J. D. Ramsey and D. Malinsky, “Comparing the Performance of Graphical Structure Learning Algorithms with TETRAD,” 2016.

28. C. E. Rasmussen and C. K. Williams, “Gaussian processes for machine learning,” MIT press Cambridge, vol. 1., 2006.

29. Rebane G, Pearl J, ""The Recovery of Causal Poly-trees from Statistical Data"". Proceedings, 3rd Workshop on Uncertainty in AI. Seattle, WA. pp. 222–228, 1987"

30. D. a. M. S. Revathi, “A Detailed Analysis on NSL-KDD Dataset Using Various Machine Learning Techniques for Intrusion Detection,” Int. J. Eng. Res. Technol., vol. 2, no. 12, pp. 1848–1853, 2013

31. S. Ruder, “An overview of gradient descent optimization algorithms *.”

32. M. Sabhnani and G. Serpen. “Application of machine learning algorithms to KDD intrusion detection dataset within misuse detection context". InInternational conference on machine learning, models, technologies and applications (MLMTA), pp. 209{215.CSREA Press, 2003.

33. P. Spirtes, C. Glymour, and R. Scheines, Causation, Prediction, and Search. 2001. 2nd Edition.

34. R. C. Staudemeyer, “Applying long short-term memory recurrent neural networks to intrusion detection,” 2015.

35. J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, “Accelerating molecular modeling applications with graphics processors,” J. Comput. Chem., vol. 28, no. 16, pp. 2618–2640, Dec. 2007.

36. S. Triantafillou and I. Tsamardinos, Constraint-based Causal Discovery from Multiple Interventions over Overlapping Variable Sets, 2015, v16

37. J. A. Tropp and A. C. Gilbert, “Signal recovery from random measurements via orthogonal matching pursuit,” IEEE Trans. on Information Theory, vol. 53(12), pp. 4655–4666, December 2007

38. B. D. Tyler, “Event-Triggered Causality: A Causality Detection Tool for Big Data,” 2018.

39. C. Tzagkarakis, N. Petroulakis, and S. Ioannidis, “Botnet Attack Detection at the IoT Edge Based on Sparse Representation,” 2019 Global Internet of Things Summit (GIoTS), Aarhus, Denmark, 2019

40. Verma, Abhishek, and Virender Ranga. "Statistical analysis of CIDDS-001 dataset for Network Intrusion Detection Systems using Distance-based Machine Learning." Procedia Computer Science 125 (2018): 709-716."

41. V. Vovk, “Kernel ridge regression,” in Empirical Inference, Springer, pp. 105--116, 2013.

42. P. J. Werbos, “Backpropagation through time: what it does and how to do it,” Proc. IEEE, vol. 78, no. 10, pp. 1550–1560, 1990.

43. C. Yin, Y. Zhu, S. Liu, J. Fei and H. Zhang, "An enhancing framework for botnet detection using generative adversarial networks," 2018 International Conference on Artificial Intelligence and Big Data (ICAIBD), Chengdu, 2018, pp. 228-234.

44. A. Creswell, T. White and V. Dumoulin, K. Arulkumaran, B. Sengupta and A. A. Bharath, “Generative adversarial networks: An overview,” IEEE Signal Process. Mag., Apr. 2017.