Security, Privacy and IPR/Data Protection Requirements translated into Game-Theoretic Agent Behaviour

Project Acronym	IoT4Industry
Document-Id	D.5
File name	
Version	FINAL document
Date	Start: 01 April 2015 End: 31 September 2015
Author(s)	Robert Mulrenin Violeta Damjanovic-Behrendt
QA Process	Georg Güntner

Table of Content

IoT4Industry in a Nutshell
IoT4Industry Task 5 Description
    Abbreviations
1. Problem Statement
2. IoT4Industry Agent-based Multilayer Architecture
    2.1 External Agent Layer
    2.2 Internal Agent Layer
    2.3 Web of Things (WoT) Layer
    2.4 Internet of Things (IoT) Layer
3. IoT4Industry Prototype as a Smart Factory Proof-of-Concept
    3.1 “Internet of Things” Layer Components
    3.2 “Web of Things” Layer Components
4. Agent- and Knowledge Technology-related Requirements in IoT4Industry
5. IoT4Industry Agent Architecture Overview
    5.1. Monitoring Layer
    5.2. Communication Layer
    5.3. Business Layer
    5.4. Core System Edge Layer and Near Sensor Edge Layer
6. Semantic Models and Terminologies in IoT4Industry
    6.1. Virtual Sensor Description (VSD) Model and Services
    6.2. Observations Model and Services
        6.2.1. Semantic annotation of monitoring component messages (Observations)
        6.2.2. SenML JSON example
        6.2.3. SenML XSD Schema
    6.3. Contributing Internal Models (Location, Monitoring Network Systems, Threat)
        6.3.1. Model describing physical location and monitoring network systems
        6.3.2. Model describing systems of collaborating sensors and other monitoring layer components
        6.3.3. Model describing sensor settings corresponding to threat levels
    6.4. Semantic Modeling in IoT
        6.4.1. Models supporting observations messaging
        6.4.2. Models supporting sensor control
    6.5. Contributing Terminologies
7. Services in IoT4Industry
8. Conclusion and Future Work
References
Appendix A: Overview of Agent Technologies for IoT
        JIAC (Java-based Intelligent Agent Componentware)

micro JIAC
thingjs (thingjs-agent)
AKKA Agent
LogStash agents
Node-RED as Edge Agent Framework
Apache Camel
Appendix B: Overview of Rule Engines
Nools (Javascript, Node.js)
JBOSS Drools (Java)
Appendix C: Overview of Data Computation Frameworks
Apache Storm
Apache Spark
Apache Spark Streaming
Apache Flink
Apache Kafka and Apache Spark Integration Aspects
Appendix D: Overview of Message Bus Technology
MQTT
Apache Kafka
Apache Flume
zeroMQ / 0MQ
Schedulers
Appendix E: Overview of Semantic Technologies for IoT
Sensor Ontologies
SensorML (Sensor Model Language)
Appendix F: Overview of Security Technologies for IoT
ENISA Threat Terminology
Shostack and Microsoft Threat Modeling Tool
SECURE (Semantics Empowered Rescue Environment)

IoT4Industry in a Nutshell

IoT4Industry is an exploratory project funded by the Austrian Research Promotion Agency, for the period October 2014 – September 2015. It stands for "Secure, Privacy-preserving Agents for the Industrial Internet". The core research areas of IoT4Industry relate to security, privacy and IPR/data protection requirements, translated into game-theoretic agent behavior, which is simulated in a demo factory floor environment, supporting a supply chain negotiation. A demo factory floor is implemented within our IoT-lab, which is located at Salzburg Research.

While it is still not clear which protocols and technologies will become mainstream for the Industrial Internet, it has become clear that security and privacy will feature strongly in any risk assessment concerning the adoption of practices using the Industrial Internet. Thus, in IoT4Industry, we focus on the use of policy-enacting, multi-agent systems that securely manage machines and manufacturing cells, and we build a feasibility demonstrator based on open source tools and firmware. In addition, IoT4Industry helps us to prepare for internationally recognized contributions to science and technology in the field of Industrial Internet, notably in Horizon 2020.

IoT4Industry Task 5 Description

(from the *IoT4Industry* project proposal)

D5: Security, Privacy and IPR/ Data Protection Requirements translated into Game-Theoretic Agent Bahaviour

Goals:

1. Technical Report specifying how industrial security requirements can be translated into agent behaviour which is bounded by specific game-theoretic models/mechanisms
2. Journal paper on specifying industrial security requirements as agent behaviour using concepts from game theory

Description of the content

At this stage, we will have the experimental setup, the full platform integration and we will have an understanding what security requirements actors in the Industrial Internet are likely to have. We will also have sufficient understanding of game-theoretic mechanisms as applied to supply chain management. This puts us in a position where we can translate the security requirements into formally specifiable agent behaviour that can be enacted on the web-based IoT / agent platform.

Method

Summary of concepts and methodology. Formal specification of agent behaviour.

Milestones, results and deliverables

06/2015: Technical Report published

09/2015: Journal paper on specifying industrial security requirements as agent behaviour using concepts from game theory

Abbreviations

BDI	Belief Desire Intention
EIP	Enterprise Design Patterns (message systems, message transformation, routing, message endpoints, etc.)
ESB	Enterprise Service Bus
DSS	Decision Support System
SSN	Semantic Sensor Network Ontology, a means to model sensor descriptions, deployment info and sensor observations.

1. Problem Statement

The scientific goals of the *IoT4Industry* project can be formulated as follows:

- Can we conceptualise software agents that follow local privacy and security rules such that they can negotiate alternative production schedules in response to dynamically changing demand and supply situations?
- Can we formulate agents behaviour in line with the current state of the art in game theory?
- Can we create a feasibility demonstrator for a multi-agent, cross-enterprise scenario where the agents autonomously manage machines and manufacturing cells with which they communicate via a mesh network of lower-level and higher-level sensors, attached to the manufacturing machines?

In this report, we explores security, privacy and data protection requirements for *IoT4Industry* agents, which are described by using behavioral models based on signaling games, and presented in D.3 “Specifying Game-Theoretic Behaviour for Agents in Industrial Supply Chain”.

Document Organization. After introducing the problem statement, Section 2 describes high-level architecture of the *IoT4Industry* agent-based multilayered system, in which we focus on four layers: external agent layer, internal agent layer, Web of Things layer, and IoT layer. In Section 3, we discuss the implementation of the IoT and Web of Things (WoT) layers, with a focus on open source solutions. In Section 4, we continue the *IoT4Industry* implementation by adding external and internal agent layers on top of both IoT and WoT layers. From Section 5 on, we focus on designing our multi-agent architecture, which includes: (i) Monitoring layer, (ii) Communication layer, (iii) Business layer, (iv) Core System Edge layer, and (v) Near Sensor Edge layer. Section 6 discusses semantic models and terminologies with the potential to link to the existing knowledge bases, and to exploit the advantages of LDP (Linked Data Platform). In that sense, we discuss Virtual Sensor Description (VSD)-based models and services, internal models describing physical location, monitoring network systems, sensor settings corresponding to threat levels, sensor control, etc. Section 7 summarizes on services to be developed in *IoT4Industry*, and finally, in Section 8, some conclusion remarks and notes on future work are emphasized.

2. IoT4Industry Agent-based Multilayer Architecture

Our overall architectural vision of an agent-based multi-enterprise manufacturing environment in the *IoT4Industry* project, is motivated by the Industry 4.0 perspectives on *Smart Factories*, exploiting both Cyber Physical Systems (CPSs) and the IoT. The architectural breakdown of *IoT4Industry* system is done into four layers, as depicted in Figure 1: external agent layer, internal agent layer, Web of Things (WoT) layer, and IoT layer. It is presumed that all communication between components, especially between agents is secured (e.g. by encryption), and that all entities are able to authenticate themselves (e.g. by exchange of certificates).

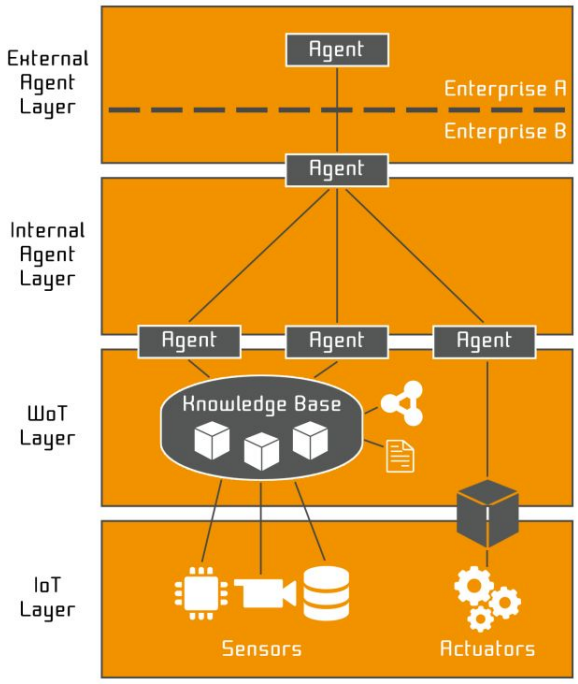

Figure 1: IoT4Industry Architectural Layers

In the following, we discuss each of four layers of the *IoT4Industry* architecture.

2.1 External Agent Layer

An external agent layer handles the enterprise-to-enterprise communication during the execution of the production processes. External agents that communicate on this layer with the outside world, follow interests of their own organization, which may result in conflicting interests with their communication partner from the other organizations. While all agents have access to internal knowledge base (either directly or via cooperating internal agents), the external agents

only get information related to their business interests, e.g. "*What is the expected delivery time and price for a specific product?*" Agents inside the company therefore need internal knowledge, such as the utilization of the production machines, costs of the raw material, additional costs, schedules of maintenance plans, etc. in order to state their offers to external agents. In order to maximize the profit and protect company business interests, external agents must hide internal knowledge and may not even tell the full truth. As an example, factories may have privileged customers, which always get priority and therefore buffers in the delivery times will be kept. Hence, in *IoT4Industry*, the external agent layer exploits game theoretic (GT) models specified in D.3, "Specifying Game-Theoretic Behaviour for Agents in Industrial Supply Chain".

2.2 Internal Agent Layer

The major difference to the previously discussed external agent layer is that in this layer all agents are "trusted". In other words, besides agent authentication, all the provided information is expected to be true, in order to support the agents' best knowledge. Although there may be errors in the data itself, the agents do not deliberately retain or distort information. The breakdown into multiple agents within a single company is done only to reduce the complexity of single agents. For examples, each of the internal agents can be responsible for separate entities such as one production cell inside a company, or for specific products. Therefore this layer is optional and may be skipped in small scenarios.

2.3 Web of Things (WoT) Layer

This layer delivers high-level information for the agents, by accumulating the available knowledge in a knowledge base. This includes data from sensors or records, as well as corresponding metadata, but also other contextually required information, which can be retrieved by ERP tools, Linked Open Data (LOD) repositories (e.g. semantic information on abstract concepts) or external services (e.g. data of a specific region). For agents, the WoT layer provides query services so that they can retrieve the status of the underlying production cell, without accessing the full range of raw sensor data. For agents to control the actuators, the WoT layer provides a control interface. Using this interface, agents can either queue orders, or deliver new design models to the underlying production cell.

In case of security boundaries inside the company, this layer needs to be vertically split into separate internal entities.

2.4 Internet of Things (IoT) Layer

Networked sensors and actuators constitute the IoT layer. Sensors on the one hand feed the knowledge base with their observations according to their configuration, continuously or triggered by observed events. In addition they can be requested to deliver information on the current status. The actuators on the other hand are able to perform specific activities along the manufacturing process as requested through their control interface.

The implementation perspectives of both layers WoT and IoT are described in more detail in the following section.

3. IoT4Industry Prototype as a Smart Factory Proof-of-Concept

D.6 "*Demo Implementation of Industrial IoT Factory Floor based on Arduino and Raspberry Pi*" describes the *IoT4Industry* scenario, which is rather an extract from a real production environment of a Smart Factory. Nevertheless, it gives us still information about the data to be collected, processed and transferred into higher-level knowledge, which can be further used by the agents for their interaction. In order to proof the feasibility of our approach, we firstly implemented the IoT and WoT layers based on existing technologies and components, giving priority to open source solutions. In Section 4, we continue the *IoT4Industry* implementation by adding external and internal agent layers.

3.1 "Internet of Things" Layer Components

The main task of the IoT layer is to interact with the physical world and provide connectivity to the network for sensors and actuators. This means that the IoT layer includes services, where sensors with low energy consumption and low processing power can connect to. In addition, the IoT layer needs to communicate sensor results and retrieve actuator commands from/to upper level services such as the "Web of Things", and other data consumers.

To proof the concept of our upstream data aggregation from the different types of sensors, we firstly integrated the existing IoT technologies and connected them as shown in Figure 2. In that way, a great diversity of technologies covering this functionality is already available, and some of them are published as open source. In our prototype implementation, we integrated a set of open source hardware and software projects to demonstrate the interoperability between a

variety of sensor types and the higher layer services. In the next paragraphs we shortly describe the sensor technologies available in our demo setup.

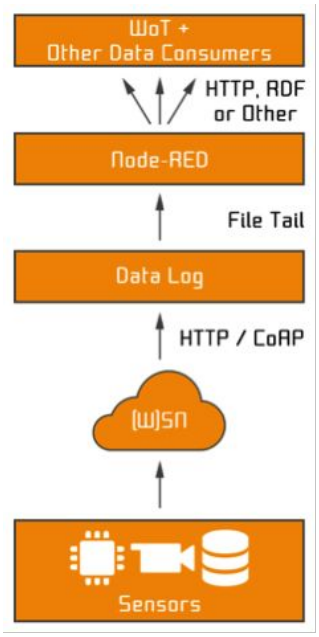

Figure 2: Technologies used in the IoT layer

Let's start with the **temperature and light intensity sensors** connected to a Texas Instruments evaluation module, which is equipped with an IEEE 802.15.4 compliant CC2538 system-on-chip. This platform can run the open source Contiki or RIOT-OS operating systems, which use 6LoWPAN for communication via a wireless sensor network. They also support the Constrained Application Protocol (CoAP), as well as HTTP to directly talk to the IoT service layer. Furthermore, the same type of sensors have been connected via the open source Arduino hardware platform, using an embedded 802.11 b/g wireless LAN module that provides onboard-support of HTTP for connectivity.

Another sensor platform that is actively used in our setup is based on open source Raspberry Pi computers. They are delivered with on board networking capabilities and can be WiFi-equipped by USB-dongles. On this platform we connect USB-based sensor hardware, such as **air quality sensors from Velux and a camera**.

Raspberry Pi is running a Debian-derived Linux system (Raspbian), which provides all required communication protocols implemented out of the box. To show interoperability with closed-source sensor platforms, we integrated a commercial sensor platform from a company called LineMetrics, for the **measurement of power consumption**. Their solution communicates

via public 3G networks with their cloud-hosted sensor management and data visualization web-services. Although proprietary solution, LineMetrics tool still provide access to the sensor data via simple REST-APIs.

The common requirement for all sensors used in *IoT4Industry* is that they must have a kind of IP network connectivity (IPv4 or IPv6) to communicate via HTTP or CoAP with an IoT service. This service can be implemented using open source software in various ways, e.g. a CoAP proxy implementation like Californium, or a commonly used HTTP server like Apache2. A large number of open source implementations for such services exist too. To simplify the prototype, we have not created a higher level IoT sensor service on top of CoAP or HTTP, but we are using information provided by the sensors directly from CoAP or HTTP request headers. Also the communication back to the remote sensors is simplified. In case the server accepts the reported values, it returns a “200 OK” (or “2.05 Content” in case of CoAP), otherwise an indicative error code is returned.

Furthermore, the service collects information from the sensor, including its identity based on the source IP address and an identifier provided by the sensor. The **local identifier i**s used to construct a globally unique URI required in the higher layers. In a real scenario, there may be distrust on the authenticity of the provided information. In that case additional authentication and security mechanisms can be implemented already on this level by adding (Datagram) Transport Level Security (D)TLS.

In order to record the sensor values, we use the logging features of the standard Apache2 HTTP server to store sensor raw data with a timestamp and the provided sensor identity into a local log file. To further pre-process the gathered information on the IoT Layer, e.g. sensing for predefined events, we connect a Node-RED server to the data logs. Node-RED is a tool for wiring together hardware devices, APIs and online services, and is able to filter information using e.g. regular expressions and further process the information in arbitrary forms – for example, sending alarms via MQTT, remotely control devices or upload extracted data in RDF triples according to the LDP principles to the WoT Layer (for more information on Node-RED, see Appendix. A.1.). In order to have a scalable solution for hundreds to thousands of sensors, the IoT layer can run multiple parallel entities; each of them provide the aggregated data to the upper layer. It has to be noted that in our current prototype implementation, the return channel towards the sensor, e.g. for reconfiguration or active value-request is not yet integrated, but will be added based on the future requirements.

3.2 “Web of Things” Layer Components

The main task of the WoT layer is to unlock the underlying IoT “data silos” and make their information available using open standards. This layer integrates not only separate IoT implementations as described in the previous section, but also additional information via internal services available through an organization and external Web services. This requirement clearly indicates that the use of technologies developed for Linked Data on the WWW could be a viable solution to apply. Therefore, our WoT layer is implemented on top of the open source Linked Data Platform (LDP) Server, which is available in Apache Marmotta (see: http://marmotta.apache.org/). The LDP application delivers knowledge collected from the sensors as Linked Data to the upper layer in *IoT4Industry* architecture. Apache Marmotta LDP backed by the SSN ontology, stores all information in its "kiwi" triple store, on a Postgres database. As depicted in Figure 3, triplified information includes:

- Sensor metadata, e.g. location, reporting/measurement frequency, unit of measurement;
- Sensor configuration data;
- Sensor raw data;
- Node-RED flows;
- Linked Data cache (local cache for links into the Linked Open Data repositories).

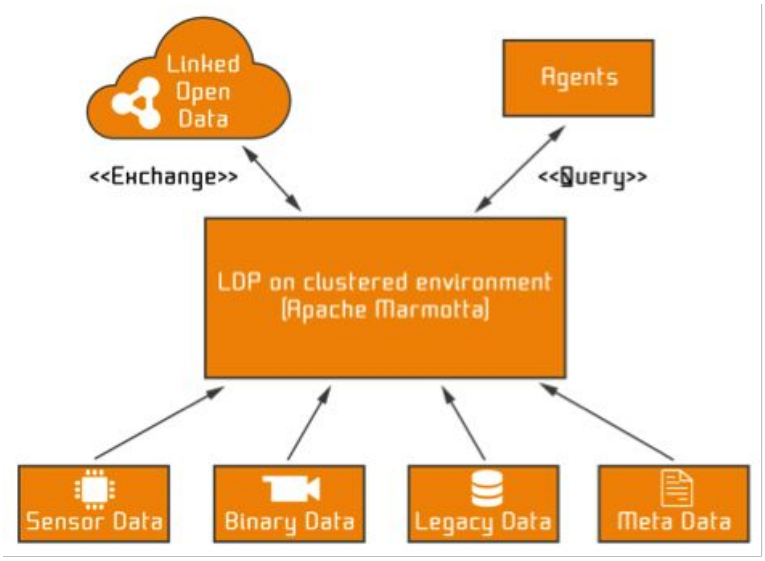

Figure 3: LDP data types and linking

There are several advantages of using LDP. Data can be accessed more simply compared to a “plain” triple store, plus there are documented constraints to provide an initial structure (e.g. RDF vs. Non-RDF resource). Furthermore, the flexibility and advantages of Semantic Web technologies are contained, and legacy data, relational documents and unstructured documents can be coupled easily. Once data or sets of data are submitted to the LDP server, they will become LDP Resources (LDPRs), which are specialized HTTP resources, accessible via an URI. In addition, they can be accessed, modified, created or deleted using HTTP requests. Finally, the concept of LDP containers makes it easier to locate existing resources, e.g. to find resources which belong to the same container. In the context of WoT, containers will help to easily access sensor information with the same type in different locations, or with different types at the same location.

4. Agent- and Knowledge Technology-related Requirements in IoT4Industry

This section overviews technologies with the potential to implement the *IoT4Industry* agent system. For more details, see Appendix A. One way to summarize the *IoT4Industry* requirements is by using horizontal and vertical layers. Horizontal layers refer on technology platforms that can scale to meet a range of system requirements, while vertical layers cover the functionality requirements, e.g. the ability to capture sensor data, ensure efficient decision making, etc. In that way, we consider the following technologies in *IoT4Industry*:

Horizontal layers:	**Vertical layers:**
● Decision Making Layer (DSS) ○ Multi-Agent System (MAS) ○ Game Theory models based on signaling games ● Control and IoT Edge Agent Layer ○ Agents, connectors, drivers ● Monitoring Layer ○ Sensors ○ Sensor networks	● Messaging ○ Near Sensor Message Brokers ○ Distributed System Message Brokers ● Agent Configuration ● Sensor and Network Configuration ○ Sensor registry ● Security

What functionalities does the IoT4Industry infrastructure have?

The following provides a summary of technological suggestions for the agent implementation.

Multi Agent Systems (MAS) in *IoT4Industry* need to support DSS components, rule engine and scheduler, workflows, services.

DSS should utilize a Belief-Desire-Intention (BDI) paradigm, and support rule engine and BPMN-based workflows. MAS should be implemented in JIAC (Java) and Node.js that can be used to develop portable user interface applications that can run on literally any device, using HTML5[1]. Edge Agents implements so called Near Sensor Agents, and provide message handling from them to the core system. DSS supports both MAS BDI component and Near Sensor Agents, and provide transformation of messages to common message (via threat lookup and mapping services) and notification mechanisms (communication over message bus with agents and other system components). **Rule engine** should support non-agent processes either via Node.js or JBOSS drools. It might have scheduler support (JBOSS Drools). **Scheduler**, or IoT based timer, is required to schedule processes, e.g., to pull data from sensor network. Node.js via Node-RED already includes schedule to pull data from services. **Messaging** should be provided: (i) between agents or processes, (ii) from sensors to Near Sensor MQTT broker, (iii) to a main message broker serving system applications. MQTT brokers are suitable and required, especially if sensor networks become offline. Various ways of messaging could be implemented:

- **Message routing**, supported by Message brokers (Apache Kafka, MQTT brokers: Mosca or Mosquitto; connectors to Kakka to Node.js) and Broker clients (Node-jst (JavaScript Template));
- **Message channeling** via PubSub channel;
- **Message transformation/ translation** using semantic annotation (e.g., transformation of unstructured input to output format based on desired ontology and vocabularies), content enrichment (e.g., resolving the semantic location). Some examples of tools to support EIP functionalities are Node-RED, Apache Camel, etc.

Sensor Registry and services need to provide sensor configuration based on both core and domain ontologies. Sensor services could be sensor configuration service, sensor observation

[1] source: http://solidcon.com/internet-of-things-2015/public/schedule/detail/40797

service, sensor planning service, notification service, etc. Models for sensor descriptions and observations could be based on SSN (Semantic Sensor Network). We might also need **data transformation library** to transform incoming messages from sensors to a desired common output format, and **threat terminology** to translate log message info or device messages to both threat and common terminology. One example of terminology to be used is LOV4IoT terminology (see: https://github.com/LOV4IoT).

5. IoT4Industry Agent Architecture Overview

Figure 4 presents teh *IoT4Industry* agent architecture, distinguishing between a core system and associated “near sensor” networks. The **core system** supports centralized and distributed messaging for applications and enables monitoring, security and decision support. The “**near sensor**” supports Near Sensor Edge agents and device connectors that manage collection (schedule, filter) and semantically transform observations, and forward them to the core system. Alternatively, the core system could transform observations depending on whether that is described in the sensor description. Sensors must be described using a VSD (Virtual Sensor Description), which is based on SensorML. Each sensor requires a unique identifier, deployment info, location, and optionally, topic based control settings.

Figure 4 shows the following layers of the *IoT4Industry* architecture: (i) monitoring layer, (ii) communication layer, (iii) business layer, (iv) application layer, (v) near sensor edge layer, (vi) system edge layer, (vii) service interface layer, and (viii) event layer.

5.1. Monitoring Layer

Monitoring layer connects the near sensor networks with the core system via Message Broker Bridges, which provides subscribe/publish features for the near sensor network. The Near Sensor Edge Layer should include connectors to filter messages or schedule polling, and transform the sensor observations to SensorML format. However, the core system layer might also handle transformations according to the VSD description.

5.2. Communication Layer

Communication is facilitated by the core message broker and near sensor message brokers. Figure 5 gives an overview of the component and messaging. In Figure 6, the message brokers of core and near sensor networks utilize a Message Broker Bridge to enable the

subscribe/publish messaging between Message Brokers (e.g. to connect a Kafka message broker to MQTT). Agents also communicate over the core message broker.

5.3. Business Layer

The *IoT4Industry* infrastructure includes a variety of components: schedule engine, rule engine, workflow engine, mail, analytics and big data components (streaming, non-streaming, Map Reduce for processing and generating large data sets) (Figure 5). Some agent frameworks provide components that can be reused as needed by the core system, such as the scheduling engine, rule engine and workflow engine.

DSS and control: Regarding control topics, the VSD for a smart sensor can match topics to defined sensor settings in the VSD. When the DSS controller issues topics, the matching settings are sent to the sensor connector, which then acts on the particular VSD sensor settings.

Agents and control: The BDI agent framework supports DSS and sensor control functionalities. The agent framework includes components for scheduling, workflow, rules and messaging. Agents following particular workflows might post process observations using analytics components (computational server) to further transform observations, provide aggregation or summaries, or metrics. Agents might require post processing analysis before decisions can be made. Sending the results to the message bus would enable agents or other processes to store semantic views (Figure 6) or store in the big data repository and make the results available to waiting agents. Smart sensors might be controlled by the agent based DSS. In the monitoring layer, the sensor connector would subscribe to control topics posted by the core system to the message broker. The control messages contain control fragments based on SensorML.

Security control: Control settings of any virtual sensor can be described in the VSD.

5.4. Core System Edge Layer and Near Sensor Edge Layer

The edge agents are concerned with connecting the virtual sensors of the monitoring layer and facilitating the collection of observations from virtual sensors (sensing) or the control of virtual sensors (actuators). The edge agents can exist in either the core system or the near sensor networks. For example, in a near sensor network, the Node-RED tool creates connectors that are run as edge agents to connect to sensors whereby observations are either polled by the connector or pushed by the sensor to the connector. Another lightweight agent framework e.g. Node.js thingjs-agent could provide more BDI-like agent infrastructure.

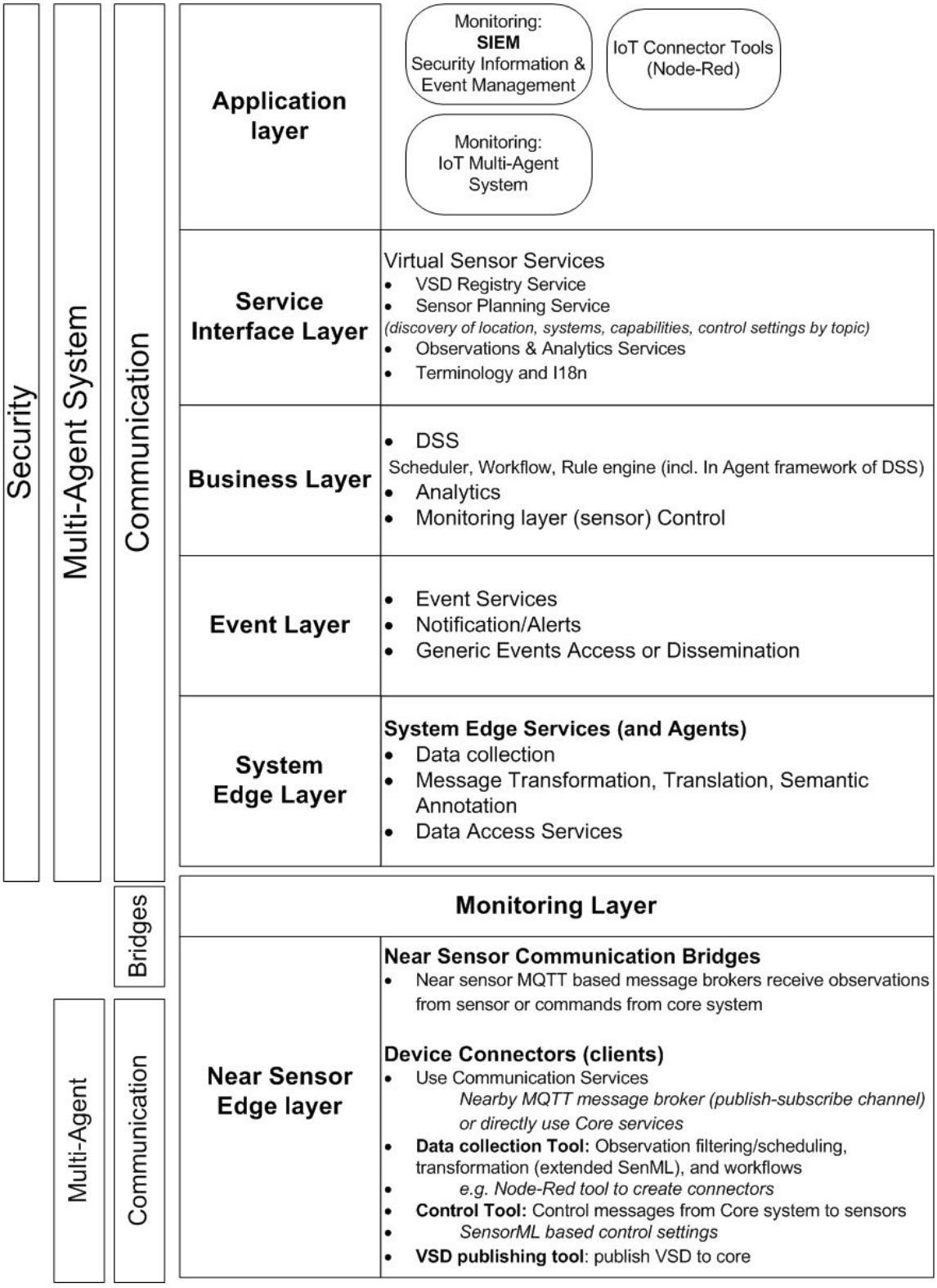

Figure 4: Architecture Overview

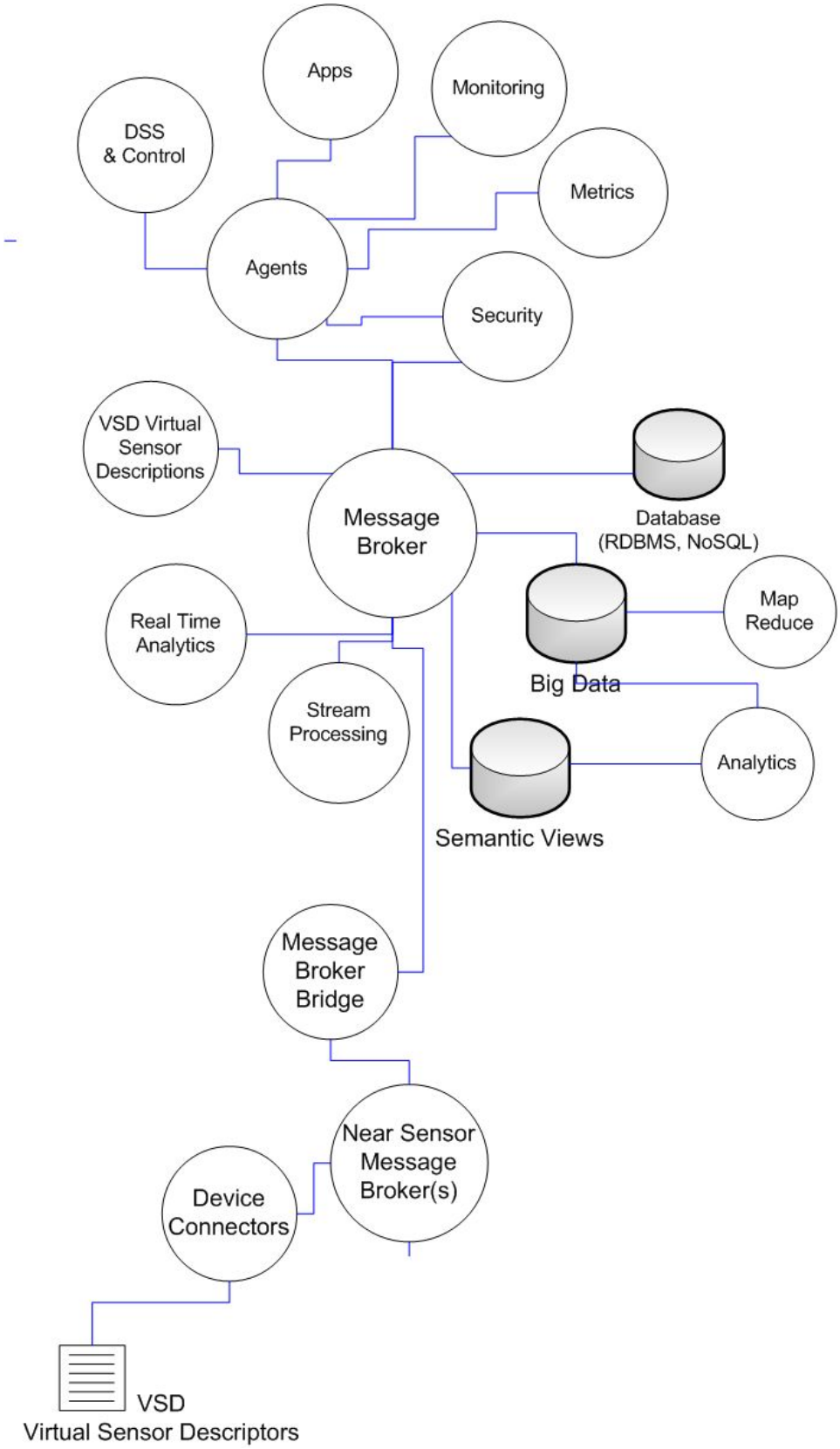

Figure 5: Architectural overview of important components

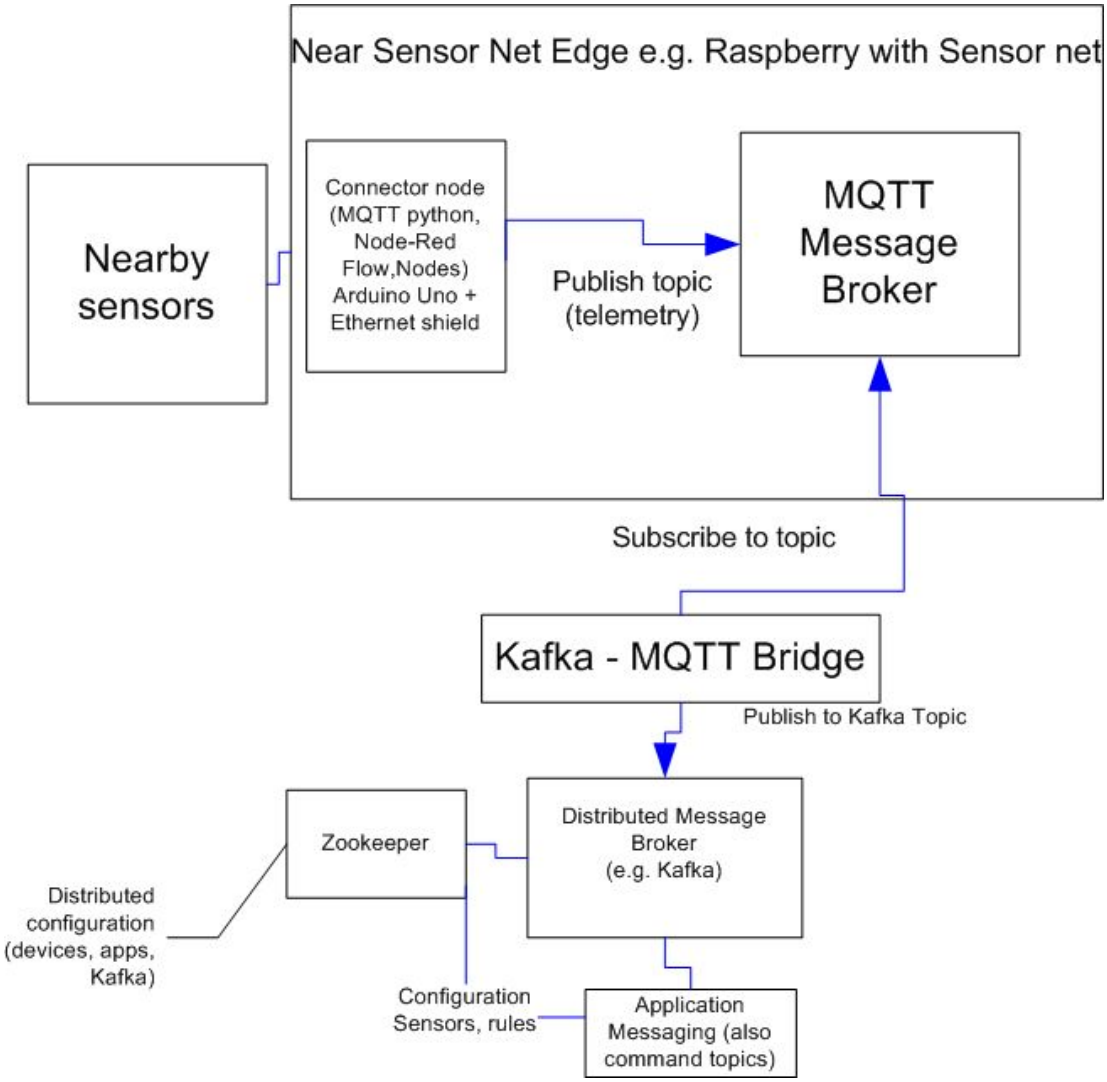

Figure 6: Bridging message brokers of the core system with near-sensor networks

6. Semantic Models and Terminologies in IoT4Industry

Semantic models and terminologies support the description of sensor and sensor-based systems, and the semantic annotation and transformation of observations emitted by the monitoring layer. In this section, we have a closer look at the following semantic models, terminologies and services in *IoT4Industry*:

- The VSD (Virtual Sensor Description) model and services: It is expected that sensors or other monitoring layer components comply to a VSD;
- The Observations (observation events) model and services:
    - the edge agents need to transform and semantically annotate incoming observations or notifications. The initial model should be updated according to the needs of the DSS and database services;
    - DSS and control layer need to support sensor planning, in regard to threat mitigation approaches.
- Contributing internal models (location, monitoring network systems, threat related);
- Terminologies supporting models and services, including DSS (threats);
- Semantic modeling;
- Threat Modeling.

6.1. Virtual Sensor Description (VSD) Model and Services

A virtual sensor is anything that can report observations coming from other sensors, persons, objects, or a system of sensors. Consequently, any monitoring layer component can be described and sensorized by humans (security guards) or logging components, etc. The VSD model also supports the DSS services, as well as the edge agents to transform observation messages (content enrichment/ code translation from a VSD).

There are strong interrelationships among the standards, e.g. the Semantic Sensor Network Ontology (SSN[2]) ontology was influenced by the SensorML[3] standard. Unfortunately, the SSN does not describe important features that were considered by SensorML, such as sensor control topics and settings, observation data processing and data access. Hence, the VSD descriptions in IoT4Industry are based on SensorML and SSN using description files as reference from IoT

[2] SSN Semantic Sensor Network Ontology http://www.w3.org/2005/Incubator/ssn/ssnx/ssn

[3] OGC SensorML The Open GeoSpatial Consortium (OGC) Sensor Model Language (SensorML)

projects such as GSN[4], OpenIOT with X-GSN[5] and M3 framework[6] (the examples are provided in Appendix B). The implementation of VSD in IoT4Industry is based on inputs from DSS and specifications of controlling layer components, supporting services such as:

- Discovery of sensor, sensor systems, and sensor processes;
- Discovery of sensor observations (store queries);
- Sensor location and system services;
- Subscription to sensor observations or alerts;
- Sensor planning based on capabilities (measurement and control) and deployment info supporting DSS mitigation approaches in coordination with the control layer.
    - DSS/ Control layer services for task interoperability.

The following lists several VSD features suggested by our implementation team:

- GUID (Global Unique IDentifier) should be mapped to a unique security token (needs to be separated from the description);
- Resource type description needs to be used;
- Monitoring layer category (e.g., logs, ambient sensors, smart sensors, etc.) to be implemented;
- Processing class;
- Observation data access info, e.g. to build queries;
- Deployment information including the following: location (geo and place), part of sensor system, e.g., electric substation A-100, service related info (life, maintenance), responsible persons and contact info (DSS mitigation task asks persons to check this info), activity status (e.g. undeployed sensor), etc.
- Measurements, entity detection, notification, logging;
- Control features: Control taking features and mapping to common codes
- Classification for resource discovery (sensor planning service, lookup)
    - Classification supporting communication message broker channel/ edge agent handling
    - Classification to support lookup services
- Control descriptions relating to control topics, e.g. threat levels

[4] GSN sensor descriptions- https://github.com/LSIR/gsn/blob/master/virtual-sensors/samples/
[5] OpenIoT with X-GSN sensor descriptions https://github.com/OpenIotOrg/openiot/wiki/X-GSN-Use
[6] M3 Framework http://semantic-web-of-things.appspot.com/?p=architecture

    - Support DSS, control layer and edge agent. Example: For each threat level (high, low, normal), what are sensor settings, access rates etc.?
- Notification code mappings: proprietary codes to common codes, e.g. system logs, sensor events;
- Implementation issues:
    - Mapping to support transformation of proprietary terminologies
    - Threat mitigation: if sensor needs to be checked for malfunction? how often?
- Particular Field element values (sensor classification, control topics, virtual sensor resource types, system ID, location ID) should be based on standard terminologies such as (LOV4IoT) Open Link vocabularies for IoT and domain model terminologies, as needed.

6.2. Observations Model and Services

6.2.1. Semantic annotation of monitoring component messages (Observations)

Observations will be semantically annotated using an extended version of Sensor Markup Language (SenML)[7], enriched with the matching features from the component's VDE, and translated via VDE bindings to common domain terminologies. In Table 1, the extended version of the SenML notation is described, and SenML JSON example is presented. SenML was chosen because it is a lightweight model that can be easily handled by upstream processes and can be easily extended. It is available as JSON or XML format. Alternatively it can be transformed to RDF or the SSN ontology.

The "*zone*" element is an extension used to communicate additional semantic annotations from the edge agents to upstream processes including message broker. Additional annotations can include VSD metadata and message channel topic(s) for message subscribers (other edge agents, logging edge agent, DSS or database services). The "*zone*" element was used in the M3 Framework project to communicate the domain name of the sensor.

In the following, we give several implementation issues to be considered: The DSS and other subscribers should propose additional elements for both the "*zone*" and "*e*" elements. Sensor clients might send notification to inform the incoming edge agents about the monitoring layer components (e.g. logging components, ambient sensors, smart sensors). Otherwise an edge

[7] SenML http://tools.ietf.org/html/draft-jennings-senml-10

agent might insert it from the VSD description to support message subscribers (other edge agents, or DSS and database services.)

Table 1: Extended SenML Observations model

Extended SenML	**JSON**	**Types**	**Notes**
Zone with parameters	*(Note: This is NOT part of the SenML standard)* Zone wraps the entire message body and enables the edge agents to semantically annotate the observations as required by the DSS and database services. Additional metadata can be included such as the VSD description of the sensor/monitoring component, e.g. location info, sensor type, sensor system to support database queries. It can support edge agent messaging the message broker.		
Name of zone	name		Name of zone is a means to partition the messages as needed, especially if there are subsystems in the network.
Topic	topic		It uses topic terminology and code space as prefix or URI. If there are more than one topic, it uses comma delimited data. Topics are relevant for message subscribers and can indicate what should be done next on the message according to the potential subscribers (DSS, database, other edge agents and services). An upstream edge agent, or subscriber, might further update or remove topics as needed.
Location	location		It uses terminology and code space as prefix or URI. The location corresponds to the location place terminology, which is defined for this domain. Note that if a sensor is moved, sensor location might change in the future. Therefore the current location should be annotated. The GPS coordinates might be included in alternative fields, if available. A VSD service will provide location and system lookup features.
Monitoring Layer	cat_mon		DSS or edge agents need to know the

Category			category of the monitoring layer component.
Security token	stoken		Only if the security mechanism requires to record a security token in the messages
Date and time when the observation was received	dt_recd		Date and time of the observation when it was received by the edge agent. It is possible that the observation was performed much earlier - due to outage or reason for late reporting by sensor. This is something that the DSS should know about in order to update the Bayesian or PCA models.
History	history (*dt_mod, modby*, *step*…)		A set of parameters to indicate any progress in a workflow performed by multiple processes (*date time modified, modified by, step*)
SenML Elements		**Types**	**Notes**
Base Name	bn	URI/ UUID	Unique Sensor Identifier is unique to the sensor network. An additional code space can be applied to make it more globally unique using the "*cs"* element. VSD descriptor of sensor ID, registered with a security token.
Sensor network code space or namespace	cs	URI	(**Note:** This is NOT part of the SenML standard) Code space or namespace of the sensor can be used to avoid sensor ID collisions in a large network. This might be annotated by using "pr" element for prefix.
Base Time	bt	dateTime	Default Base date and time of observation, unless it is provided as an observation parameter.
Base Units	bu	int	Default base units of observations, unless it is provided as an observation parameter.
Version	ver	int	Version of the sensor observation. Default value is "1".
Measurement or Parameters	e	Collection	The element "e" is a set of elements. However, only one value is possible (string,

			integer, float, Boolean).
Name	n	URI	Use terminology and code space as prefix or URI. The name of the measurement can be used with namespace, e.g. *aaa:temperature, aaa:salinity*
Units	u	string	The convention should include the units terminology namespace, e.g *UCUM:voltag*e, or a local project namespace.
Value	v	float	Float value
String Value	sv	string	String value
Boolean Value	bv	Boole an	Boolean value
Value Sum	s	float	Value sum value
Resource Typed Value	rv		(**Note:** This is NOT part of the SenML standard) Resource type value uses code space prefix to reference to a terminology.
Time	t	dateTi me	Agreed upon date time format; observation time.
Update Time	ut	dateTi me	Expected next observation time (optional).
Resource Type	rt	URI (Type of observ ation)	It uses topic terminology and code space as prefix or URI, e.g., type of sensor, monitoring layer component) The Resource type namespace should also be included.

6.2.2. SenML JSON example

Here is an example of observation model in SenML JSON. Element “cs” is the sensor namespace; “zone” provides additional semantic annotations for other message subscribers, e.g. edge agents, loggers, DSS, database; “bn” gives a base name (URI), etc.

```
{“zone”:
```

```
        name="scizone:AAAA"
        location="AA-11-22"
        topics:"scitopic:XYZ",
        cat_mon:"",
        dt_recd:"",
        history:{ },
        stoken:"sen01@test.com"
        otherAttributesFromVSDRequiredByDSS=""
        {"e":[
              {"n": "aaa:temperature", t="1374069830362","u": "UCUM:Cel",
              "v":22},
              {"n": "aaa:salinity", t="1374069830362", "v":0.031}
              ],
              "bn":"http://myiot/sensors#comboTempSalinitySensor0001/"
              "rt":"bbb:EnvNode"
              "cs":"http://myiot/sensors"
        }
}
```

6.2.3. SenML XSD Schema

The SenML XSD schema is also provided to better understand the observation model requirements. The "zone" element is not shown and should be updated for implementation needs and requirements from other upstream components, e.g. DSS and database services.

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            targetNamespace="urn:ietf:params:xml:ns:senml"
            xmlns:ns1="urn:ietf:params:xml:ns:senml">
   <xs:element name="e">
     <xs:complexType>
       <xs:attribute name="n" type="xs:string" minOccurs="0"/>
       <xs:attribute name="u" type="xs:string" minOccurs="0"/>
       <xs:attribute name="v" type="xs:float" minOccurs="0"/>
       <xs:attribute name="sv" type="xs:string" minOccurs="0"/>
     <!-- non standard element, to handle codespace or namespace of sensor →
     <xs:attribute name="rv" type="xs:string" minOccurs="0"/>
       <xs:attribute name="bv" type="xs:boolean" minOccurs="0"/>
       <xs:attribute name="s" type="xs:decimal" minOccurs="0"/>
       <xs:attribute name="t" type="xs:int" minOccurs="0"/>
       <xs:attribute name="ut" type="xs:int" minOccurs="0"/>
     </xs:complexType>
   </xs:element>
   <xs:element name="senml">
     <xs:complexType>
     <xs:sequence>
         <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns1:e"/>
```

```
            </xs:sequence>
            <xs:attribute name="bn" type="xs:string"/>
            <xs:attribute name="bt" type="xs:int"/>
            <xs:attribute name="bu" type="xs:string"/>
            <xs:attribute name="ver" type="xs:int"/>
<!-- non standard element, to handle codespace or namespace of sensor →
            <xs:attribute name="cs" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:schema>
```

6.3. Contributing Internal Models (Location, Monitoring Network Systems, Threat)

Here are some contributing internal models to support the VSD services, DSS and controlling layers.

6.3.1. Model describing physical location and monitoring network systems

The VSD model will support the relationships between virtual sensors and locations. The following models shown in Table 2 (The Location Model) and Table 3 (The Virtual Sensor System Organization Model) can be further improved depending on the requirements of the DSS and controlling layers.

Table 2: Location Model

UUID	Unique ID
Local ID	Using local identifier (query on this) Unique for site
Label	Human readable label
I18n	Internationalization code
locationType	Type of location; it requires location type terminology, e.g. electrical substation, room...
partOfLocation	Part of a parent location, refers UUID of linking site.
tags	Tags to find one or more locations. Tags might not be unique to the location.
geolocation	Geo location

Table 3: Virtual Sensor System Organization Model

UUID	Unique ID
Local ID	Using local identifier (query on this)
Label	Human readable label
I18n	Internationalization code
systemType	Type of system; it requires system type terminology and refers to monitoring layer components. It could be a sensorized human system.
partOfSystem	Part of a system, that refers to UUID of linking site
location	UUID or local ID if unique
tags	Tags to find one or more systems. Tags might not be unique to the system.
features	Features that DSS requires to support queries

These models will support the VSD services for the location and system discovery services. The goal is to provide the DSS the ability to determine sensors that are part of locations or systems, and vice versa. Domain vocabularies, such as site's place, local identifier and names should include a local namespace prefix. Note that human operators will need to view human readable information in order to act on any alert or log message. A supporting location lookup service should support the DSS service and include a part of relationship to link locations.

6.3.2. Model describing systems of collaborating sensors and other monitoring layer components

The monitoring layer consists of virtual sensors, which are described by VSD. The VSD specifies the system and supports system related queries in the VSD services. One or more sensor systems, as a collection of collaboration sensors, should be also assigned to location e.g. a particular electrical sub-station. Additionally, logging related components of the monitoring layer can be described as part of a system.

6.3.3. Model describing sensor settings corresponding to threat levels

This model describes threat levels and threat types, using SenML.

6.4. Semantic Modeling in IoT

6.4.1. Models supporting observations messaging

These models provide a common treatment of observation messages from the monitoring layer. The first message transformation can include content enrichment, semantic annotation, code translations to common system codes, e.g. monitoring component events from log messages (error, alerts, etc.), smart camera (human, gun, etc.) The role of Incoming Observation Edge Agent is to collect observations that can be further sub-typed depending on the monitoring layer component, e.g. log component versus smart camera. Semantic partitioning of responsible edge agents can be based on the message channel topic of the incoming message and the subscribing observation edge agent. In general, the data is collected and transformed to a common semantic model, then published again to a message broker and received by other subscribers, other edge agents, DSS processes or database subscribers.

Incoming observations can be measured or even transformed into notifications (logs, smart cameras reporting Entity Detection Objects (EDO); location information and sensor capabilities, etc.) Furthermore, the common terminologies and mappings (bindings) to proprietary observation content need to be defined, e.g. log file notifications, control topics and settings, measurement units, EDO (such as human, gun, or fire, etc.). The transformation of incoming messages includes the translation of proprietary codes to common system codes; again these bindings might be defined in the VSD for the particular sensor – this pertains especially to the smart sensors that can be controlled, or utilize proprietary notification codes or entity detection codes. Note that if the DSS layer determines that there is a threat of fire, therefore, the EDO term is reused to report alerts to human operators, likewise, in case of human threats, the EDO for human or gun is reported. Consequently, EDOs are possible threat attributes or threats themselves.

Table 4 presents basic requirements and implementation issues regarding incoming observations from the monitoring layer.

Table 4: Incoming observations sent from the monitoring layer to the edge agents

Expected:

UUID (Unique ID of a sensor) to be used for the IBE (Identity by Encryption)
Unique (IBE) security token to be used
Topic or category of component layer category to facilitate the edge agents
Sensor location to be verified against the VSD
Expected next observation time (optional)
Validation:
Are UUID and security token the same?
UUID should have corresponding VSD record in registry; Content e.g. measurement, should be according to the VSD
Malformed or invalid observation message content; Some rules required to determine if sensor is malfunctioning

6.4.2. Models supporting sensor control

The VSD model, incorporating SenML elements could support sensor control by the association of control topics to sensor settings. A sensor client (connector) is required to interpret these commands. The VSD model can provide task capabilities and address interoperability aspects. Models supporting the decision and control layer need to address threat levels or threat types. In addition, the controlling layer can use the VSD description for sensor tasks related to control topics. Setting modes can be associated with control topics, e.g. settings relevant to one or more threat levels. Control settings are assigned to a control topic (threat level) as illustrated in the following example.

```
<sml:modes>
       <sml:ModeChoice id="THREAT_LEVEL_MODE">
       <sml:mode>
              <sml:Mode gml:id=" scscontrol:lowThreat">
                     <gml:description> Setting when nothing has been detected
</gml:description>
                     <gml:name>Low Threat Mode</gml:name>
                     <sml:configuration>
                     <sml:Settings>
                            <sml:setValue
                     ref="parameters/settings/samplingRate">0.1</sml:setValue>
                            <sml:setValue ref="parameters/settings/gain">1.0</sml:setValue>
                     </sml:Settings>
```

```
                </sml:configuration>
            </sml:Mode>
</sml:mode>
<sml:mode>
        <sml:Mode gml:id="scscontol:highThreat">
                <gml:description> Setting when something has been detected
        </gml:description>
                <gml:name>High Threat Mode</gml:name>
                <sml:configuration>
                <sml:Settings>
                        <sml:setValue
                ref="parameters/settings/samplingRate">10.0</sml:setValue>
                        <sml:setValue ref="parameters/settings/gain">2.5</sml:setValue>
                </sml:Settings>
                </sml:configuration>
        </sml:Mode>
</sml:mode>
</sml:ModeChoice>
</sml:modes>
```

Figure 7 illustrates the VSD control aspects, which are possible VSD control topics, such as: threat level topics (e.g. threat modes of sensors), threat types (an outage, malfunction, fire, etc., diagnostic checks (perform alive check). The VSD Threat Mode Description describes threat modes and associates each threat level to tasks that should be undertaken. It sends VSD tasks descriptions to the Control Layer, which together with the Decision Support Layer, sends sensor settings messages to the near sensor message busses and to the sensor client. Sensor client interprets the messages received and, based on that, performs the tasks.

The following is an example of sensor control using VSD control topics. The DSS mitigation strategies are based on a sensor control terminology from the Control Layer. VSD services such as sensor planning service, are required to determine capabilities of sensor(s) for performing specific tasks of the sensors (for example, to move camera), or determine which sensors collaborate in a sensor system in order to perform threat assessments.

Utilizing on the concept of *fan-out* messages, we can explain the following situation: the Control System sends commands, based on the sensor control terminology, into the Message Broker (topics) and Control Layer components (smart sensor driver/client), receive messages that are subscribed to *fan-out* of commands. For example, Microsoft Azure IoT[8] describes the use of *fan-out* messages by IoT sensors, as shown in Figure 8. Possibly, a Near Sensor Message Broker, e.g. lighter weight MQTT Message Broker has a bridge to a Central Message Broker Server.

[8] Microsoft Azure Fan Out messages https://msdn.microsoft.com/en-us/magazine/jj133819.aspx

<u>Control topics</u> as associated with VSD sensor settings that are interpreted and executed by sensor clients.

Possible VSD control topics:

- Threat level (threat modes of sensor) Need a normal level !!
- Threat type (Is there an outage, malfunction, fire?)
- Diagnostic checks (Perform alive check)

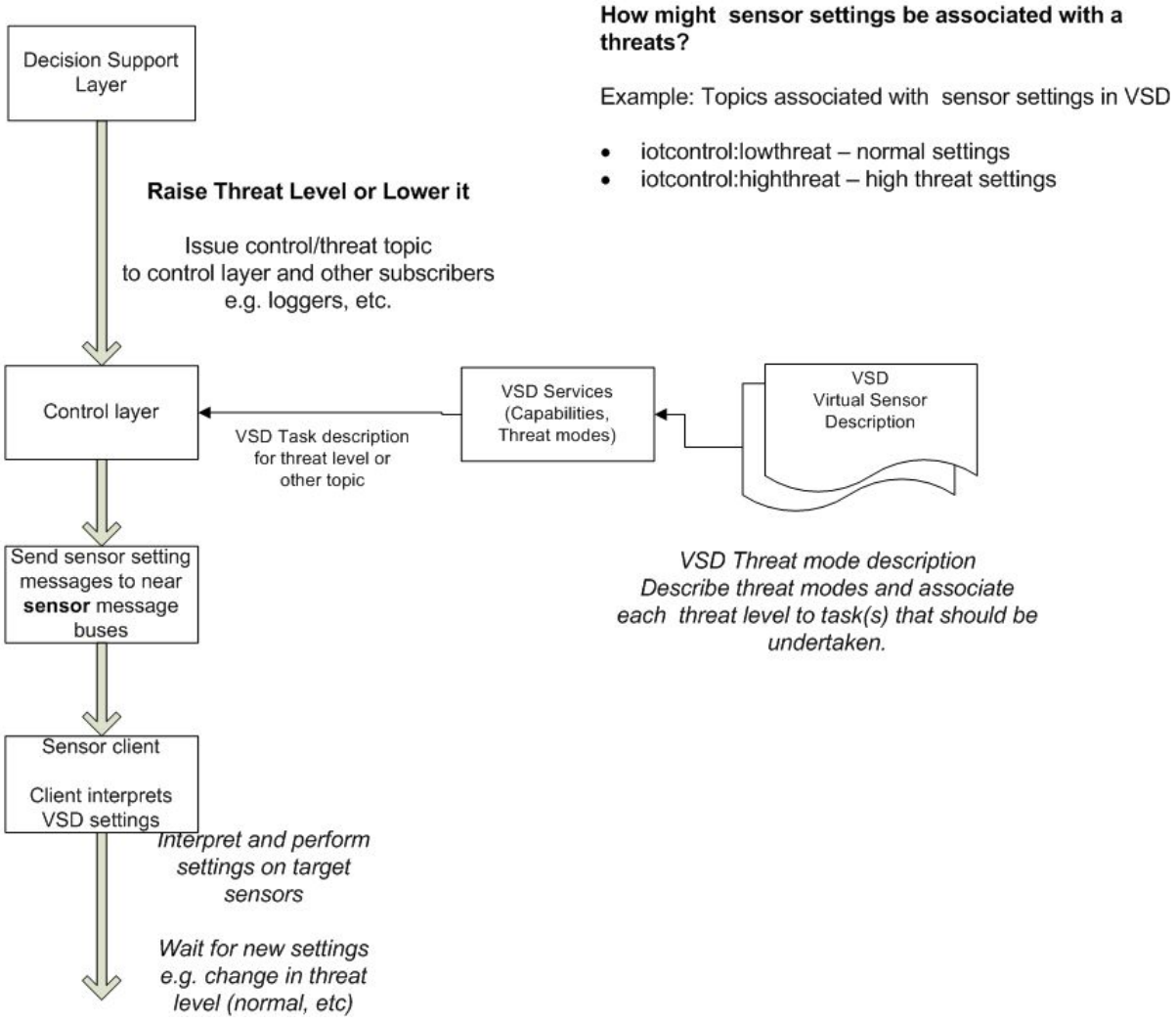

Figure 7: VSD control topics

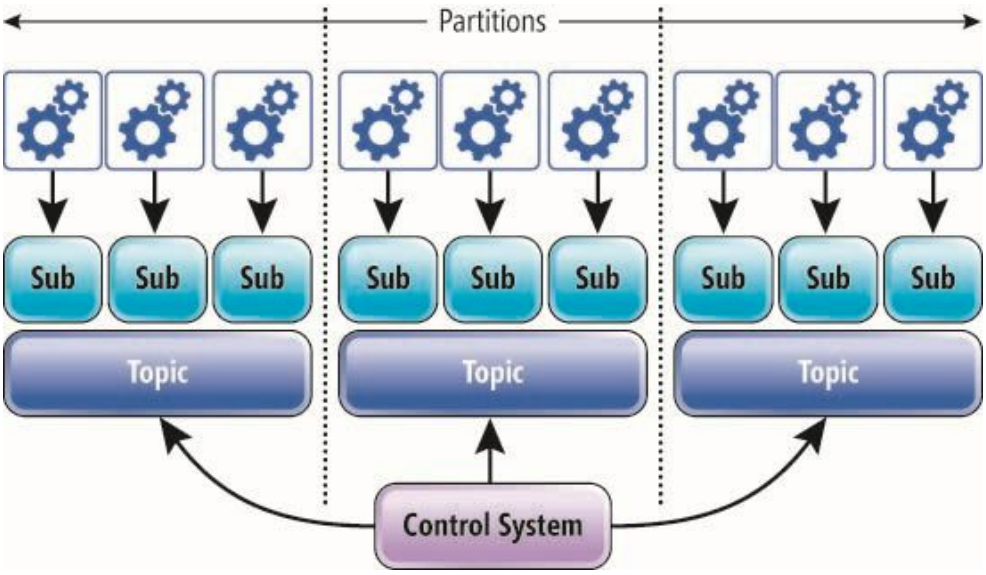

Figure 8: The Control System sends commands to the Message Broker (topics), and IoT devices receive messages for *fan-out* commands to which they have been subscribed (source: https://msdn.microsoft.com/en-us/magazine/jj133819.aspx)

Sensor control topics will be forwarded by **Outgoing Edge Agents** to the Message Bus and onward to a Near Sensor Message Bus for retrieval by sensor clients. Message body contains control setting information about the sensor from the VSD. Lastly, the SSN ontology does not model sensor control (tasking) features. Oracle API, Wolfram sensor framework, or SWE/SensorML/ support control of sensors.

6.5. Contributing Terminologies

Monitoring Layer components might use proprietary terminologies. To support service queries and relationships among components, measurement units, etc., the VSD requires standard or domain terminologies. Edge agents transform incoming messages such as: translating proprietary information from incoming notifications and observations, inserting unit codes. Terminologies are used not only to support communication between components, but also to communicate information to human operators; therefore human intelligible labels are required. Some resources are listed, such as the Linked Open Vocabularies for IoT (LOV4IOT). However, sensor providers should indicate their preferred standard terminologies and from there consensus can be achieved during implementation.

Table 5 lists terminology requirements and suggestions for further discussion on their use in *IoT4Industry*.

Table 5: Purpose and suggestions on possibly contributing terminologies in *IoT4Industry*

Terminology	**Purpose and suggestions**
Entity Detection Objects (EDO) Terminology	**Purpose**: Entity detection is performed by particular smart sensors that emit an event describing the detected entity. Proprietary terminologies likely describe an entity and need to be translated to a common standard terminology. The DSS might create rules based on threat attributes that are also EDOs. Some attributes can be used as threats attributes (TDA). **Example**: A smart sensor might detect a dog, human, gun, or fire, certain objects or even threat attributes, depending on intelligence of monitoring component. **Suggestion**: It can be extended towards LOV4IOT (Linked Open Vocabularies for IoT): http://sensormeasurement.appspot.com/documentation/NomenclatureSensorData.pdf http://sensormeasurement.appspot.com/?p=ontologies
Sensor Control Topic terminology	**Purpose**: DSS and Control Layer require common control terminology. Outgoing Edge Agents need mapping to sensor proprietary control terminology or API. Possibly Message Broker can be used to provide communication between Edge Agents and Sensor Clients. **Example**: A means to communicate between components and also with human operators **Suggestions***:* SensorML resources, Wolfram or Oracle API. Semantic Sensor Network ontology (SSN) does not yet model sensor control. Next step is to identify what tasks sensors can do and derive simple task terminology.
Units of Measure	UCUM (Unified Code for Units of Measure) http://unitsofmeasure.org/trac/ LOV4IoT

	http://sensormeasurement.appspot.com/documentation/NomenclatureSensorData.pdf http://sensormeasurement.appspot.com/?p=ontologies BIPM http://www.bipm.org/en/measurement-units/ CUU http://physics.nist.gov/cuu/Units/index.html
Location vocabulary and Sensor system terminologies	**Purpose**: A physical site map should be constructed, and place names identified uniquely. Likewise a Logical Sensor System Vocabulary should be created, with unique identifiers for each sensor system. A supporting location lookup service should support the DSS service and include a part of relationship to link locations. **Example**: A local namespace should be prefixed when using a place identifier. **Suggestions***:* Requires supporting location type and system type terminologies
Resource Types	Sensor type: SensorML, GSN and LOV4IoT Location type: local terminology System type: local terminology
Monitoring layer Notification terminology	Support translation of monitoring layer notifications from particular components by Edge Agents Log components messages
System Notification and Task terminology	A means to communicate between components and also with human operators.
Threat Detection Attributes (TDA) terminology	This has relevance for the DSS for codifying the threat rules and also understanding attributes linked to a threat. What attributes might imply a threat? These might be used as part of any DSS rules. For example, which EDOs might be considered threats? Reuse EDO identifiers (using namespace) as necessary. Also, items from the Threat Landscape Taxonomy might be relevant.

Threat Landscape Taxonomy	See Threat modeling from Appendix C

7. Services in IoT4Industry

In the following, we lists possible VSD services supporting both sensor system components (Table 6) and Edge Agents (Table 7).

Table 6: VSD Services supporting sensor systems components

VSD Registry Service	Supports registration of virtual sensors and systems. Add, update VSDs (no delete, only un-deploy sensor).
Sensor Planning Service	Supports: - Sensor discovery (uses system and location discovery services) - Deployment and activity status (deactivated sensor) - System discovery (collection of components/sensors) - Location discovery service - Site and location discovery - Other activities of Edge Agents, Control layer, DSS services
System Discovery and Management	Find systems and their sensors, or vice versa. Find systems and children or parent corresponding to a sensor. Find sibling sensors in a system or parent system. Services should utilize both location and system to provide the DSS with desired information as described in the VSD section. A VSD service should be provided to lookup parent system of a sensor or children or siblings as well as sensors associated with that query. Also, based on location, find systems and sensors: ● Based on location, find sensors (option: include child locations) ● Find systems or sensors based on a location
Location Discovery and Management	Find locations and sensors associated with locations and child locations. Find sibling sensors of a location. Database service or VSD service (TBD) can lookup: ● Location info ● Sub-locations of location

	● Parent location of a location

Table 7: VSD Services supporting Edge Agents

Sensor Planning Service	● Find sensor descriptions and capabilities by sensor ID, deployment info, e.g. location, sensor system, control topic and settings etc. Who to contact in case of possible threat e.g. OUTAGE or MALFUNCTION threat. ● Get sensor or sensor network capabilities about measurements, control info (e.g. sensor tasks and mapping from proprietary actions to system control topics and settings are in VSD configuration), deployment and responsibility info. ● Location and system discovery ● Find sensors of systems or locations
Transformation Support Services	● The VSD services should support the access to the VSD for purposes of content enrichment, including calculations, code translation (sensor events, etc.), and semantic annotation. ● Transform proprietary formatted and coded observations to a target standard model and format such as SenML in JSON or XML. ● Alternative models are possible.
Semantic Annotation Service	Annotation Agent handling transformation of incoming messages to a particular target output model. The SenML standard provides a lightweight model for example, instead of Semantic Sensor Network (SSN).

8. Conclusion and Future Work

IoT4Industry agent-based multilayer architecture has been proposed as a scalable foundation for industrial IoT/WoT applications. The usage of established protocols and ontologies allows for integration into existing networks or plants without raising the need for new (expensive) hardware. At the same time, the distribution on numerous agents enables the usage of inexpensive replaceable endpoints. Due to the focus on existing open source solutions, the components are reliable and can easily be integrated, exchanged or adjusted.

Nevertheless, closed source components will have to offer interfaces for extracting data and the IoT community needs to make data publically available and link it to the existing knowledge to create better-connected LD containers in order to exploit the advantages of Linked Data Platform.

Furthermore, we have to regard notable challenges related to security, trust and privacy in order to use all benefits responsibly. In the course of this project, we addressed the most important issues in this field and proposed viable solutions. Moreover, the laboratory setup and scenario need to be extended by adding more sensors and devices and approaching automation where applicable. For example, more 3D printers could be added and enhanced by robot arms, conveyor belts and racks in order to automatically process and distribute print orders and label and pack up finished jobs.

Finally, we should take a look at end-user interaction. In case decision making is not (only) done by the agent but (also) by the end-user, demand for appropriate information and interfaces emerges. In the context of 3D printing, there could be a web interface for uploading models, adjusting print settings, monitoring the print process (e.g. remaining time, video stream), manual abortion, etc. Some of these features have been implemented already (e.g. video stream, remote access to the 3D printer), but we need to add more automated processing, error detection and merge functionalities into a single end point.

Finally, we would like to add some lessons learned notes, drawn from our experience in *IoT4Industry* project implementation:

- Focus should be always on the tools supporting near sensor networks to connect with sensors and process observations.

- Use IoT community based connectors in the near sensor networks using popular tools, such as Node-RED. The core system edge connectors likely cannot support a great variety of devices.
- Use IoT tools rather than generic tools to avoid reinventing transformation, data collection management (filtering, push, pull), and control aspects.
- Use MQTT based Message Brokers to support virtual sensors in near sensor networks as these are more suitable to the IoT; for example, operating conditions. Eventually, MQTT based Message Brokers will be scaleable to include in the core sensor system.

References

[BUTLER D3.2] Integrated System Architecture and Initial Pervasive BUTLER Proof of Concept. October 2013. Online available: http://www.iot-butler.eu/wp-content/uploads/downloads/2013/10/D3.2-Integrated-System-Architecture-v1.50.pdf

[CALA01] G. Cachon, & M. Lariviere. 2001. Contracting to assure supply: How to share demand forecasts in a supply chain. Management Science. Vol.47, 629-646.

[CALA99] G. Cachon, & M. Lariviere. 1999. Capacity choice and allocation: Strategic behavior and supply chain performance. Management Science. Vol.45, 1091-1108

[FIVR96] Filar, J. and Vrieze, K., 1996. Competitive Markov decision processes. Springer-Verlag.

[SECURE] SECURE: Semantics Empowered Rescue Environment (Demonstration Paper). Online available: http://www.knoesis.org/library/resource.php?id=1631

Appendix A: Overview of Agent Technologies for IoT

The following are the best examples of agent frameworks supporting the Near Sensor Networks.

JIAC (Java-based Intelligent Agent Componentware)

Website: http://www.jiac.de/agent-frameworks/jiac-v/

Architecture: http://repositories.dai-labor.de/sites/jiactng/5.1.5/

JIAC framework supports the design, implementation, and deployment of software agent systems. In addition, it supports development of BDI agents. It can be used with a set of development, configuration and monitoring tools, such as:

- VSDT tool, which is a BPMN editor, and multi-language transformation and workflow simulator;
- Asguard, for controlling distributed multi-agent infrastructures at runtime;
- AWE (Agent World Editor) for modeling and configuring an MAS. It also generates files for deploying the system;
- JIAC Toolipse, which is an integrated development environment containing all above mentioned tools (for more: http://www.jiac.de/development-tools/jiac-toolipse/).

Active development of JIAC and publications (via TU Berlin) target industry scenarios (see: http://www.jiac.de/publications/). Figure A.1.1 shows the JIAC architecture, which includes rule engine, scheduling engine, workflow engine (BPMN), and message broker (ActiveMQ).

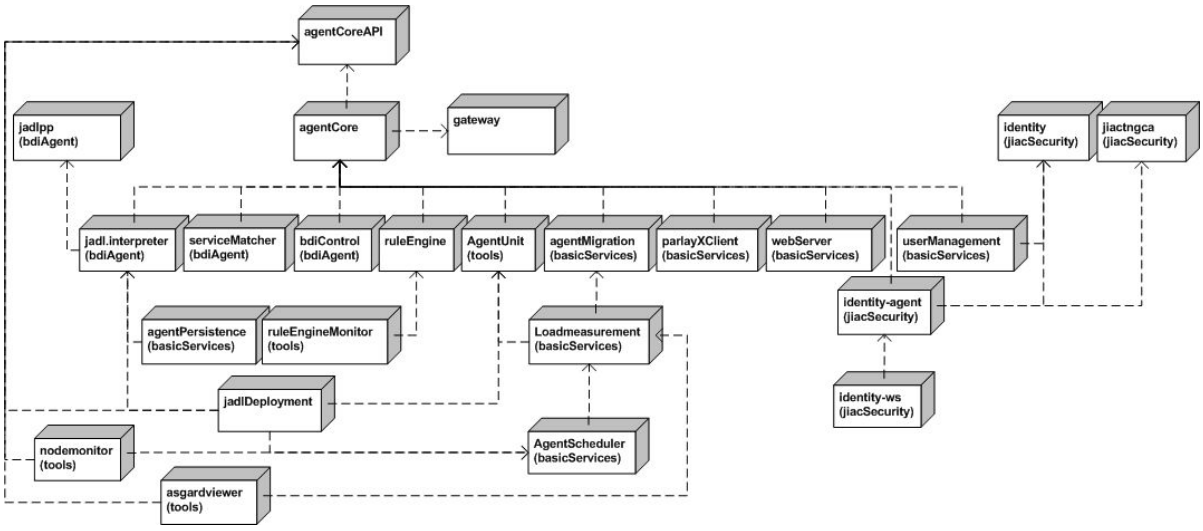

Figure A.1.1: JIAC Architecture Components Overview

Figure A.1..2 gives a view of the JIAC agent with regard to the behaviour control scheme.

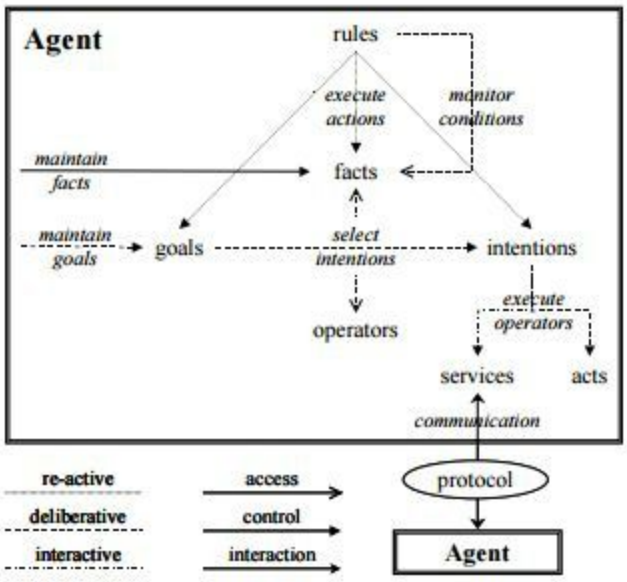

Figure A.1.2: JIAC Behaviour Control Scheme

micro JIAC

Website: http://www.jiac.de/agent-frameworks/microjiac/

Source: https://github.com/mcpat/microjiac-public

micro JIAC is a lightweight multi-agent architecture and framework for running JIAC agents on various devices.

thingjs (thingjs-agent)

Website: https://github.com/thingjs/agent

Installation details: https://www.npmjs.com/package/thingjs-agent

Thing.js is an agent framework written in JavaScript for building IoT applications. It supports Node.js, Browsers (ES5+, Chrome, Safari, Firefox, Opera), Tessel 2, Phonegap/Cordova, JavaScriptCore and other Mobile Containers. It is characterized by the following features:

- Abstractions, Inheritance and Interfaces
- Annotations and Templates
- Passivity and Singletons
- Simple, Series, Parallel, Queue, MapReduce and Waker Primitive Behaviours
- HRRN Scheduling and Micro-containers
- Asynchronous Messaging, Selectors and Filters
- JSON-LD Ontologies and Message Translation

- MQTT Sensors, Actuators and Bridging

AKKA Agent

Website: http://doc.akka.io/docs/akka/snapshot/java/agents.html

AKKA agents are lightweight, reactive agents, used with AKKA actors (actor based system). They come without any built-in rule, scheduling or workflow infrastructure; modeled after Closure agents (for more: http://clojure.org/agents).

AKKA agents are bound to a single storage location for their lifetime, and only allow mutation of that location (to a new state) to occur as a result of an action. Actions dispatched to an agent from another thread will occur in the order they were sent, potentially interleaved with actions dispatched to the same agent from other threads (see URL above for more details).

LogStash agents

Website: https://www.elastic.co/products/logstash

LogStash supports a number of inputs, codecs, filters and outputs:

- Inputs are sources of data; there exists plugins for custom data sources.
- Codecs transform an incoming message format into an internal LogStash representation as well as into a specified output format. These are usually used if the incoming message is not just a single line of text.
- Filters are processing actions on events and allow for modifying events or drop events as they are processed.
- Outputs are destinations where events can be routed.

Furthermore, LogStash centralizes data processing of all types, normalizes varying schema and formats, extends to custom log formats, provides easy creation of plugins for custom data sources. Apache Kafka can be integrated with LogStash (see example: https://github.com/joekiller/logstash-kafka).

Node-RED as Edge Agent Framework

Website: https://github.com/node-red/node-red

These agents are relevant for creating connectors between sensors and a core system that collects observations. Using a graphical tool, connectors can filter and transform messages. They can run according to a schedule (polling); route message stream from sensors to services, etc.

Apache Camel

Website: http://camel.apache.org/

EIP patterns http://camel.apache.org/enterprise-integration-patterns.html

This is relevant to build Edge Agents to transform or route messages (EIP patterns). It can be connected to message brokers such as MQTT based brokers, or Kafka, ZeroMQ.

Appendix B: Overview of Rule Engines

The following are the best examples of relevant rule engines, which demonstrate that Near Sensor Networks using Node.js can be supported with rule framework and coupled with larger servers supporting Java based rule engines. In order to produce outcomes (objects, actions), rules in rule engine's knowledge base are applied either stateful or stateless. Stateless rules create knowledge base either prior to processing rules or as part of the rule process. Rule might be grouped as sets using the rule language, and conditionally executed. They can also be grouped and associated with a session (stateless or stateful), such as "*packaging the rules across multiple files belonging to a package defined in the rule file or rule set*". The knowledge base can be created via DB queries, user input, etc., and it also might be stateful or stateless.

Nools (Javascript, Node.js)

Website:https://www.npmjs.com/package/nools

Source: https://github.com/C2FO/nools

This is relevant for Edge Agents, and useful for integration into Node-RED or Thingjs-Agent.

JBOSS Drools (Java)

Website: http://drools.org/

JBOSS Drools is a Business Rules Management System (BRMS) solution. It provides a core Business Rules Engine (BRE), a web authoring and rules management application (Drools Workbench) and an Eclipse IDE plugin for core development. It supports stateful and stateless knowledge sessions, and can be associated with one or more rule packages. For example, agents frameworks without an internal rule engine could use stateful sessions to maintain agent state over time. Drools offers advanced rule based workflows, so-called KIS environment (knowledge is everything) and knowledge sessions across JBOSS products, which can be assigned to particular rule packages.

Appendix C: Overview of Data Computation Frameworks

Our motivation in researching currently existing data computation frameworks is to find the best way to support analysis of real-time, near real-time and static data in IoT. Agents or other processes will cause heavy system load when computing therefore, they should utilize a computational server. An agent or process might need to enrich the content of sensor data or create an abstraction of sensor data requiring computational resources.

In the following, we overview several frameworks, such as Apache Storm, Apache Spark, Apache Spark Streaming, Apache Flink, and have a look at their integration features.

Apache Storm

Website: http://storm.apache.org/

It supports real-time distributed data computation of streaming data, by using Complex Event Processing (CEP) server that is based on Apache Kafka. It also uses Predictive Model Markup Language (PMML)[9], which supports applications to describe and exchange models produced by data mining and machine learning algorithms. It supports common models such as logistic regression and feedforward neural networks.

Apache Spark

Website: http://spark.apache.org/

It is used for computation of static, non streaming processing, and to analyze data in database.

Apache Spark Streaming

Website: http://spark.apache.org/streaming/

Supporting real time distributed data computation of streaming data, in Java, Scala, Python, etc.

Apache Flink

Website: https://flink.apache.org/

This is a streaming dataflow engine for analysis of both static and streaming data. There exist several APIs for creating applications on top of the Apache Flink engine:

- DataSet API for static data embedded in Java, Scala, and Python,
- DataStream API for unbounded streams embedded in Java and Scala, and
- Table API with a SQL-like expression language embedded in Java and Scala.

[9] PMML https://en.wikipedia.org/wiki/Predictive_Model_Markup_Language

Apache Flink can be integrated with other open source systems for data input and output, as well as deployment. Figure C.3.1. shows Apache Flink overview architecture, and Figure C.3.2. illustrates the integration aspects of Apache Flink and other community tools.

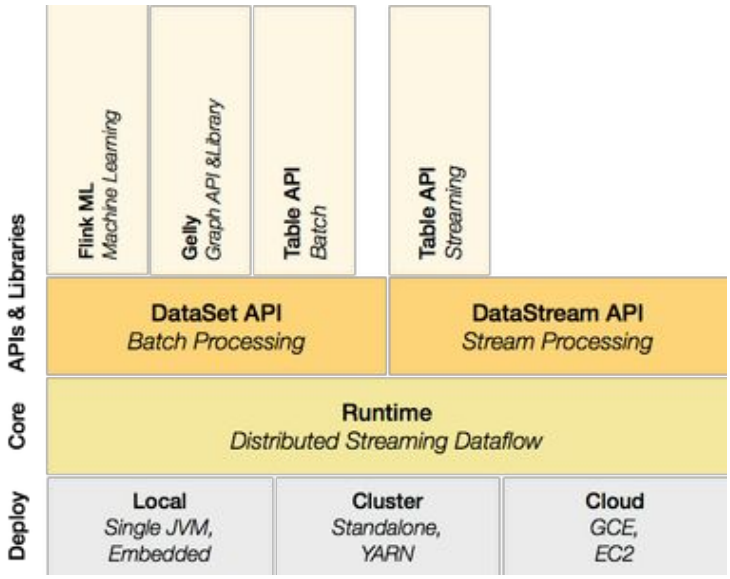

Figure C.3.1: Apache Flink Architecture Overview

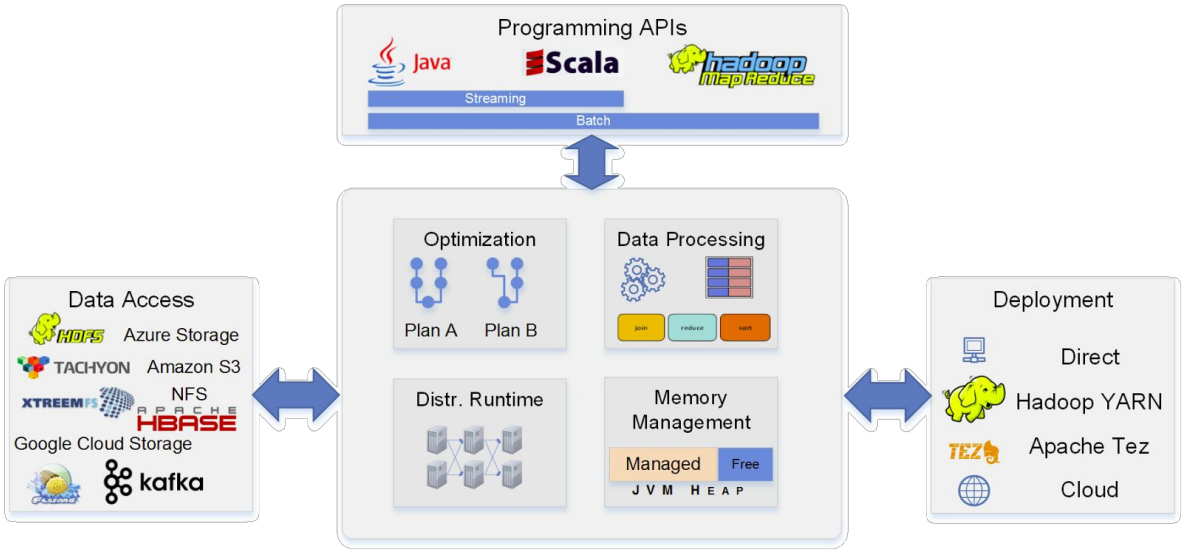

Figure C.3.2: Apache Flink API, Data Access & Deployment integration with community products

Apache Kafka and Apache Spark Integration Aspects

Some integration aspects of Apache Kafka and Apache Spark can be retrieved from the following links:

- Apache Kafka, Apache (Twitter) Storm & Elastic Search
    - https://www.youtube.com/watch?v=LpNbjXFPyZ0
- Apache Storm and Kafka
    - http://www.zdatainc.com/2014/07/real-time-streaming-apache-storm-apache-kafka/
    - http://de.slideshare.net/gschmutz/kafka-andstromeventprocessinginrealtime

Appendix D: Overview of Message Bus Technology

The following overview is necessary to answer the question on how to handle high throughput of IoT messages. We explore: MQTT, Apache Kafka, Apache Flume, zeroMQ, etc.

MQTT

Website: http://mqtt.org/

MQTT is Machine-to-Machine (M2M)/ "Internet of Things" connectivity protocol. It is OASIS standard (see http://mqtt.org/), "*designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios. It is also ideal for mobile applications because of its small size, low power usage, minimised data packets, and efficient distribution of information to one or many receivers."*

MQTT brokers are listed below:

- Mosquitto: http://mosquitto.org/ (Python broker and client library)
- Mosca: http://mcollina.github.io/mosca (Node.js MQTT broker)

Apache Kafka

Website:http://kafka.apache.org

Apache Kafka provides Java and multi language connectors/bridges, high performance message queue (publish subscribe) over PUBSUB protocol, etc. For more details, see:

- http://www.infoq.com/articles/apache-kafka

- https://engineering.linkedin.com/27/project-kafka-distributed-publish-subscribe-messaging-system-reaches-v06

Apache Flume

Website: http://flume.apache.org/

Flume is a distributed service for collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows:

- Apache Flume Sink and Source for Apache Kafka (consumer, producer) (see Figure D.4.1):
- A new channel that uses Kafka
- Hive Sink based on the new Hive Streaming support (HDFS)
- End-to-end authentication in Flume
- Simple regex search-and-replace interceptor

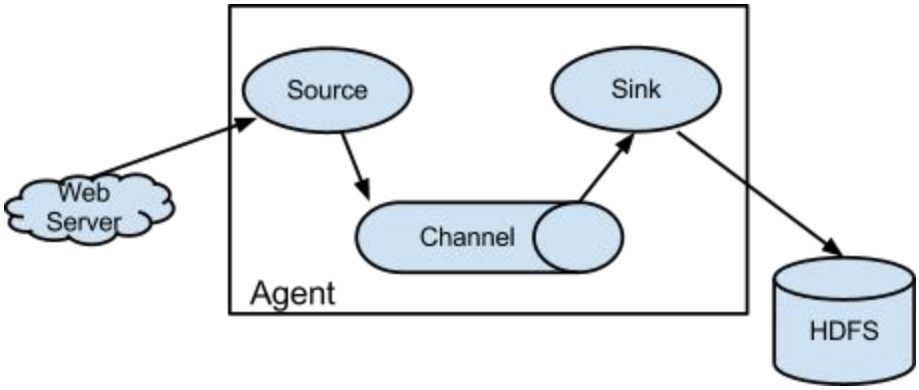

Figure D.4.1: Apache Flume Architecture Overview

zeroMQ / 0MQ

Website: http://zeromq.org/

zeroMQ is a high-speed asynchronous I/O engine, backed by a large and active open source community. It supports every modern language and platform, and carries messages across inproc, IPC, TCP, TIPC, multicast, using smart patterns like pub-sub, push-pull, and router-dealer.

Schedulers

Schedulers might already be included to support particular components e.g. Agent framework, IoT device Connectors such as Node-RED, or Thing-agent.js (Node.js), JIAC Agent framework and JBOSS Drools (Quartz based).

Appendix E: Overview of Semantic Technologies for IoT

Sensor Ontologies

Modeling of sensor configuration information requires sensor description and deployment aspects, which is based on standard ontologies such as SSN (Semantic Sensor Network) or SensorML ontology. Registering the sensor can require more than the base SSN ontology. Units and measures, tasks, etc. may have their own ontologies. Some examples of sensor configuration are provided by projects such as GSN, X-GSN, M3 (see Table E.1. for more details). Some additional goals of these projects are to support decision making, evaluate the capabilities of sensors (e.g. measurement range/limits, control topics possible), and provide sensor location(s).

Table E.1.: Overview of relevant projects on sensor configuration: GSN, X-GSN, M3

Project	**About**	**Details**
GSN	SSN and other ontologies and vocabularies	Provides configuration scheme for sensor configuration based on SSN and supporting ontologies and vocabularies (part of M3)
X-GSN	In OpenIoT project	Part of GSN for OpenIoT project (GitHub)
M3	Ontology and software extending SSN	Describe SSN, LOV4IOT (Linked Open Vocabularies for Internet of Things), domain ontologies but configuration files cannot be found

A set of sensor related ontologies describing sensors, sensor based observations, sensor deployment, control, maintenance, responsibility, etc. is given below:

- Sensor related ontologies: http://www.sensormeasurement.appspot.com/?p=ontologies
- Semantic Sensor Network (SSN) (W3C):
    - http://www.w3.org/2005/Incubator/ssn/ssnx/ssn
    - http://www.sciencedirect.com/science/article/pii/S1570826812000571
- M3 sensor ontology and software (including LOV4IOT:
    - http://sensormeasurement.appspot.com/
    - source: https://github.com/gyrard/M3Framework/tree/master/war/SPARQL

In addition, sensor systems can be associated to locations in the sensor description to support VSD sensor planning services. Modeling of location semantics can be done by using SSN ontology, as shown in the FP7 Butler project [BUTLER D3.2]. In BUTLER, in order to provide semantic and spatio-temporal reasoning, the following aspects are considered:

- Describing the spatial characteristics of different locations in sensor environment;
- Using the W3C SSN ontology to further describe the sensors and their position;
- Translating positions in coordinates into semantic locations (e.g. Bedroom);
- Semantically linking locations with activities (e.g. Sleeping in a Bedroom).

This background information can be modeled with ontologies as shown in the next Figure C.1.:

```
ex:Room             a                   owl:Class;
                    rdfs:subClassOf     geo:Feature

ex:LivingRoom       a                   ex:Room;
                    rdfs:label          "Living Room";
                    geo:hasGeometry     ex:GeoLivingRoom

ex:GeoLivingRoom    a                   sf:Polygon ;
                    geo:asWKT           "POLYGON ((290 600, 580 600, 580 700, 900
                                         700,     900     260,     290     260,     290
                                600))"^^sf:wktLiteral .

ex:activity         a                   owl:DatatypeProperty;
                    rdfs:domain         ex:Room;
                    rdfs:range          xsd:string

ex:LivingRoom       ex:activity         "Watch TV"
ex:LivingRoom       ex:activity         "Play Game"
ex:MasterBedroom    ex:activity         "Sleep"
```

Figure C.1.: Semantic modelling of context and location information

SensorML (Sensor Model Language)

SensorML (see http://www.sensorml.com/index.html) is a standard created under OGC's Sensor Web Enablement (SWE) activity (OGC 07-000), with the aim to enable interoperability between sensors and actuators as well as computational processes, so that they can be better understood by machines, and more efficiently used in complex workflows.

SensorML and SSN are used for modeling virtual sensors and their properties (e.g. sensor location), as shown in the following VSD (Virtual Sensor Description) examples:

- VSD based on GSN (Global Sensor Network) project:
    - **examples**: https://github.com/LSIR/gsn/blob/master/virtual-sensors/samples
- VSD based on X-GSN in OpenIoT project:

    - **examples**: https://github.com/OpenIotOrg/openiot/wiki/X-GSN-Use
- VSD based on SensorML 2.0:
    - provides general sensor information in support of data discovery;
    - supports the processing and analysis of the sensor measurements;
    - provides the geolocation of the measured data;
    - provides performance characteristics (e.g. accuracy, threshold, etc.);
    - archives fundamental properties and assumptions regarding sensor;
    - describes sensor tasks corresponding to control topics or threat levels (requires control topic or threat level terminology);
    - examples: http://www.sensorml.com/sensorML-2.0/examples/index.html

Here are some links to SensorML based ontologies:

- Ontology of general sensor terms: http://sensorml.com/orr/#http://sensorml.com/ont/swe/property
- Community Sensor Model (CSM) ontology: http://sensorml.com/orr/#http://sensorml.com/ont/csm/property

Appendix F: Overview of Security Technologies for IoT

IoT4Industry explores questions such as: *Which sensors can address particular threats? Where are these sensors deployed? What tasks can the sensor(s) perform at a particular location? What are the maintenance issues and who is responsible? What are data properties and measurement units?*

Threats modelling in a form of terminologies and ontologies mapp threats e.g. those found in log files, with topics and/or sensors. For example, a threat level (high, medium, low, normal) could be assigned to particular camera settings and retrieved depending on the DSS (Decision Support System) outcome. Some applications might report warnings or errors that need to be resolved by an edge agent.

Here is a list of related tools and projects in security for IoT:

- SIEM (Security Information & Event Management) systems collect, normalize, sort, aggregate, correlate and report all security-related events, independently of the product brand or license giving rise to such events. SIEM systems provide alerts, analysis and

archiving of messages and can be integrated with existing logging systems such as system logs, syslog, flat files, etc.

- Alien Vault (website: https://www.alienvault.com/). Alien Vault Open Threat Exchange (OTX™) is an open threat information sharing and analysis network.
- FP7 Secure project. This project creates rescue robots (agents) that abstract the sensor data and determine if there is a threat. The agents are using OwlAPI and OwlDB from SourceForge, as well as SSN and common.owl ontologies. No software found so far.

The following resources might support security/ threat terminologies:

- ENISA and its Incident Taxonomy: http://www.enisa.europa.eu/activities/cert/support/incident-management/browsable/incident-handling-process/incident-taxonomy/existing-taxonomies
- Shostack and Microsoft Threat modeling tool

ENISA Threat Terminology

The European Union Agency for Network and Information Security (ENISA) provides thematic based threat landscapes[10] that might be useful for creating an IoT security threat taxonomy. ENISA's threat terminology supports DSS and communication within system and with humans, as well as logging, e.g. report coded about threats OUTAGE or FIRE. Unlike the Shostack's threat modeling tool, no mitigation approaches are detailed. However, one can see the relevance regarding the high level threat categories and details organized according to the source cause of the threat: Physical attacks, Disasters, Outages, Failures and malfunctions, Unintentional damages (accidental), Nefarious activity/ Abuse, Damage/ Loss (IT assets), Eavesdropping/ Interceptions/ Hijacking, Legal.

Shostack and Microsoft Threat Modeling Tool

The Shostack and Microsoft Threat modeling tool[11] supports system and security design and advises threat mitigation approaches. According to Shostack, threat modeling is a means to discover security design flaws and support the overall software and systems design process. Threat modelling helps to understand possible threat mitigation approaches. For example, the Software Development Lifecycle (SDL) Threat Modeling Tool[12] from Microsoft is based on

[10] https://www.enisa.europa.eu/activities/risk-management/evolving-threat-environment/enisa-thematic-landscapes

[11] http://blogs.microsoft.com/cybertrust/2014/04/15/introducing-microsoft-threat-modeling-tool-2014/

[12] https://msdn.microsoft.com/en-us/magazine/dd347831.aspx

Shostack's book, which is about threat modeling Designing for Security. The SDL tool enables developers or system architects to achieve the following:

- Communicate about the security design of their systems,
- Analyze those designs for potential security issues using a proven methodology,
- Suggest and manage mitigations for security issues.

Shostack provides a categorization of threats, which is called STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege) (Table F.2.1.)

Table F.2.1: STRIDE Threats Categories[13]

Property	Threat	Definition	Example
Authentication	Spoofing	Impersonating something or someone else.	Pretending to be any of billg, microsoft.com or ntdll.dll
Integrity	Tampering	Modifying data or code	Modifying a DLL on disk or DVD, or a packet as it traverses the LAN.
Non-repudiation	Repudiation	Claiming to have not performed an action.	"I didn't send that email," "I didn't modify that file," "I certainly didn't visit that web site, dear!"
Confidentiality	Information Disclosure	Exposing information to someone not authorized to see it	Allowing someone to read the Windows source code; publishing a list of customers to a web site.
Availability	Denial of Service	Deny or degrade service to users	Crashing Windows or a web site, sending a packet and absorbing seconds of CPU time, or routing packets into a black hole.
Authorization	Elevation of Privilege	Gain capabilities without proper authorization	Allowing a remote internet user to run commands is the classic example, but going from a limited user to admin is also EoP.

SECURE (Semantics Empowered Rescue Environment)

SECURE rescue robots constitute an agent based system that generates abstractions for event detection in emergency scenarios. They are equipped with a variety of sensors gathering

[13] http://blogs.microsoft.com/cybertrust/2007/09/11/stride-chart/

sensory observations. The authors in [SECURE] discuss Semantic Web enabled system for collecting and processing sensor data within a rescue environment. The real-time system collects heterogeneous raw sensor data from rescue robots through a wireless sensor network. The raw sensor data is converted to RDF using SSN ontology and further processed to generate abstractions for event detection in emergency scenarios. An abstraction is a representation of an environment derived from sensor observation data. Generating an abstraction requires inferring explanations from an incomplete set of observations and updating these explanations on the basis of new information. Figure F.3.1. shows the hazards and the types of fire threats which are possible in each location (room), plus sensor status and whether a human is detected in the room or not.

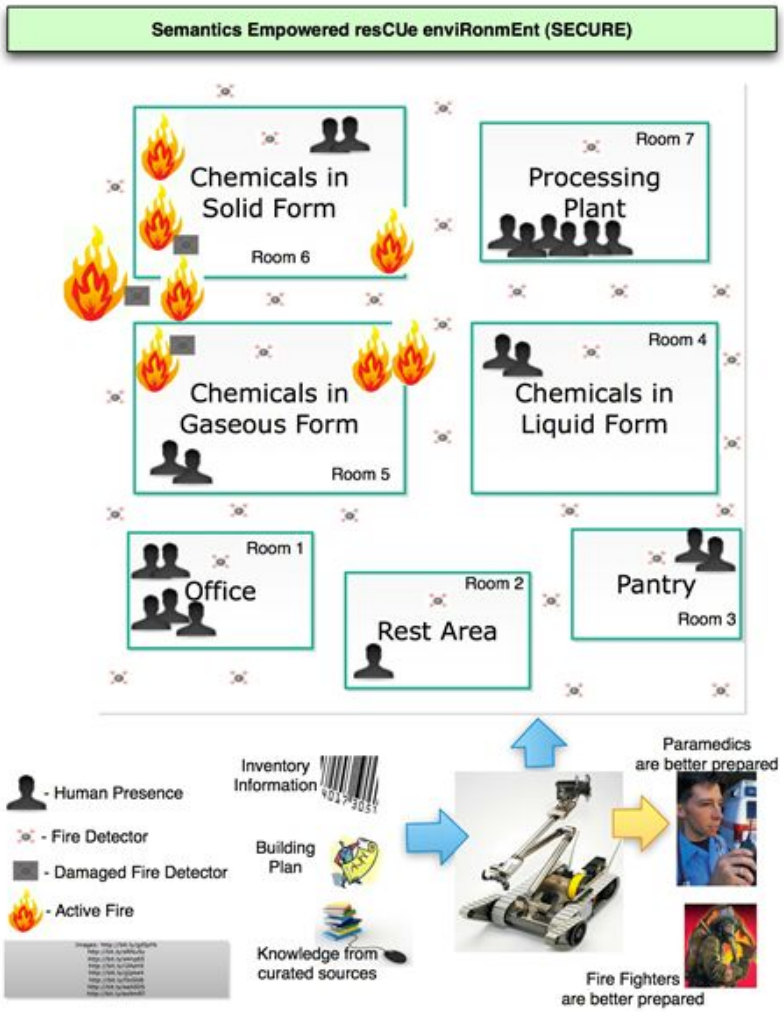

Figure F.3.1: SECURE: location (room) and threats, sensor status, human presence