WP2 Programmable Smart City
D2.3 Self-aware distributed city data platform
-First Release-

Grant Agreement N°723139
NICT management number: 18301

BIGCLOUT

Big data meeting Cloud and IoT
for empowering the citizen ClouT in smart cities

H2020-EUJ-2016 EU-Japan Joint Call

EU Editor: **CEA** JP Editor**: NII** Nature: Report

Dissemination: PU

Contractual delivery date: 2018-07-01

Submission Date: 2018-07-13

Co-funded by the EU H2020 GA. 723139 and NICT GA. 18301

ABSTRACT

This deliverable describes the first release of the self-aware distributed city data platform of the BigClouT project. The platform offers and manages capabilities for data collection and data redistribution, integrated with the self-awareness mechanism. The architectural design and technical implementation details of the platform are articulated in the document, with an emphasis on the updates made during the 2nd year of the project, therefore the functional blocks or components already described in the Deliverable 2.1 are not discussed in any detail.

Disclaimer

Revision history

Revision	Date	Description	Author (Organisation)
V0.1	2018-05-08	Initial ToC	(NII)
V0.2	2018-05-21	Updated Section 4	ENG
V0.3	2018-05-22	Updated Section 4	LANC
V0.4	2018-05-23	Various updates	LANC
V0.5	2018-05-23	Updated Section 1 & 2	NII
V0.6	2018-05-25	Updated Section 3	KEIO
V0.7	2018-05-28	Merged current sections and formatted the document	NII
V0.8	2018-05-29	Updated Abstract and Section 7	NII
V0.9	2018-06-01	Updated Section 6 for Dependability	NII
V1.0	2018-06-18	Removed the section for Service Composition and updated the figure references	NII
V1.1	2018-06-28	Addressed review comments	NII
V1.2	2018-07-10	Additional contributions and review	CEA
V1.3	2018-07-13	Final corrections and submission	CEA/NII

TABLE OF CONTENT

LIST OF FIGURES

1 INTRODUCTION

1.1 Scope of the Document

This deliverable is to provide the specification of the self-aware distributed city data platform in the BigClouT project, along with details about the design and implementation of its functional components. The platform encompasses a variety of software tools addressing **data collection**, **data distribution** and the **self-awareness** mechanisms for adaptability and dependability.

In the first year of the project, the Deliverable 2.1 [1] provided the analysis of requirements from the data collection, redistribution and self-awareness perspective, which lead us to build the BigClouT architecture's functional blocks related to the data collection and redistribution platform. The deliverable also provided some information about the developed/extended components from the project partners. The overall BigClouT architecture is also presented in the Deliverable 1.4 [3].

This deliverable focuses on the implementation of additional components in the second year of the project, as well as the updates of the components if they are significant. This deliverable puts thus an emphasis on the newly-added capabilities and components which are part of the following three functional blocks in the BigClouT architecture: Data Collection, Redistribution and Homogeneous Access; Edge Storage & Computing; and Security & Dependability.

1.2 Target Audience

The target audience of this deliverable are mainly the following two groups:

- BigClouT project members / developers. This document serves as a reference to enable members to understand existing data collection and redistribution functionalities, further implement new components, or modify existing ones.
- Smart City Ecosystem integrators who plan to develop a smart city ecosystem based on the BigClouT reference architecture by leveraging the self-aware distributed city data platform. The document outlines the relevant components of the platform and details the technical implementation, facilitating an in-depth understanding of the platform.

1.3 Structure of the Document

The document is divided in six sections. Section 2 gives an overview of the over-arching architecture of the self-aware distributed city data platform within the BigClouT reference architecture. It identifies the re-used functional blocks from the ClouT architecture, highlighting the updates made in the BigClouT project which will be the main focus of this deliverable. Following that, three dedicated sections (3-5) are provided for describing the three functional blocks of the platform that contain the main updates since ClouT, detailing the specific architecture, its specification and implementation respectively. In detail, Section 3 describes the Data Collection, Redistribution and Homogeneous Access functional block. Section 4 presents the Edge Storage & Computing functional block. Section 5 presents the City Resource Access block. Section 6 presents the Security & Dependability functional block focusing the Trust and Privacy component. Finally, Section 7 concludes the document.

2 SELF-AWARE DISTRIBUTED CITY DATA PLATFORM: OVERALL ARCHITECTURE

Referring to the related components described in deliverable D2.1 [1], Figure 1 highlights those components in the BigClouT architecture concerning data collection, redistribution and self-awareness. As concerns the self-aware distributed city data platform, the related components are grouped and briefly described according to the functionality they contribute to the platform. For each component, the main relevant updates since ClouT will be described.

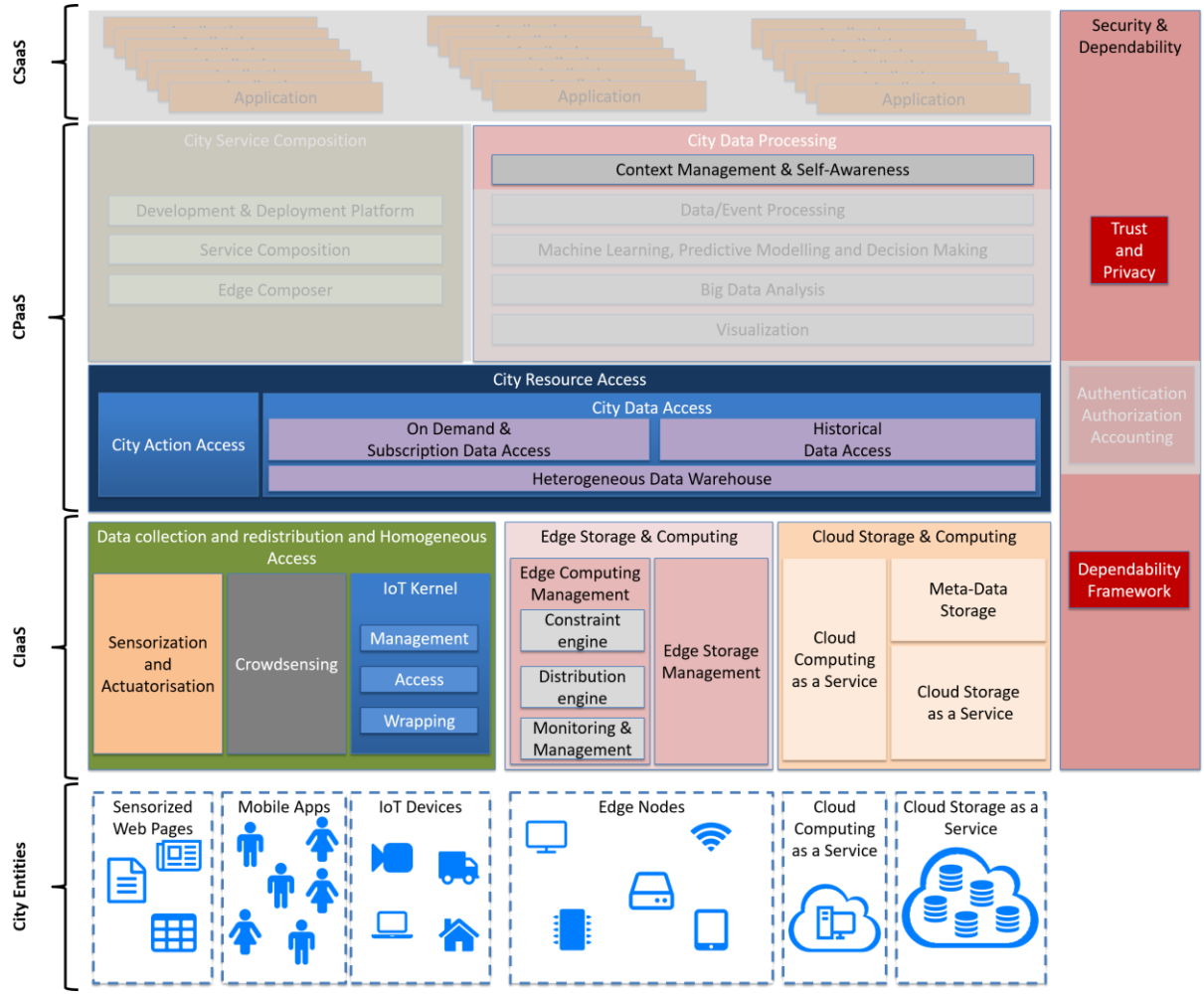

FIGURE 1 RELATED COMPONENTS FOR THE SELF-AWARE DISTRIBUTED CITY DATA PLATFORM

Related to Data Collection:

- **City Entities layer**. It accommodates the virtual or physical devices that interacts with the BigClouT platform, very often as sources of data feeding the platform. This layer is completely inherited from the ClouT architecture and therefore its description is **omitted** in this document.
- **Data Collection and Redistribution and Homogeneous Access functional block**. This is a **new** component responsible for collecting city data, managing interactions with city actuators and resources, providing event-based accesses and instantiating and maintaining the service model of the platform. It contains three sub-components, namely Sensorization and Actuatorisation, IoT Kernel, and **Crowdsensing**. While the former two

sub-components are inherited from the ClouT project and extended for the BigClouT project, Crowdsensing is a newly developed component during the BigClouT project.

Related to Data Redistribution:

- **Data Collection and Redistribution and Homogeneous Access** (cf. see above).
- **Cloud Storage & Computing functional block**. It provides functionalities to access cloud storage and computing. It is inherited from the "Storage & Computing" module of ClouT. Therefore, this document will **omit** further description of this functional block.
- **Edge Storage & Computing functional block**. It provides functionalities regarding edge nodes, responsible for managing edge storage and processing. This is a **new** component introduced in the BigClouT project and hence will be described in this deliverable.
- **City Resource Access functional block**. It offers capabilities to access real-time and historical data from various city resources, in addition to capabilities to perform actions on resources. The functional block is inherited from the ClouT architecture and a new sub-component "Heterogeneous Data Warehouse" is **added**, which is already presented in the Deliverable 2.1 [1]

Related to Self-Awareness:

- **Context Management & Self-Awareness component from the City Data Processing functional block**. It is responsible for storing and delivering high-level context information derived from the processed data and events. It manages two types of information: *User Context* information (such as profiles, preferences, localization, etc.) and *City Context information* (such as temperature, pollution, etc.). This component is inherited from the ClouT project and no significant updates will be reported in this deliverable.
- **Edge Storage & Computing functional block** (cf. see above).
- **Cloud Storage & Computing functional block** (cf. see above).
- **Dependability Framework component**. Its role is to maintain the state of the platform and generate rules to enable automatic configuration for relevant components of the platform in order to keep the expected level of dependability (e.g. reliability, availability, maintainability and safety) of the system. Inherited from the ClouT project, this framework has been **extended** in BigClouT by introducing the self-awareness capability.

The Trust and Privacy component within the Security & Dependability functional block will also be described here since it contributes to the overall quality of the generated data in terms of data structures holding access rights to existing resources and their level of trust regarding who is providing the data. The interactions among those relevant functional blocks and components have been addressed in D2.1 [1]. Therefore, this document will focus on the design and implementation of each new or updated functional block and component mentioned above.

3 DATA COLLECTION, REDISTRIBUTION AND HOMOGENEOUS ACCESS

3.1 Architecture

The original ClouT architecture provided basic functionalities to collect large volumes of data from various resources including websites, SNS, citizens and IoT devices. In addition to collecting such data, the architecture provides homogenous access methods to various types of applications. In the BigClouT architecture, we also followed basic functionalities of ClouT architecture such as Sensorization or IoT Kernel. In addition, we explore another important challenge - how we involve citizens-in-the-loop of city sensing. In the ClouT project, we mainly focused on providing tools to users - there were no special functionality to encourage users to collect city information. In the BigClouT architecture, we consider that it is important that the architecture itself has to provide engagement-enhancing functionality. To enhance engagement of citizen, we have to consider many aspects in terms of technical point of view such as usability and psychological point of view such as enhancing internal motivation. We addressed these problems to BigClouT architecture, and proposed new functionalities in data collection, redistribution and homogenous access components, which is highlighted in Figure 2.

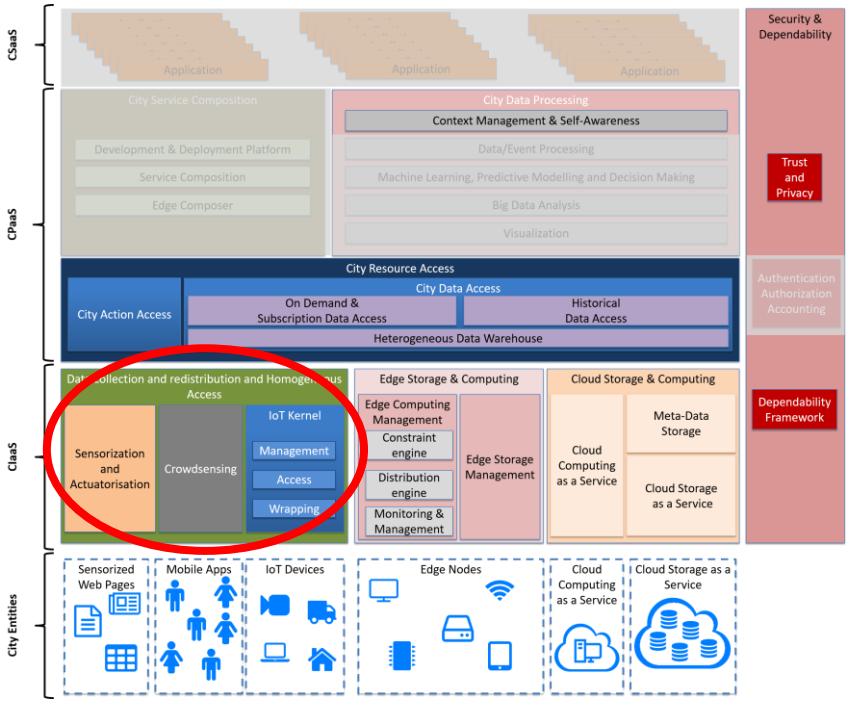

FIGURE 2 SECTION OF THE ARCHITECTURE FOR DATA COLLECTION, REDISTRIBUTION AND HOMOGENEOUS ACCESS

In this section we particularly focus on the Crowdsensing part, which is the main update at this layer related to the data collection and redistribution module.

3.2 Specification

With the population moving into urban areas, city governments are required to respond flexibly to various demands of people being aware of the current situation of the city. Crowdsensing (or participatory sensing) is recognized as one of the major sensing technologies to understand the

situation, since it can detect both objective and subjective information leveraging human perception.

There are many citizen-centred crowdsensing applications with which citizens cooperatively gather and share data (e.g., the applications for monitoring the environment). Crowdsensing that users report data with their actual sensing positions entails the following two issues. Firstly, we need to motivate people to participate. People can be gifted something when they report their findings, and/or be guided to a sensing target when they are close to there. Secondly, we need to protect people from their location privacy leakage. For users' motivation, a crowdsensing platform should allow users' contribution without anonymizing their identity, though it exposes users' privacy. For privacy protection, it is technically possible to allow users to contribute anonymously, however, this may decrease their motivation. It is thus necessary to solve these existing issues simultaneously in a well-balanced manner since they influence each other.

To cope with this problem, we have built 2 crowdsensing solutions: Minarepo and Lokemon. Minarepo has been used in the project pilots and explained in the WP4 deliverables. This deliverable will give more detail about the novel developments on the Lokemon.

Lokemon is based on the notion of user label. Instead of making a report with a username, or contributing anonymously, Lokemon lets users to contribute as a monster associated with a sensing target, i.e., Point of Interest (PoI). This brings the following four features to the crowdsensing platform.

Firstly, users can find monsters on a map and get their information. By doing so, we attract users to a PoI and induce them to visit there.

Secondly, users can ask a question to a monster instead of asking someone with his/her username or anonymous name. By using known identity to interact with people, we activate and facilitate communication between users.

Thirdly, by using a monster as an alias when contributing, we entertain people and motivate them to post information. It also hides users' real identity. It can thus make it easier for users to post information while reducing the loss of location privacy.

Finally, multiple users can share a monster when they are simultaneously present within a PoI. A shared monster is a common identity of the users, which narrows a talking target to one. This shared identity simplifies interaction with users. All in all, Lokemon monsters are the intermediation mechanism among cooperative people.

In Lokemon, a monster is associated with a Point of Interest (PoI). A range of a PoI corresponds to the area where desired information can exist. For example, as shown in Figure 3, if you want to know the congestion level at a bus stop, you should decide a range within a distance that users can recognize the bus stop. If you choose a shopping mall as a PoI and want to know the atmosphere, the range of the PoI should be wider than that of the bus stop. In such ways, an

interest range could be different depending on a PoI. Therefore, a range of a PoI should be decided optionally.

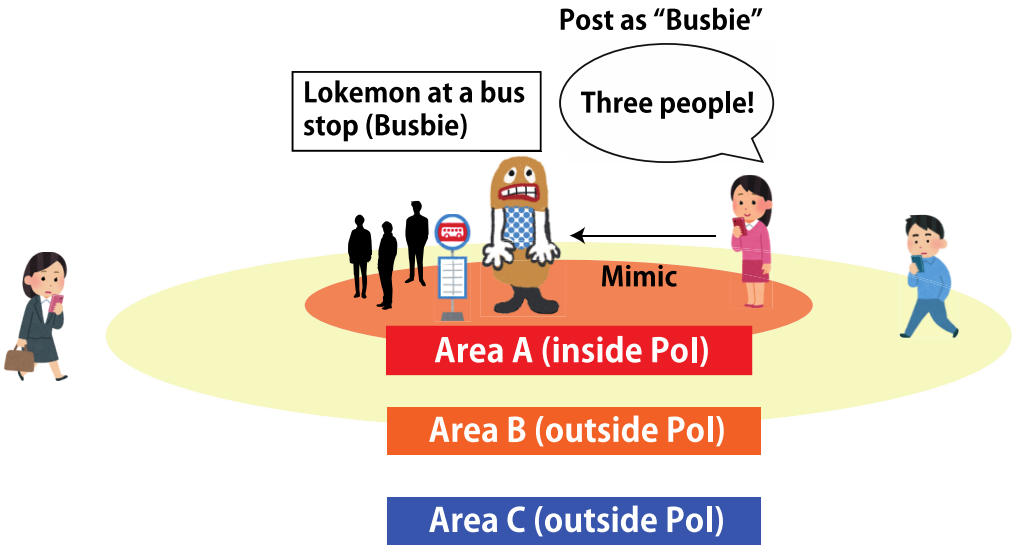

FIGURE 3 RELATIONSHIP BETWEEN A POINT OF INTEREST AND AN AREA

Besides, even though objects of interest are different, they can exist in the same place. For instance, the bus stop could be located at the shopping mall. Therefore, a centre position of a PoI should be set independently. Depending on a variable range and centre position of a PoI, the positional relationship can be categorized into three types as shown in Figure 3. In the figure, an area of a PoI is illustrated as a circle.

As below, we define the three areas around a PoI (see Figure 3). By changing the user's role according to the distance between a PoI and a user, we motivate people while reducing the risk of users' location privacy leakage.

- Area A: Inside a target PoI where users can contribute to crowdsensing by playing the role of a monster, e.g. using a monster's name as users' alias to post information. This area is referred as "mimickable area".
- Area B: Outside Area A where any user can get detailed information about nearby monsters. However, users in this area cannot play the role of a monster. This area is referred as "non-mimickable area".
- Area C: Outside Area B where users can get detailed information about monsters after they visit the PoI and add the monster to their collection. This area is also referred as "non-mimickable area".

If the respective centre positions are different and the ranges of a PoI do not overlap each other, then the positional relationship should be like Figure 4-A. When the ranges of a PoI overlap each other, the positional relationship can be illustrated in Figure 4-B. When the respective centre positions are near/the same and/or a range of a PoI is enough larger than another PoI, the large

PoI can include the small PoI (Figure 4-C). If information about a PoI is no longer needed, the monster can be deleted.

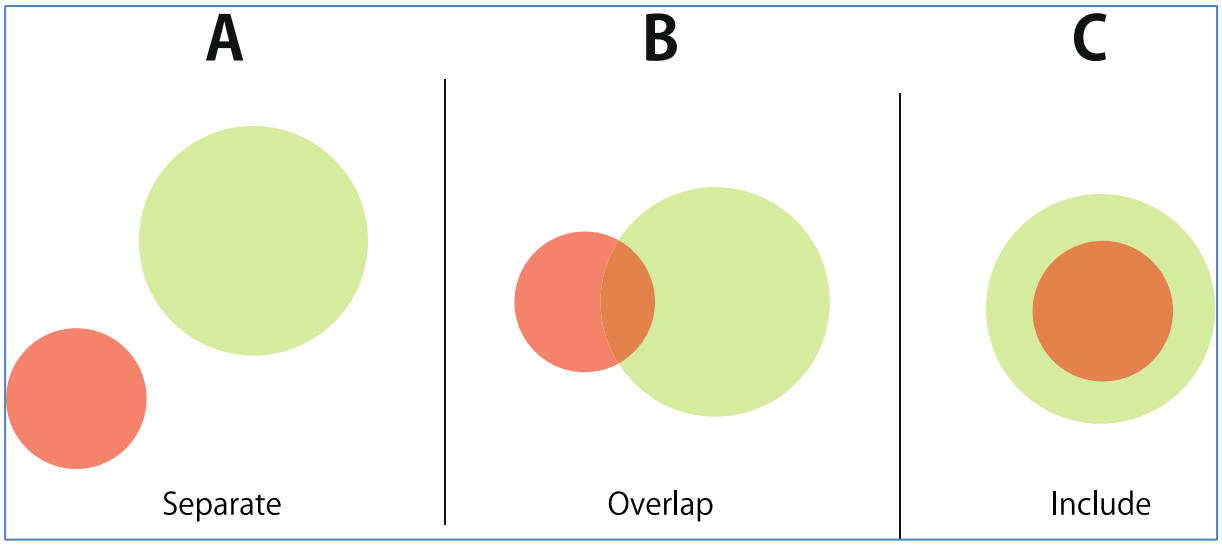

FIGURE 4 POSITIONAL RELATIONSHIP BETWEEN POINT OF INTERESTS

Lokemon enables users to perform the following actions, and the actions suggest solving the aforementioned issues in the following four steps:

I. Find monsters on a map

Users can get all monsters' geographical information on a map. Based on the map, they can find nearby monsters and collect them. As for nearby and collected monsters, they can get detailed information about monsters such as their shape and design. Through finding monsters, we let users have interest in PoIs. This action will eventually induce them to visit there.

II. Ask monsters a question

Users can ask a question regarding a PoI to a monster. In a typical crowdsensing model, users ask a question to someone with his/her username or anonymous name. In such a model, we cannot know the identity of users. On the other hand, in the Lokemon model, users can ask to a monster whose identity is known to the Lokemon system users. By putting well-known identity between users, we activate and facilitate communication between strangers.

III. Use a monster as an alias to post information

When a user is within a PoI, he or she can achieve crowdsensing as a monster which is virtually located at a PoI. Figure 5 shows a comparison between the typical crowdsensing model and the

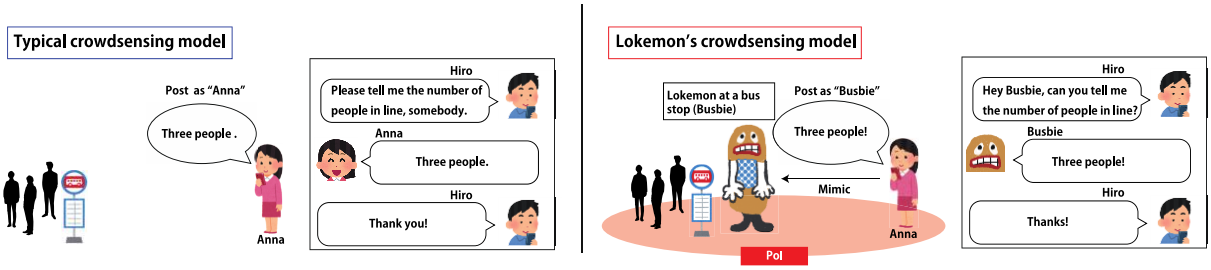

FIGURE 5 THE COMPARISON OF THE CURRENT PARTICIPATORY SENSING MODEL AND THE LOKEMON SENSING MODEL

Lokemon model. In a typical crowdsensing model, users send sensing data with their user names. In the Lokemon model, by contrast, users send sensor data in the name of the monster in the PoI. By using a monster as a sender label, users can post information without exposing their identities. This reduces the risk of privacy leakage, since other users cannot know who is actually reporting the data.

IV. Share a monster with other users

Multiple users who are simultaneously present within the same PoI can share the monster until they leave there. This mechanism realizes a shared identity among users and makes a new form of online communication. In a typical crowdsensing model, people interact with one another directly (many-to-many communication). On the other hand, in Lokemon, they interact with a monster (many-to-one communication). In this way, by narrowing the talking target to one, we make social interaction simple and easy.

3.3 Implementation

The Lokemon system is designed as a simple client-server architecture. At the same time, BigClouT's city resource access module can access data obtained by Lokemon via the SOXFire module. Figure 6 shows the overall system architecture of Lokemon. Figure 7 shows the screenshots of Lokemon application.

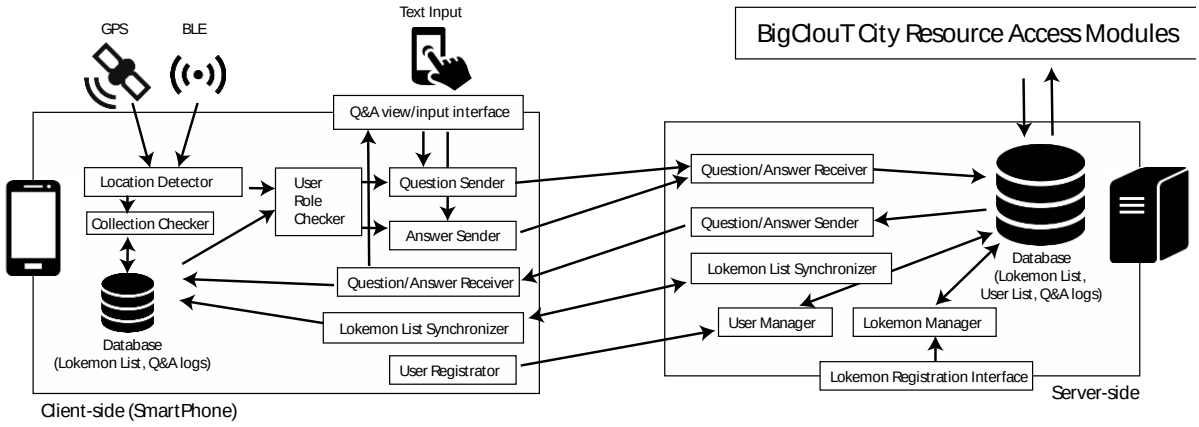

FIGURE 6 SYSTEM ARCHITECTURE OF LOKEMON

On the client-side, users can register their user information through a User Registration module. The minimum information to be offered at the time of the user registration is a user name and an icon image; any information that is personal in nature is not required at all.

On the map screen (Figure 7-A), users can see the monsters' location with their current positions. By tapping a monster on the map, users can see the detail of the monster which contains the monster's profile (Figure 7-B). The search screen shows users if there is a Lokemon character they can adopt at users' current position (Figure 7-C-1 and C-2). The reporting screen has a message UI, like chat, to help people understand that they can talk to a monster which is mimicked by someone (Figure 7-D). For real-time message exchange between the client and the server, we used the WebSocket protocol.

FIGURE 7 SCREENSHOT OF LOKEMON APPLICATION

On the server-side, several data are stored in a database, such as those about registered users, monsters, and question & answer logs. Monsters are registered through the Lokemon Registration Interface by entering the monster's name, image, location with latitude and longitude, and the maximum distance at which it can be mimicked. This distance is referred as “mimickable distance”. To detect a monster's area, we used either BLE's beacon or GPS. In the case of using GPS location information as the mimickable distance, the distance is defined in meters. On the contrary, in the case of using a BLE beacon for identifying users' location, mimickable distance is not defined but a flag of "use BLE" is set true and the corresponding BLE's ID is defined. To reduce power consumption of smartphones, we ensured that the GPS was turned on only when the application went foreground. We notified users when they got a new monster. The monster list is synchronized both on the client and server-side when a new monster is registered on the server-side.

To specify the role of the user, the User Role Checker checks whether the user is in Lokemon-mode or not. The user specifies one of the monsters to start questioning or answering. When the user is eligible to mimic a specified monster, the user can answer of the questions from other users through the Answer Sender module. Alternatively, when the user is not eligible to mimic the monster, the user can send a question to the monster by using one's own name. The sent question or answer is stored in the database on the server-side, and the information is subsequently notified to other users. All data was stored in PostgreSQL DB. We built Ubuntu server and used the Django web framework to provide API for accessing/saving data from smartphones.

4 EDGE STORAGE & COMPUTING

4.1 Architecture

In ClouT most of the data produced by the data sources (sensors, virtual sensors, social networks etc.) were processed and stored on the Cloud. The scalability and the elasticity of the Cloud were considered the best approach to store the large volumes of produced data.

BigClouT, besides the requirements concerning the large volumes of data, takes into account more use cases in which also **data latency** and **network traffic** play important roles. It is important to remark that data latency and network traffic do not have to be low in all the cases: instead, they should not introduce unnecessary performance issues and should be appropriate to the goal of the specific application. For instance, a real-time application *should* have low latency while non-real-time applications *could* have higher latency; if a video must be stored as historical data on the Cloud, a burst of network traffic cannot be avoided, but if the result of a simple processing activity on it is needed locally, it is unnecessary to transfer data back and forth on the network.

It is easy to understand that these aspects are not always compliant with the use of a pure Cloud paradigm: a further example can be useful to better clarify the context. A traffic light automatically configures the duration of red or green lights depending on traffic data, collected in real time by an integrated sensor. Specifically, traffic data is processed in the following way:

1. data is collected by the sensor
2. data is stored
3. when a *relevant* amount of data is collected, a service processes it to define the durations of red and green lights.

The meaning of the word *relevant* depends on a trade-off between the needed speed of the decisions (depending on the amount of data to be collected) and its reliability. A pure Cloud paradigm, in this case, would require immediate sending of the collected data to the Cloud in order to be processed and stored: then the results would be sent back to the semaphore to determine its behaviour. This flow introduces two issues:

- **data latency**, since an extra delay between the data collection and the result processing is introduced by the network between the semaphore and the Data Centre.
- **unnecessary network traffic**, since data are collected on a certain place, processed and stored on another place and the final result is sent back to the first place.

In this example, the second point can become a significant issue, especially if a certain number of *smart traffic lights* are deployed in a smart city. A large volume of unnecessary network traffic can also affect data latency having impact on this application (even if in this case real-time results cannot be so important) or in other applications in which real-time results are critical. There is

also an aspect related to security: the less data is moved the less security issues should be managed.

For such a use case, a small local P*rocessing and Storage System* would increase dramatically the overall efficiency. However, if we consider also the smart city context, to have an isolated sensor system is not acceptable, since data should be stored as historical data and could also be required for different applications.

The solution proposed and included in BigClouT platform is based on the *Edge paradigm*: i.e. autonomous and interconnected local nodes, composing a system that locally processes and stores data and provides the capability to process it elsewhere (for example, when more processing capability is required) and access it regardless of its exact location.

The term Edge is sometimes used synonymously with *Fog*: in BigClouT both the terms are used with the same meaning and the *official* name used on the architecture is **Edge**. However, to be more precise, the term *Edge* also refers to the *devices* or the *nodes* at the *edge* of the network, closer to the clients. In the example above an *Edge Node* close to the traffic light could provide local *processing and storage functionalities* and cooperate with an Edge (or Fog) System. *Edge Nodes* may be deployed on specific machines or leverage part of the resources of the IoT devices: for example, if the traffic lights were a *smart device*, connected on the network and with processing or storage capabilities, it could be an Edge Node by itself. Edge Nodes are important to guarantee low latency and to reduce the network traffic to and from the centre of the network.

Figure 8 shows a section of BigClouT architecture where the modules providing Edge (Fog) and Cloud functionalities are put in evidence.

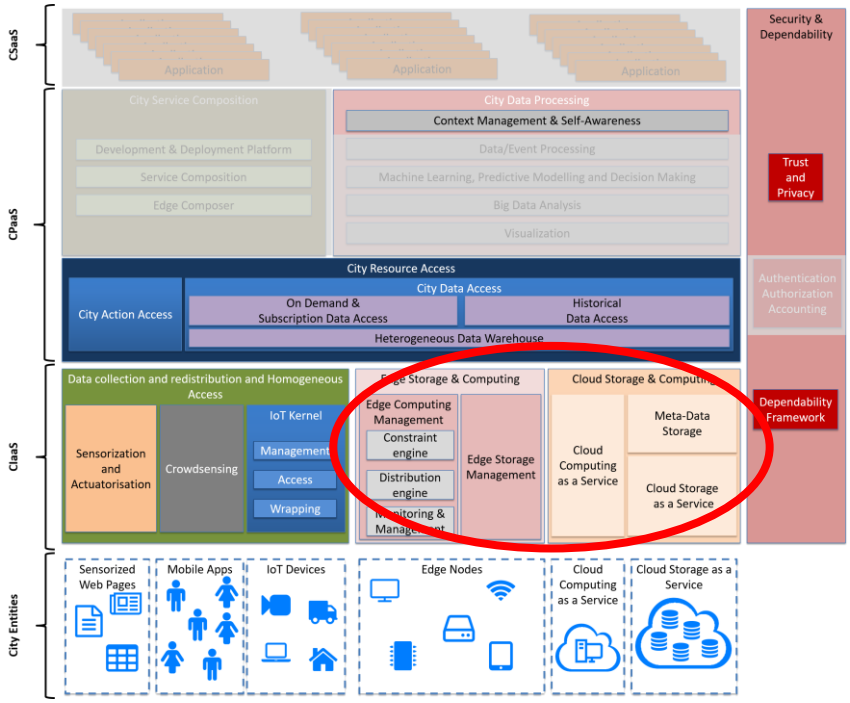

FIGURE 8 SECTION OF THE ARCHITECTURE INCLUDING CLOUD AND EDGE STORAGE

4.2 Specification

4.2.1 Edge Computing Management

The Edge Computing Management subsystem manages the activities related to distributed data processing by applying the Edge (or Fog) Computing paradigm. It is a purely logical component: its actual implementation includes centralized and distributed elements at different architectural layers.

Processing elements will be distributed to remote processing nodes from the central BigClouT platform to meet requirements such as latency and processing power, which operation should be performed at device level, which ones at one of the intermediate levels and which ones at cloud level.

As shown in Figure 9 below, the edge computing management subsystem is composed of three sub components:

- *Constraint engine*: takes application constraints and maps to edge processing capabilities. Constraints are specified in the edge composer by application developers and concern the context of a particular node (e.g. this processing element must run on a processing node with 2MB of memory, a 4 core processor and be located in the Henleaze area of Bristol).
- *Distribution engine*: partitions code and distributes to remote processing nodes. This component uses the initial constraints and runs a constraint satisfaction algorithm to determine an optimal distribution of processing elements. The engine then partitions the

application flows into separate processing units and physically replicates the code to the edge processors to meet the constraints.

- *Monitoring & Management*: monitors status of remote nodes, accepts new nodes to the cluster and removes nodes. A final component of the edge computation system monitors the overall system and ensure that changes to the network and processing/edge nodes is reported to the distribution engine. These updates are used to rerun the constraint satisfaction algorithm as needed when the monitoring component detects changes.

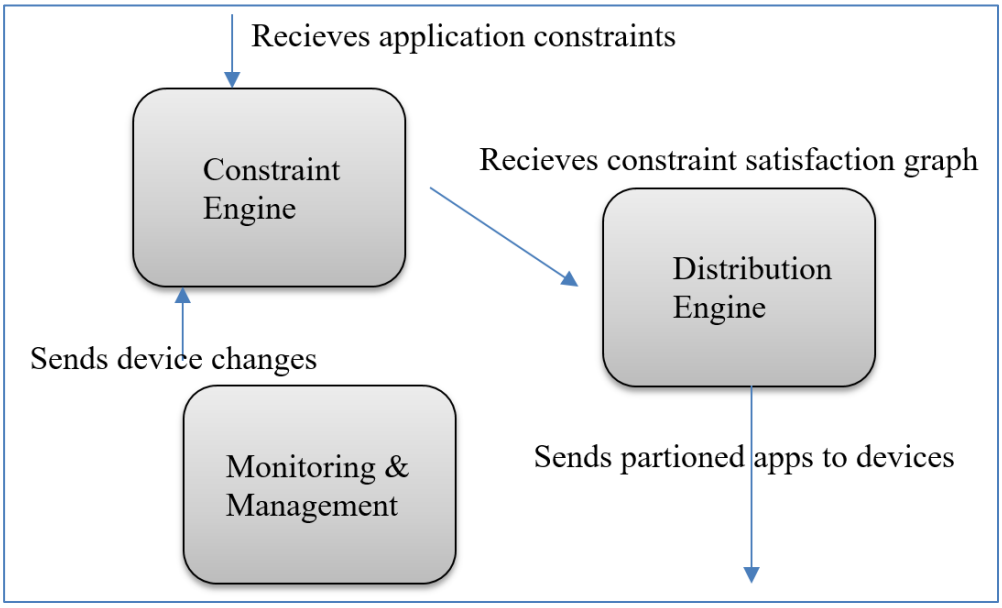

FIGURE 9 COMPONENTS AND DATA FLOWS OF THE EDGE COMPUTING SUB-SYSTEM

4.2.2 Edge Storage Management

BigClouT **Edge Storage System** is composed by a set of autonomous nodes. The Nodes are deployed based on the *master/slave paradigm*: in particular, the system includes a single Master Node and an arbitrary number of Slave *Edge* Nodes.

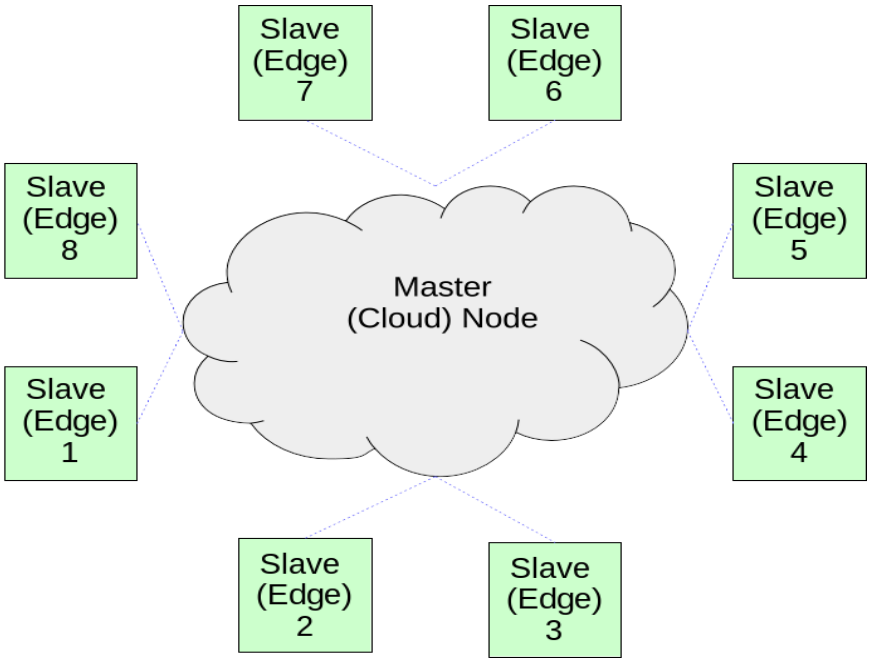

FIGURE 10 EDGE STORAGE SYSTEM

Figure 10 provides a schema of the deployment of BigClouT Edge Storage System. The Master Node is a Cloud Storage and the Slaves are smaller Nodes with limited storage capabilities. In general, the Master can be any Node of the System. It is not mandatory that the Cloud Storage Node is the Master.

Each Node has a certain amount of storage capability, depending on its architecture and its configuration: however, a small node, with low capabilities, can be deployed everywhere while the Cloud Storage can exist only on the Data Centre. The Master is the only Node that can identify the exact location of each piece of data in the System: for this reason, if a GET DATA request is sent to the Master and requested data is not present locally, the Master can redirect the request to the Slave containing requested data. If a Slave receives a GET DATA request for data not present locally, it redirects the request to the Master that can directly answer or, in turn, redirect to another slave.

The BigClouT Edge Storage System is implemented as an extension to the ClouT Cloud Storage. All the exposed functionalities are compliant with CDMI 1.1.1 standard.

The data structure is organized in containers collecting data produced by a certain *entity*. The mapping on BigClouT domain model is the same defined in ClouT (please refer to ClouT deliverable 4.2 [4] for the details). Each piece of data is formatted in JSon and the exposed CRUD operations follow the REST paradigm. **Multipart messages** (for data objects, i.e. images, videos etc.) and **query queues** are supported.

A new feature, with respect to ClouT, is the support for **nested containers**: this aspect provides more flexibility to better map containers on the domain model and on the Edge Nodes.

All the mentioned functionalities are supported by the Cloud Node and the Slave Nodes. The interfaces are the same, so a client able to talk with ClouT Cloud Storage is also able to talk with each Node of BigClouT Edge Storage System.

Each Slave Node should be configured at deployment time with the URL of the Master. This is the only additional configuration step with respect to ClouT. When a new container is created on a Slave Node, a notification message is sent to the Master which will register the container and associate it to the sender. A similar notification message is sent if the container is deleted.

It is possible to distinguish three main use cases which define three different data flows through the storage:

- **Data stored on a certain Node is requested by using the endpoint of the same Node**. In this case the Node (Master or Slave) acts as an isolated node, since data are written and read through its endpoint: if the Node is a Slave Node, it sends a notification to the Master when the container of the data is created or deleted.
- **Data stored on a certain Slave Node is requested by using the endpoint of the Master**. When the Master Node receives a read request, first of all it checks if data is present on its local (Cloud) storage. If data is not present, it checks whether the Container is associated to one of the Slave nodes: if the Slave Node is found, the read request is forwarder to that Node which will answer autonomously.
- **Data stored on a certain Slave Node is requested by using the endpoint of another Node**. If a read request arrives to a Slave Node and data is not present locally, the request is immediately forwarded to the Master Node: then the process goes ahead as in one of the previous points.

4.3 Implementation

4.3.1 Edge Computing

Figure 11 depicts the technical architecture of the edge computation system. The system is implemented as a distributed run-time co-ordinated by a set of message brokers. A master node (DNR operator) acts as an initial seed point for the architecture and carries out initiation of the distributed network. As new nodes join the network, they instantiate a broker instance which joins the distributed group, announces its capabilities and listens for requests from other brokers.

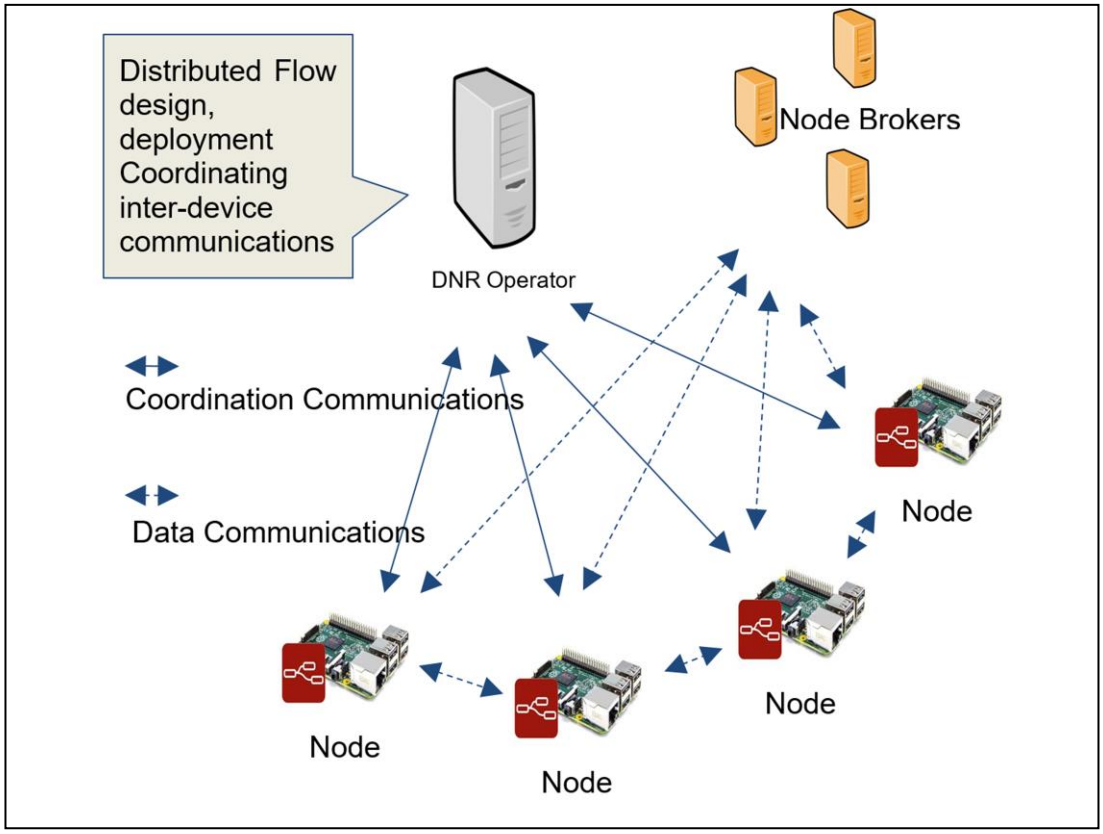

FIGURE 11 CO-ORDINATED BROKERS MANAGE DISTRIBUTED EDGE PROCESSING

The brokers run as a distributed co-ordination layer, communicating amongst themselves and providing status update as context changes at their associated edge processing node as well as forwarding messages to/from partitioned flows.

In Figure 12, the details of the partitioning of an application flow are shown. In this case a simple 3 processing element flow (inject, function, debug) is mapped to three physical edge nodes. The distribution component then replaces inter-module communication links with inter-process or inter-device communication links - shown as wire-in and wire-out elements - which allow the 3 elements to be physically located on different edge processing nodes. The edge computing

subsystem then distributes the processing element to the three devices, using their respective brokers, and arranges for the brokers to act as forwarding points for wire-in and wire-out.

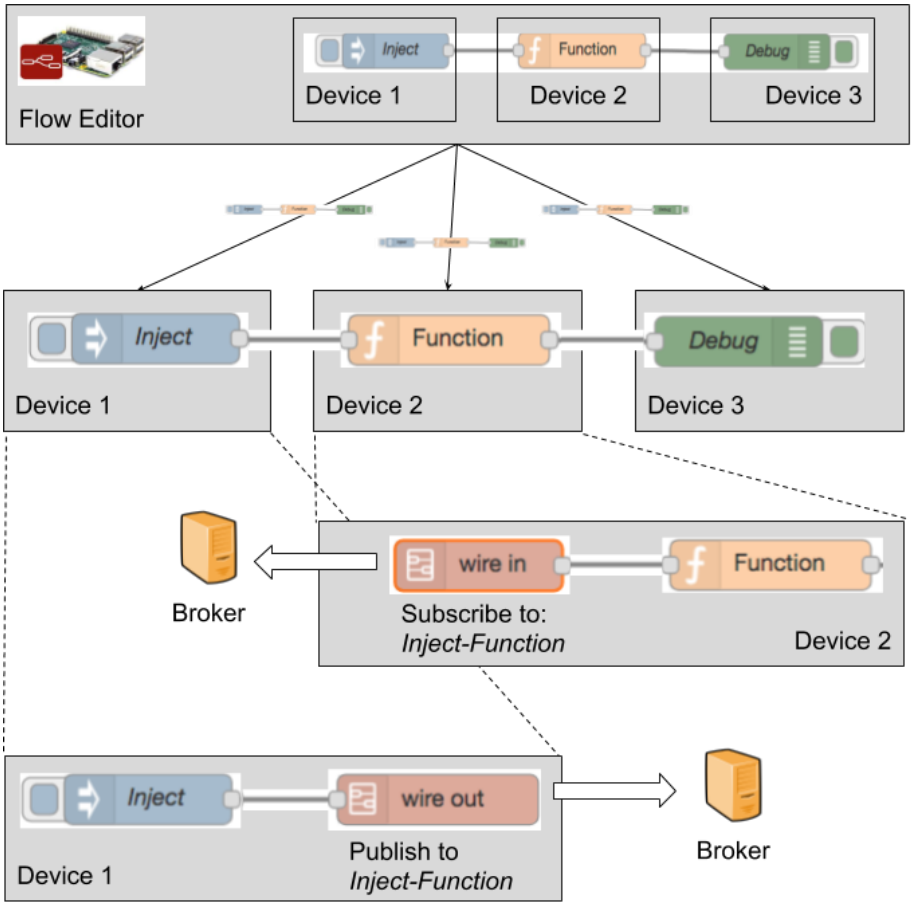

FIGURE 12 IMPLEMENTATION OF EDGE COMPUTATION ACROSS DEVICES

4.3.2 Edge Storage

Figure 13 depicts the technological architecture of BigClouT Edge Storage System. As mentioned above, all the Nodes expose CDMI interfaces. Moreover, the Master exposes a non-standard interface that receives the notifications of creation and disposal of new containers.

The Cloud Storage is based on the same technologies used for ClouT, so it integrates **OpenStack Swift** and **Hypertable NoSQL** database.

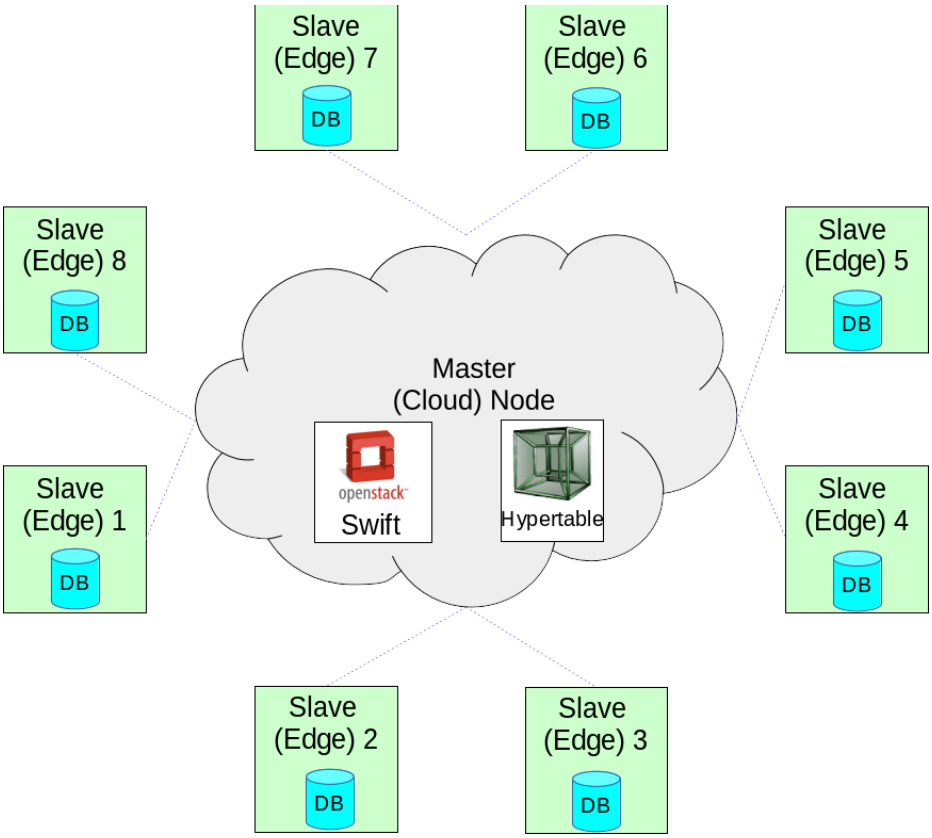

FIGURE 13 EDGE STORAGE, TECHNOLOGICAL ARCHITECTURE

Each Slave Node can be deployed on a single machine: the minimum configuration includes, along with CDMI interfaces, a local database. Since CDMI interfaces are mandatory and a web service exposing them should be present on all the Nodes, it is very difficult (even if potentially not impossible) to exploit the storage capabilities of the smart sensors: Nodes should be deployed on dedicated machines. However, there are not constraints on the usable databases, which can be SQL or NoSQL.

At time of writing this deliverable, the mentioned features are implemented and deployed on a test environment with a limited Cloud Storage and a couple of virtual machines on which instances Postgres are running. In Year 3 advanced aspects will be designed in details and implemented: among them the most important are *security* and *Edge Node limits*. The latter aspects concern the possibility that a Slave Node exceeds its storage capability: in this case part or all the stored data will be sent to the Master.

5 CITY RESOURCE ACCESS

5.1 Architecture

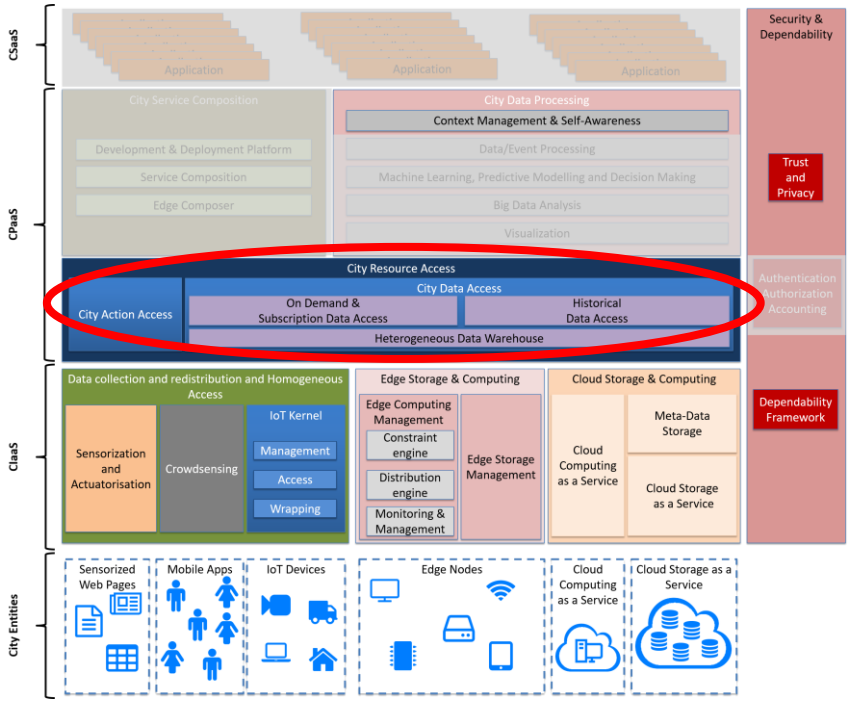

FIGURE 14 SECTION OF THE ARCHITECTURE INCLUDING CITY RESOURCE ACCESS

The city resource access block is in charge of providing a homogeneous access to underlying heterogeneous city data and action resources. Those resources are typically data from (and actions to) sensor/actuator devices, sensorized web pages, mobile apps, social networks, edge nodes, etc. Behind generic APIs and unified data models, the city resource access layer ensures the secured and homogeneous access to city resources.

More details on the architecture can be found on the Deliverable 1.4 [3] .

5.2 Specification

A first high-level specification of the resource access layer has been given in the Deliverable 2.1[1] . This section introduces more specific details about the APIs and data models of the city resource access layer.

The City resource access layer allows accessing resources exposed by city services. The services are characterized by a service identifier, representing a concrete physical device or a logical entity not directly bound to any device. Each service exposes resources and could use resources provided by other services. Figure 15 Generic resource/service model. depicts the Service and Resource model.

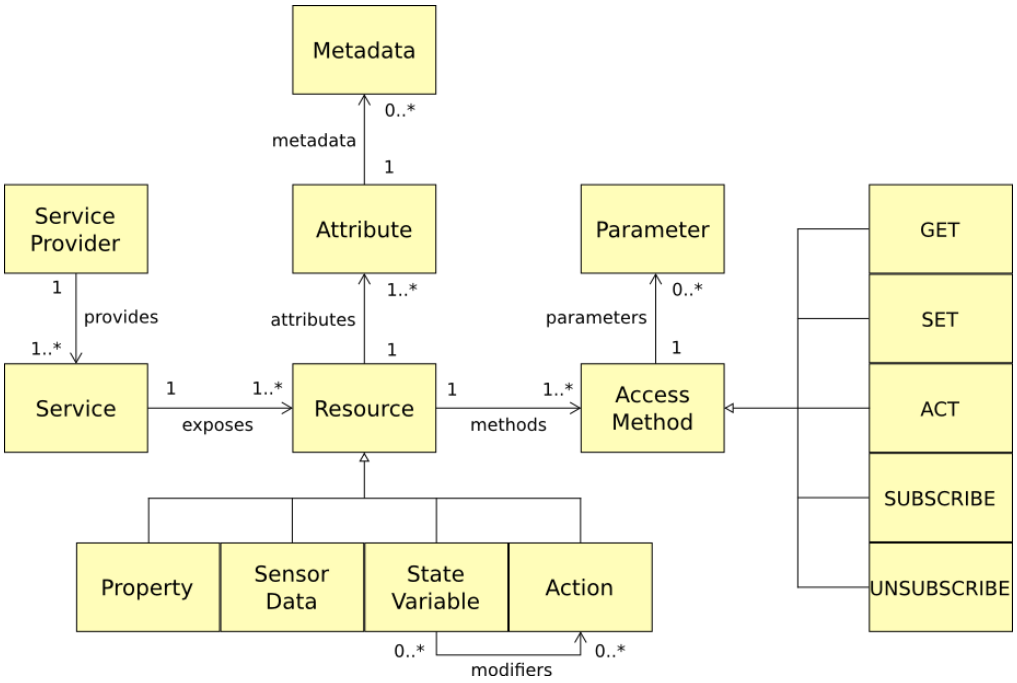

FIGURE 15 GENERIC RESOURCE/SERVICE MODEL.

Resources and services can be exposed for remote discovery and access using different communication protocols, such as HTTP REST, web sockets, JSON-RPC, MQTT, etc., and advanced features may also be supported (as semantic-based lookup). Resources are classified as shown in Table 1, while the access methods are described in TABLE 2.

TABLE 1 CITY RESOURCE TYPES AND DESCRIPTIONS

TYPE	DESCRIPTION
SENSORDATA	Sensory data provided by a service. This is real-time information provided, for example, by the SmartObject that measures physical quantities.
ACTION	Functionality provided by a service. This is mostly an actuation on the physical environment via an actuator SmartObject supporting this functionality (turn on light, open door, etc.) but can also be a request to do a virtual action (play a multimedia on a TV, make a parking space reservation, etc.)
STATEVARIABLE	Information representing a SmartObject state variable of the service. This variable is most likely to be modified by an action (turn on light modifies the light state, opening door changes the door state, etc.) but also to intrinsic conditions associated to the working procedure of the service
PROPERTY	Property exposed by a service. This is information which is likely to be static (owner, model, vendor, static location, etc.). In some cases, this property can be allowed to be modified.

The access methods that can be associated to a resource depend on the resource type, for example, a GET method can only be associated to resources of type Property, StateVariable and SensorData. A SET method can only be associated to StateVariable and modifiable Property resources. An ACT method can only be associated to an Action resources. SUBSCRIBE and UNSUBSCRIBE methods

can be associated to any resources. TABLE 2 RESOURCE ACCESS provides access methods included in the API.

TABLE 2 RESOURCE ACCESS

TYPE	DESCRIPTION
GET	Get the value attribute of the resource
SET	Sets a given new value as the data value of the resource
ACT	Invokes the resource (method execution) with a set of defined parameters
SUBSCRIBE	Subscribes to the resource with optional condition and periodicity
UNSUBSCRIBE	Remove an existing subscription

Additionally, the resource access layer defines an application model which uses event resources, do some processing and act on action type of resources. Based on the Event-Condition-Actions (ECA) axiom, the application is only triggered when the required events occur. Then, if all conditions are satisfied, the actions are done. Events are created from data type of resources and the actions are performed using the specified actuators available in the environment.

An application is considered a set of bound components. Each component processes a single function (e.g., addition, comparison, action). The result of this function is stored in a variable in the current instance of the application. The components using this result as input listen to the corresponding variable. When the variable changes, they are notified and can process their own function leading to a new result.

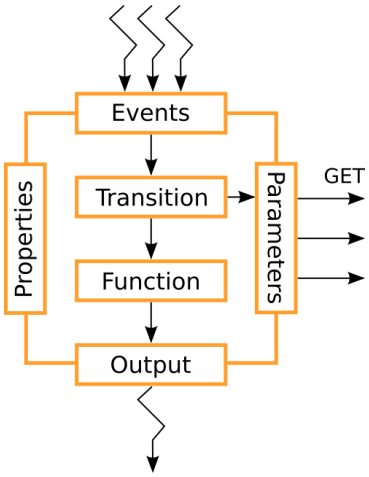

FIGURE 16 ARCHITECTURE OF AN APPLICATION COMPONENT

The component is the atomic element of an application; thus an application can consider a single component to perform an action. It holds the minimal requirements to create an ECA application:

- Events: events that trigger the process of a component. Trigger can be conditioned to a specific event or a specific value of the event (e.g., when the value of the sensor reach a threshold);
- Function: function wrapped in the component (e.g., addition, comparison, action). The acquisition of the parameters is realized in the transition block before the function block;

- Parameters: parameters of the function that are not available in the event (e.g., static value, sensors values).
- Output: result of the function that is stored in a variable and that triggers a new event.
- Properties: non-functional properties of the component (e.g., register the result as a new resource).

The Applications are managed by a service (AppManager Service) which provides lifecycle management resources (such as INSTALL and UNINSTALL applications). It enables to process various checks during different steps of the lifecycle of the application (e.g., syntactic consistency, resources permissions). The first step is to install the application, i.e., send the description of the application. If there is a problem, the AppManager Service returns an error. Once the application is installed, it can be started and its state changes to "Resolving". If there is a problem during this step, the application enters in the "Unresolved" state. Otherwise, the application is active until it is stopped or an exception occurs.

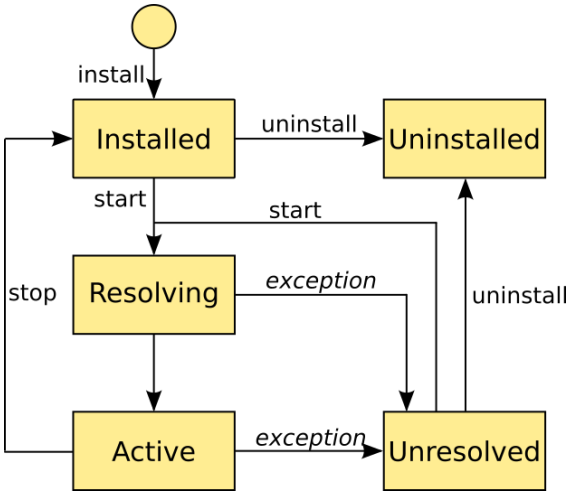

FIGURE 17 LIFECYCLE OF AN APPLICATION

The AppManager allows multiple instances of the same application to run in parallel. When an event occurs, the InstanceFactory of the application instantiates a new set of components and passes the event to the first component. The number of instances can be set in the application properties. If, there is more events than available instances, events are stored and processed when an instance ends.

This generic data model can be extended to the application model described in Figure 18 below, including the possibility for an application component to instantiate a resource. In other words, processing functions transforming one or several events into other high level evets, can be considered as new resources, which can be subject of entry points for new functions,

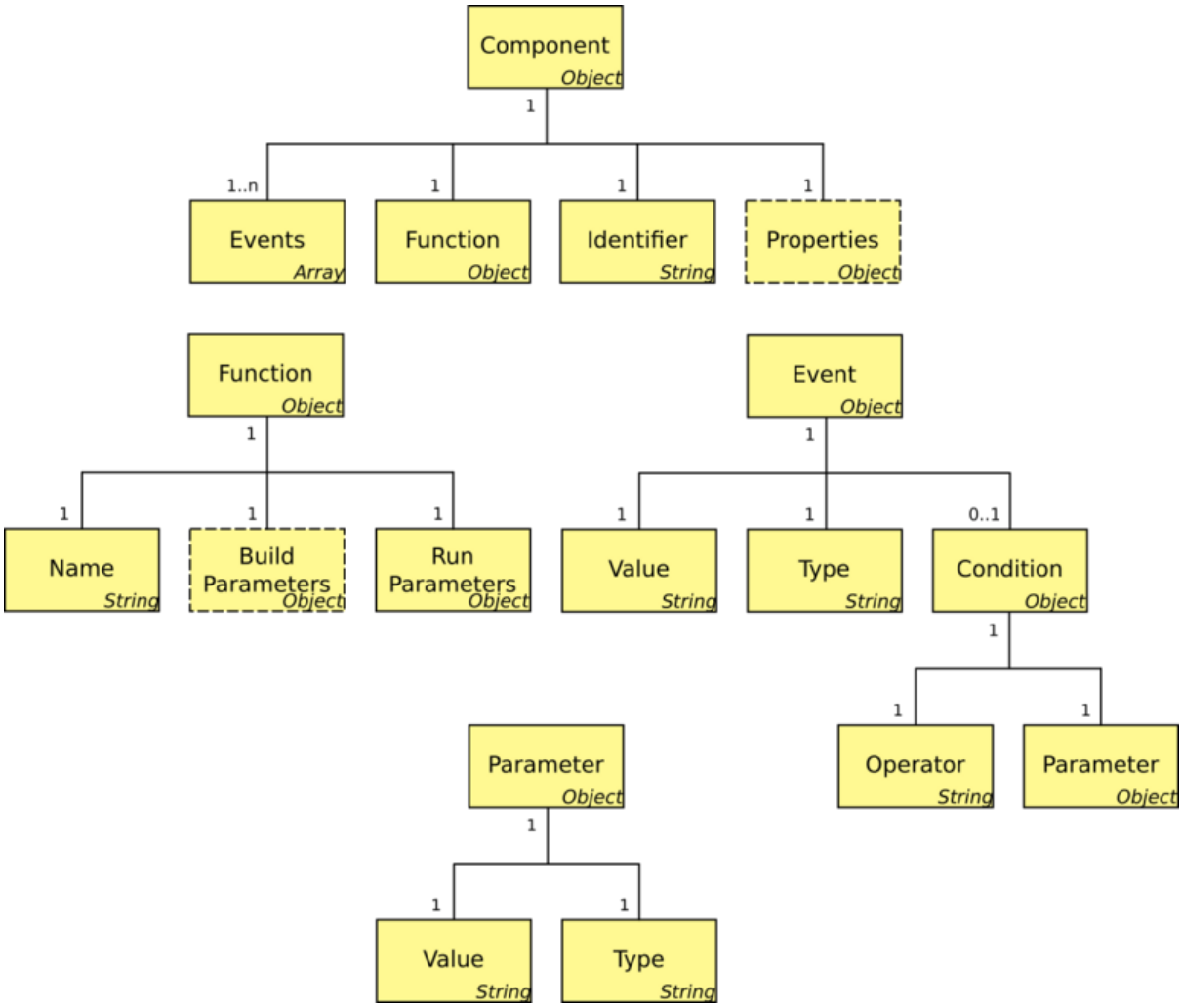

FIGURE 18 RULE BASED APPLICATION MODEL

Next section provides more details in terms of implementation and provides examples

5.3 Implementation

sensiNact platform developed by CEA provides the core functionalities of the resource access layer. Its architecture is depicted in the Figure 19.

Responding to the requirements in terms of homogenised access to various resources exposed by different city entities, the interactions between the sensiNact and other entities are performed through an extensible set of northbound and southbound bridges.

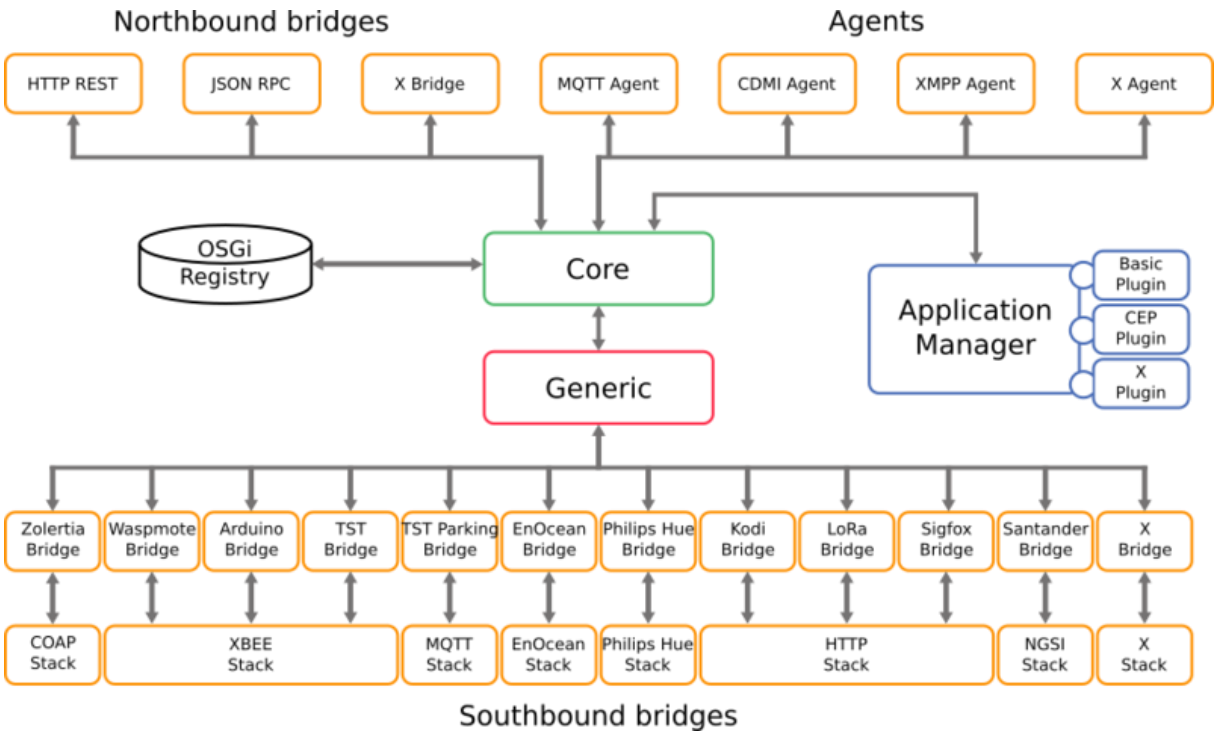

FIGURE 19 SENSINACT ARCHITECTURE

Southbound bridges are specialized into the interaction with devices, which can be sensors or actuators. Each bridge is in charge of the communicating with a specific kind of device, using a given protocol. Out of the box, sensiNact ships with southbound bridges for using a lot of common devices including Zigbee (motion sensors, force sensor,etc.), EnOcean (remote controls, windows opener detectors, etc.), CoAP (sliders, buttons, etc.). It also provides a bridge for retrieving context information using NGSi 9/10 protocol. Thanks to an OSGi based architecture, it's possible to add bridges on the fly, while the gateway is running, to allow the communication with new kind of devices. Of course, the creation of bridges relies on an API which delegates most of the integration work to the gateway, letting the programmer focus on the communication protocol and the data model of the device to be integrated.

Symmetrically to the southbound bridges, **northbound bridges** are in charge of publishing information to remote systems. It can be using common protocols, for example MQTT, XMPP, NGSi 9/10. The set of northbound bridges is also extensible, for tailoring special needs or singular systems. The REST API, which is a northbound bridge, is a key part in our architecture. It is designed for the administration of the gateway, thanks to a well-documented API. This administration can be performed directly by an administrator, or using third parties, such as a studio (see below).

A pool of sensiNact northbound and southbound bridges including EnOcean, LoRa, MQTT, OpenHab, are available today. Below is the complete list of the supported protocols so far.

TABLE 3 SENSINACT SUPPORTED PROTOCOLS

Protocol	Domain	Layer	Connection edges
Android IMU	accelerometer gyroscope and magnetometer	Sensor	Device to WAN
BLE	Personal Area Network	Sensor	Device to Gateway
COAP	Generic	Sensor	Device to Gateway
EchoNet	Smart Building	Sensor	Device to Gateway
e-Lio app	TV activity, social networking, sound	Gateway	gateway
EnOcean	Smart Building	Sensor	Device to Gateway
Free Mobile	sms	Service	Gateway to WAN
HTTP REST	Generic		
KNX-RF	Generic	Sensor	Device to Gateway
KODI	TV	Device	Device tyo Gateway
LoRa	Outdoor	Sensor	Device to Gateway
LoRaWAN	Outdoor	Sensor	Device to WAN
MQTT	Generic	Service	Gateway to WAN
Netatmo	Weather station	Sensor	Device to LAN
NFC ACR12	RFID card reader	Sensor	Device to Gateway
NGSI 9 & 10 v1	Generic	Interoperability	Gateway to Gateway
NGSI 9 & 10 v2	Generic	Interoperability	Gateway to Gateway
OpenHab	Generic	Gateway	Gateway
Philips Hue	Domotic lighting	Sensor	Device to LAN
Sigfox	Generic	Sensor	Device to WAN
Tikitag	RFID card reader	Sensor	Device to Gateway
X3D	Generic	Sensor	Device to Gateway
Xbee	Generic	Sensor	Device to Gateway
XMPP	Generic	Sensor	Device to Gateway
Z-Wave	Generic	Sensor	Device to Gateway

The **Generic** module provides a set of intermediate data structures at different complexity levels to ease and accelerate the integration of new southbound bridges. The reader can find a tutorial on the sensiNact Wiki[1] to learn more about it.

All the communications managed by sensiNact Gateway are converging to a central piece: the **Core** gateway. This element is in charge of the overall coordination of information, involving two managers: the device manager and the application manager. The device manager, with its associated database stores all the information regarding devices. This include devices availability, devices properties, location, etc.

The goal of the **application manager** (also known as the AppManager) is to instantiate and execute applications, listen for devices availability and update applications lifecycle. Application management is performed through the REST API. The application is sent to the gateway packed into a JSON file. Then, the application is installed by the gateway, and can then be started, stopped or uninstalled.

The application manager provides a plug-in facility to extend its behavior. The two main plugins are the Basis plugin, and the Complex Event Processor plugin. The Basis plugin provides the basic

[1] https://wiki.eclipse.org/SensiNact/Tutorial_Bridge

structure for creating applications. It allows the triggering of rules, based on events. Then, a condition can be evaluated, and actions performed accordingly. Of course, events, conditions and actions are related to devices, using the device manager. Using the Esper Engine, the Complex Event Processor plugin is able to generate high level events based on low level ones. For example, it's possible to know that two events occur in a given time interval, with a given order. The events generated by the CEP plug-in can be used in the Basis plugin.

The sensiNact communication model handles the edge communication level (with sensor and actuator devices) and the cloud communication level (between gateways).

The edge communication model is based on the sensiNact bridges. Bridges are responsible for the automatic device discovery and peering management when the protocol makes it possible, and for the intercommunication between sensor/actuator devices and IoT gateway.

The cloud communication can be managed using a hierarchical model or a distributed model.

Hierarchical cloud communication model. In this model, a sensiNact instance installed on a cloud server is a supervisor that concentrate data coming from/to sensiNact instances installed on IoT gateways. sensiNact instances on IoT gateways are not interconnected.

The sensiNact distributed cloud communication model is illustrated in Figure 20. In the distributed model, sensiNact instances on gateways and cloud servers are all interconnected. There is no supervisor. A cloud deployment of such a distributed communication model has been implemented on the top of the PIAX [1] framework.

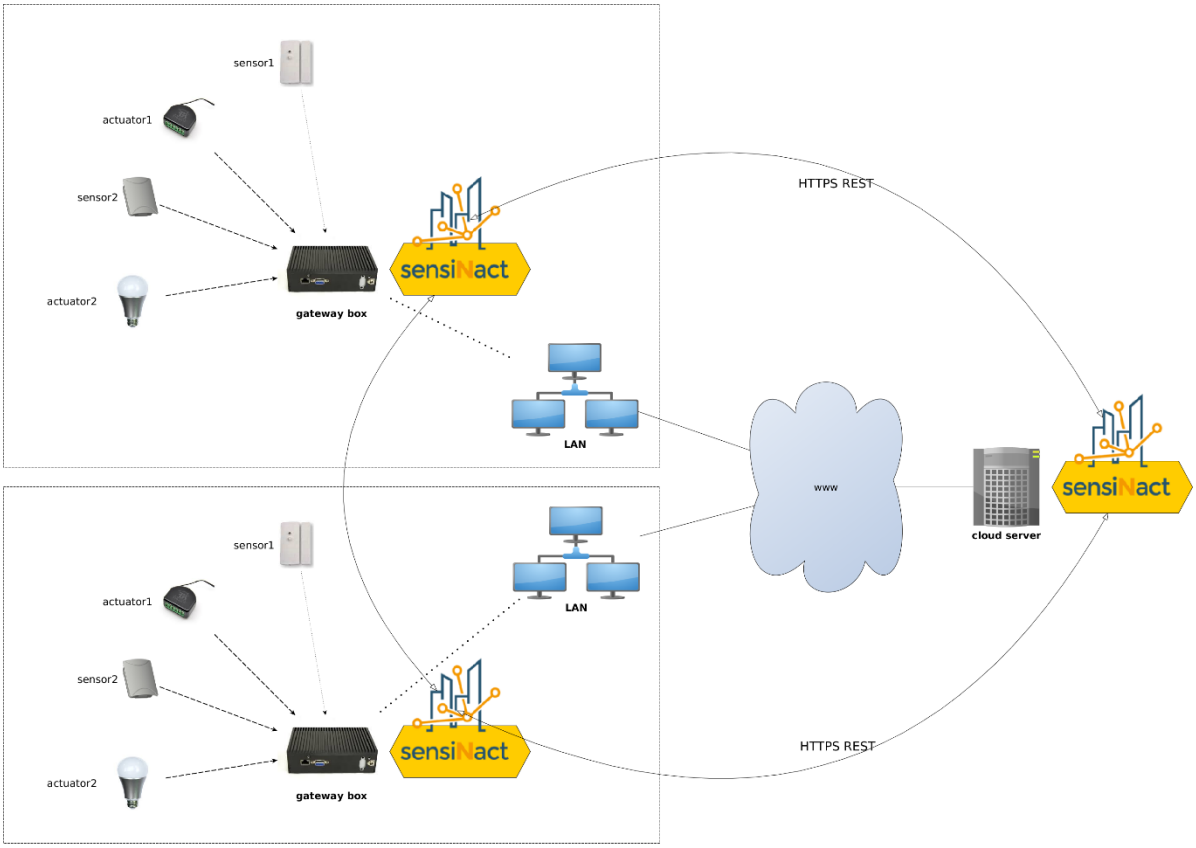

FIGURE 20 SENSINACT DISTRIBUTED CLOUD COMMUNICATION MODEL

More information about the implementation of the sensiNact Core modules and the AppManager can be found at https://wiki.eclipse.org/SensiNact/Gateway_Core and at https://wiki.eclipse.org/SensiNact/AppManager respectively.

The next section introduces the security and dependability layer, which has a strong link with the core resource access layer.

6 SECURITY & DEPENDABILITY

The Security & Dependability functional block copes with all security, privacy and dependability concerns for the entire platform. The components of this functional block, including Trust and Privacy, Authentication Authorization Accounting, and Dependability Framework, provide their functionalities to all the other components in the platform. The specification of the Authentication Authorization Accounting component has been inherited from the ClouT project. However the implementation of the component has been done during the BigClouT project. For Dependability, the approach to its core functionality "by design" has also been mostly inherited from the previous effort in the ClouT project. In the BigClouT project, the improvement has been made on the essential extension for "at-runtime" capabilities for dependability, and that is, self-awareness. Additionally, there are also efforts made to enhance trust and privacy as the overall qualities of the platform.

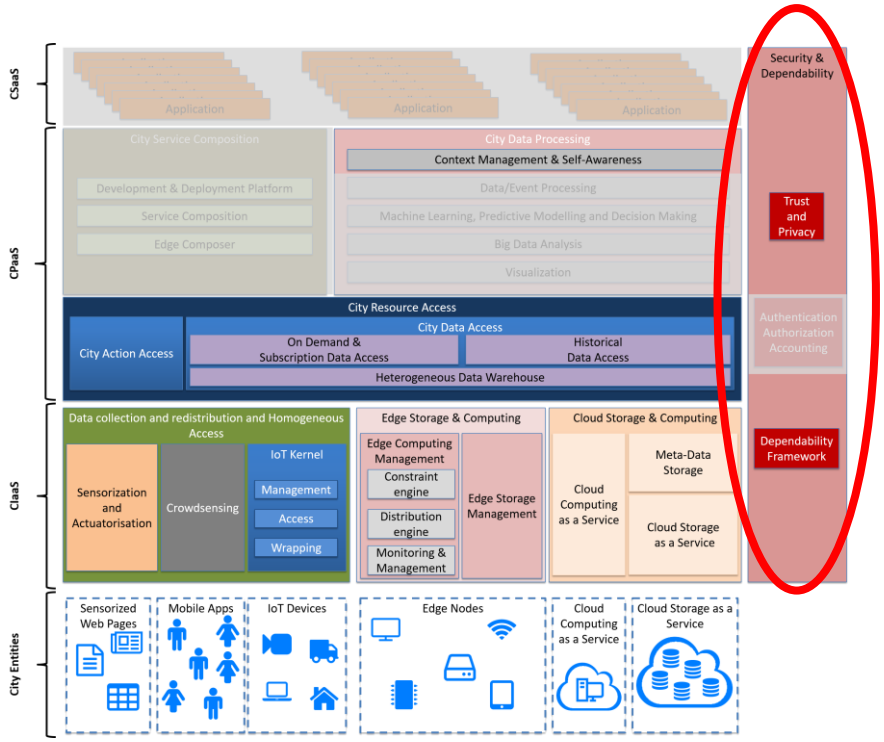

FIGURE 21 SECTION OF THE ARCHITECTURE INCLUDING TRUST, PRIVACY AND DEPENDABILITY

6.1 Security, Trust and Privacy

6.1.1 Architecture

The security, trust and privacy component is in charge of reifying and maintaining the data structures, holding access rights to existing resources as well their level of trust according to who is providing them. The main principle to ensure trust and privacy is first to keep as little private data as possible in the system. It is also to anonymize those that will be used for analysis. For example, the anonymization starts by breaking the link between the identifier of the user in the system and the identifier to which analysed data records are linked to, in manner of preserving all possible statistic calculations but disallowing the simple association between a data record to a unique person. Whatever the anonymization mechanisms are, some private data will remain into the system because of their use in the case of social features; for example, in the Innovallee Grenoble field trial profile sharing when creating an event and opening it to other users. In manner

of being compliant with the new European rules relative to private data known as GDPR, we apply its seven principles in the private data management:

- *Loyalty and lawfulness of treatment:* the processing of data must be carried out for legitimate reasons and its use must conform to the regulations and transparent.
- *Collection of consent:* the consent of the individual whose data is collected and processed must be express, that is, it must result from a positive act. It cannot be the result of a silence or a checkbox that is checked by default.
- *Purpose of treatment*: The data that is being processed must be treated for a specific, explicit and legitimate purpose. This purpose cannot be altered: The data collected for a treatment cannot be processed for a purpose other than that for which the persons have given their consent.
- *Proportionality*: It is only possible to collect data that is adequate, relevant and not excessive to the purpose of the treatment.
- *Data security*: The controller has the obligation to undertake all the necessary measures to ensure the security of the data and to avoid their disclosure to unauthorised third parties.
- *Accuracy of data*: the data collected and processed must be accurate. To do this, it is better to promote a regular update of the data in question. Measures must be put in place to ensure that inaccurate data is erased or rectified. The most appropriate way to comply with this principle is to set up an application that allows users to modify their own data.
- *Deleting data*: The data that is no longer needed should be deleted. The shelf life is variable and depends on the nature of the data and the purposes pursued.

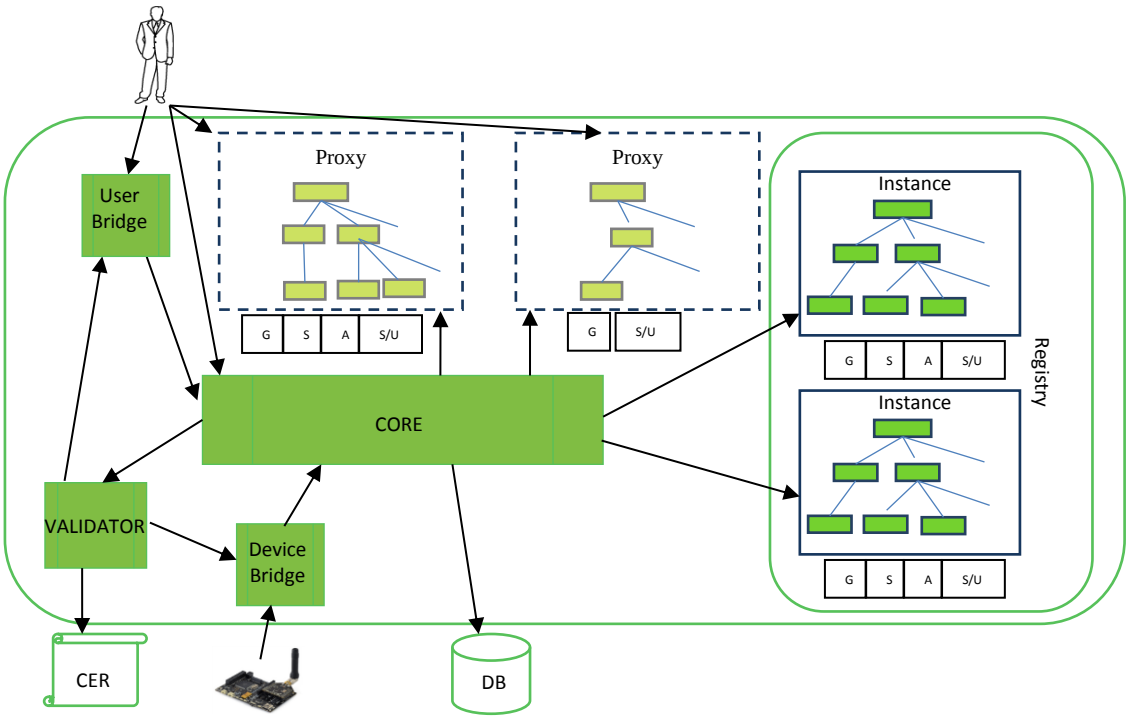

FIGURE 22 SECURING PRIVATE DATA

In particular to respect the data security principle, we use the same model that we use for city entities (cf. Figure 22): we reify an instance of the BigClouT inner system's service model to host the private data, only accessible by the user. It means that sharing personal data result only in an explicit sharing request coming from the user itself.

6.1.2 *Specification*

Permission to use a service is required in order to access resources regarding the service. Untrusted clients should not be able to use the services, event not aware of their presence. The resource access layer defines service permissions in such a way that access to the ones it provides is forbidden except if a specific condition is met (a specific conditional permission). The condition is that the client is an instance of "Secured Access Service". The clients can be one of the following five profiles:

- Owner,
- Administrator,
- Authenticated,
- Anonymous,
- Unauthorized.

A UserProfile can be defined at each level of the hierarchical resource model: ServiceProvider, Service, and Resource.

When asking for a data structure of the resource model, the access rights of the user are retrieved; the set of this user's accessible access methods for the specific data structure is built and returned as part of the description object. Each future potential interaction of the user on the data structure will be made by the way of this description object. For a remote access, a security token is also generated and transmitted to the user, to avoid repeating the security policy processing. A token is defined for a user and a data structure (and so it previously created description object).

The Security & Dependability functional block is used for authentication and to retrieve identity material from which it will be possible to associate a user and a resource model data structure to a use profile.

6.1.3 *Implementation*

The security mechanism for the resource access layer has been implemented by the sensiNact platform. Two levels of security are implemented. A first security level handled by the underlying service framework that hosts the sensiNact platform, which is the OSGI security framework, to secure installation and activation of software modules. The second security level tackles the user accesses to registered resources with respect to authentication (login/password authentication) and thus is provided as a service to the application layer.

OSGi [2] Framework which provides a first level of with its ServicePermission and ConditionalPermissionAdmin services. The ServicePermission is a module's authority to register or use a service:

- The register action allows a module to register a service on the specified names.
- The get action allows a module to detect a service and use it.

Permission to use a service is required in order to detect events regarding the service. Untrusted modules should not be able to detect the presence of certain services unless they have the appropriate ServicePermission to use the specific one. The ConditionalPermissionAdmin is

[2] https://www.osgi.org/

framework service to administer conditional permissions that can be added to, retrieved from, and removed from the framework.

In addition to the database managed by the Security & Dependability functional block, used to authenticate a user and to retrieve its identity in the system, the sensiNact platform manages an internal database allowing to link this identity to a UserProfile for a specific data structure. For all data structures for which the user has not been registered the Anonymous user profile is used by default (except if the owner of a resource has defined this default profile to another one). The internal database also gathers information relative to the minimum required UserProfile to access to data structures. This definition can be made at each level of the resource model, knowing that if no UserProfile is defined for a data structure, the one specified for its parent is used.

For example, according to the figure below, a user trying to access to the ServiceProviderX for which its UserProfile is Anonymous will receive a description object in which only one Service will be referenced (ServiceX1), containing a single Resource (ResourceX1S2) providing two AccessMethods: GET and SUBSCRIBE.

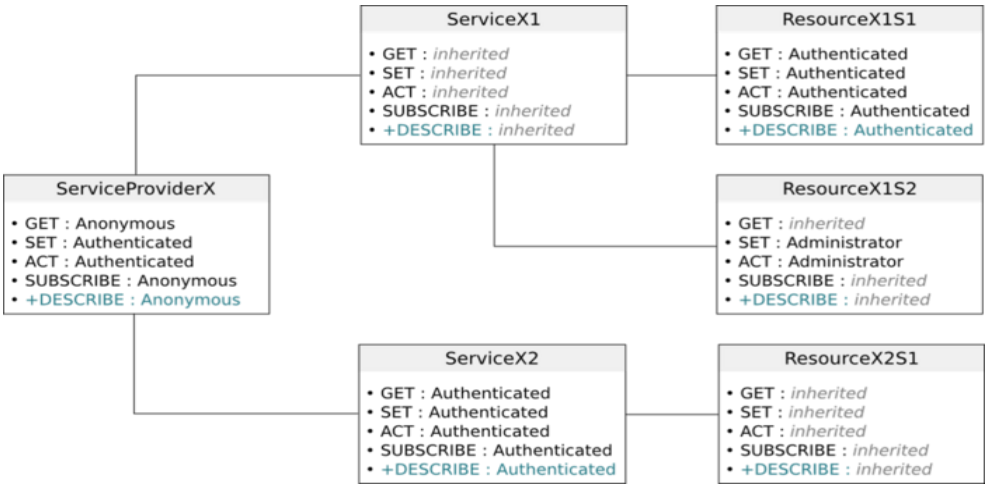

FIGURE 23 EXAMPLE OF SECURITY RIGHT INHERITANCE AMONG A SERVICE PROVIDER, ITS SERVICES AND RESOURCES.

More details about the implementation of the security mechanism implemented for the City resource access layer (aka sensiNact platform) can be found at: https://wiki.eclipse.org/SensiNact/Gateway_Security

6.2 Dependability and Self-Awareness

The self-awareness feature of the platform aims at maintaining an up-to-date system context and being able to adapt the platform configuration and behaviour in response to changes of context. As a cross-platform feature, it is connected to other features through the intermediate of the Data/Event Processing component which produces context information and knowledge, etc. Meanwhile, the self-awareness mechanism will also connect to certain adaptation APIs provided by other components to change the behaviour of the platform. The exact self-aware capabilities or functionalities that different components want to achieve vary depending on the concrete use cases. In this section, a generic self-awareness mechanism will be firstly introduced in Section

6.2.1 and Section 6.2.2. Following that, Section 6.2.3 articulates an implementation applying the described generic self-awareness mechanism to the Service Composition component.

6.2.1 Architecture

It is increasingly difficult to completely capture the requirements and environments of BigClouT-like systems, systems that strongly interacts with the real, physical world and perception of end users. In other words, some of the decisions made at the design time can be invalidated at runtime due to the uncertainty at the time of the design and due to changes in requirements or environments at runtime. A promising approach is then to check the validity of the decisions and correct them by the system itself, i.e., self-awareness, leading to capabilities such as self-healing and self-adaptation, etc. This approach is known as "models@run.time" because we use models of requirements, environments, and designs at runtime, which, previously, were typically used only at the design time to make decisions once. The models should be continuously synchronized with the system implementation or the sensed environmental information, thus representing the systems at runtime in terms of its requirements, environments, and designs.

In terms of the architecture, it is significant to consider the design principle of Separation of Concerns so that capabilities of healing or adaptation can be flexibly attached and extended. Separation of Concerns encourages the separation of a program into distinct sections with each section addressing a separate concern. In light of this principle, the target components that are to have the self-X capabilities such as self-healing or self-adaptation shall provide reflection APIs, so that the runtime information of the components is monitored and updated without modifying their internal state. Then, we can think of external components responsible for monitoring, analysis, or reasoning about the adaptation strategies regarding the components (typically, in the MAPE-K loops [2]: Monitor, Analyse, Plan, and Execute by leveraging Knowledge). Figure 24 illustrates this architectural principle.

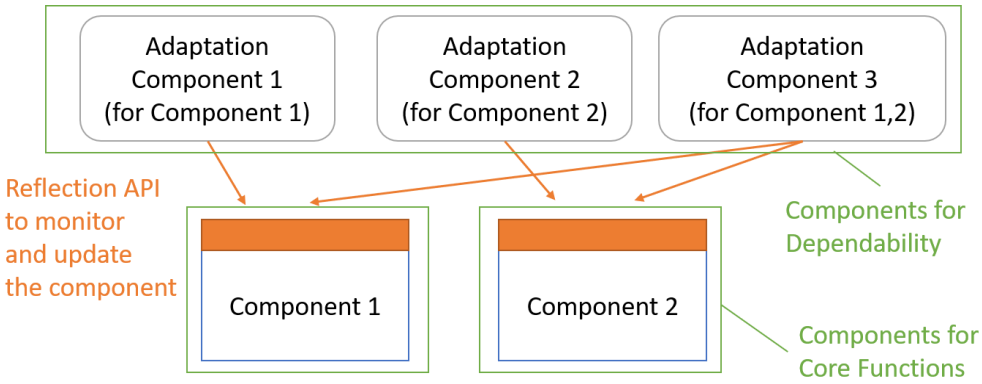

FIGURE 24 ARCHITECTURAL PRINCIPLE FOR SELF-AWARENESS

6.2.2 Specification

Implementation of the self-adaptation functions greatly depends on the target core function components. Optimization techniques are often used for quantitative dependability properties such as reliability by resource allocation or replication techniques. Reasoning on the state transitions is also used for correctness and consistency of the application behaviours.

As the common architectural specification, we define the reflection APIs rather than the APIs of a variety of self-adaptation components.

- getModel(): get the model from the target component about its runtime states and relevant information necessary for reasoning.
- subscribeModelChange(topics): subscribe to changes in (the model of) the target component
- updateModel(updatedModel): update the model so that the target component adaptively changes its behaviour or configurations.

6.2.3 Implementation

Service Composition allows to develop and deploy IoT applications in the platform, which are one main objective of the project. For such a platform, its self-awareness is targeting more towards enabling the design and implementation of distributed IoT applications that meet self-awareness requirements guarantying self-controlling and self-optimizing characteristics. The platform adopts a data-flow programming approach for the service composition features. It will allow to build IoT applications using heterogeneous sources and automatic configuration to execute application in a distribute way leveraging both cloud and edge computing principles.

We focus on event-driven IoT applications that make use of not only sensing but also actuating capabilities. When deploying such applications, conflicts can easily arise between multiple applications, users, and devices. Self-adaptation is necessary to keep consistency upon changes such as installation of new applications and unavailability of devices.

Figure 25 shows our implementation approach to extend the Service Composition function by self-adaptation. Service composition in IoT is often event-driven.

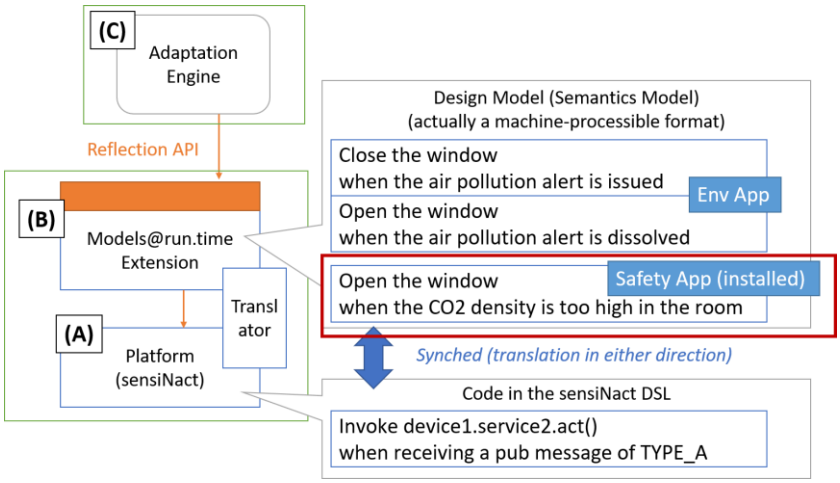

FIGURE 25 SELF-ADAPTATION FOR SERVICE COMPOSITION

The component (A) in the figure is the target platform. As an example, the name of sensiNact is specified. Our approach is to minimize the necessary changes in this part.

The component (B) complements the platform and holds the design model. The design model contains the semantics information of the implemented code in the platform. The figure shows examples of the information held by the platform and the model in the call-outs. The code keeps

only the information necessary for execution. Actions in the code refer to end points of invocation where as those in the design model denote the effects over the cyber and physical state spaces.

It is possible to expect a development process in which a design model is originally used before implementation of the code. However, in the majority of cases there is no such design model suited for verification or repair. We therefore provide a translator from the code to the design model by using metadata. The metadata gives the semantics information on elements in the code, e.g., mapping between "device1.service2.act()" in the code and "open the window" in the design model.

This translation has already been used for our previous work for verification at development time in the ClouT project. When used for models@run.time, the translation from the design model to the implementation code is also necessary. This is because the runtime adaptation is realized by rewriting the design model and reflecting the change to the implementation code.

In addition to the design model, the component (B) also holds a model of the environmental information. We omit the detail of this point. The environment model denotes the present state of the environment, such as availability of devices. The model also denotes assumptions on the behaviour of the environment, e.g., the window can be "locked" only if it is "closed". The search space of verification depends on the assumptions to define all the possible scenarios in which the system behaviour is examined.

The component (C), adaptation engine, implements the adaptation logic. For our problem of repair to remove potential conflicts, there can be many potential techniques. We use model checking techniques to generate "meaningfully diverse" scenarios of conflicts so that users understand the impact of conflicts and make decisions on the priority in different situations. We may also apply a generic technique of controller synthesis, which generates or updates the behaviour that satisfies given properties under a given set of environmental assumptions.

To be more specific, the translator between the component A – the SensiNact Platform, and the component B – the Models@run.time Extension – is being implemented by mapping the original Event-Condition-Action (ECA) rules specified by developers in SensiNact Studio to another domain-specific rule specification language designed for "models@run.time" in this project. One example of the ECA rules for the environment application (i.e. Env App shown in Figure 25) implemented in the SensiNact Studio could be specified as following in the text box:

```
resource pollution_alert = [localgateway/airSensor/airPollution/state]
resource window = [localgateway/window/switch/state] //the window sensor
resource OPEN = [localgateway/window/switch/open] //the open window actuator
resource CLOSE = [localgateway/window/switch/close] //the close window actuator

on pollution_alert.subscribe()    //the air pollution alert is triggered
if window.get() == true           // if the window is currently open
do CLOSE.act()                    //close the window
end if
```

Then, through the translator the corresponding higher-level design model of the above specified ECA rules will be generated, using the XText framework. The generated model shall capture not only the original ECA rules but also general information about the environment, e.g. possible states that the window can be. Figure 26 represents part of the generated Env App model.

env_app.spec

```
environment {
    resource window can be OPEN, CLOSED, LOCKED
    resource pollution_alert can be ON, OFF
}

app EnvAppMeta {
    spec1
    on pollution_alert == ON
    if window == OPEN
    do window := CLOSED
```

FIGURE 26 MODEL TO BE GENERATED FROM ECA RULES FROM SENSINACT STUDIO

At the moment, the translator work is in progress and the next step is to integrate with the Adaptation Engine so that the engine can produce adaptation strategies according to the design of the generated model. Then we will test the implementation and then integrate the solution from end to end with the SensiNact Platform.

7 CONCLUSION

This deliverable describes the first release of the self-aware distributed city data platform in the BigClouT project. It clarifies the three main capabilities offered by the platform, namely 1) the Data Collection capability that enables collecting and maintaining the real-time and historic data; 2) the Data Distribution capability that provides on-demand or publish/subscribe data in addition to historic data; and 3) the Self-Awareness capability that enables reconfiguration of the platform in response to context updates. More importantly, to realize the three capabilities, the current design and implementation details are articulated in the document according to the composed functional blocks of the platform. It should be noted that the platform is developed on top of the BigClouT architecture and baseline implementations adapted and extended from the ClouT project. Therefore, three functional blocks are described in detail which encompass the main updates made since the first year of the project, which are are mainly the Data Collection, Redistribution and Homogeneous Access functional block, the Edge Storage & Computing functional block, and the Security & Dependability functional block.

BIBLIOGRAPHY

[1] BigClouT, *Deliverable 2.1 Data collection tools and architecture.* 2017. https://projectnetboard.absiskey.com/viewdocument/6c3e77-c4d6b8-cdd3c9-8dc181-000016

[2] Kephart, J.O. and D.M. Chess, *The vision of autonomic computing.* Computer, 2003. **36**(1): p. 41-50.

[3] BigClouT, *Deliverable 1.4 Updated use cases, requirements and architecture". 2017* https://projectnetboard.absiskey.com/viewdocument/6c3e77-c4d6b8-cdd3c9-8dc181-000016

[4] *ClouT, Deliverable 4.2 Middle report of city application developments and in-lab evaluation and field trials. http://clout-project.eu/documents/deliverables-access/*