SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

Project Number:	**FP7–SMARTCITIES–2013(ICT)**
Project Acronym:	**VITAL**
Project Number:	**608682**
Project Title:	**Virtualized programmable InTerfAces for innovative cost-effective IoT depLoyments in smart cities**

D4.1.2 Intelligent virtualized Discovery of resources V2

Document Id:	VITAL-D412-231015-Draft
File Name:	VITAL-D412-231015-Draft.pdf
Document reference:	Deliverable 4.1.2
Version:	Draft
Editors:	Nathalie Mitton, Valeria Loscrí, Riccardo Petrolo, Salvatore Guzzo Bonifacio
Organisation:	INRIA
Date:	23/10/2015
Document type:	Deliverable
Security:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Salvatore Guzzo Bonifacio	INRIA	08/07/2015	Initial table of contents and updates to previous version
V02	Salvatore Guzzo Bonifacio	INRIA	24/07/2015	Update of Discoverer interface definition
V03	Salvatore Guzzo Bonifacio	INRIA	27/07/2015	Changed document structure. Updated enhanced mechanisms
V04	Salvatore Guzzo Bonifacio	INRIA	29/07/2015	Updated discovery for ICOs, Services and Systems
V05	Aqeel H. Kazmi	NUIG	26/08/2015	Updated Data Management Sytem DMS section
V06	Angelos Lenis	SiLo	06/09/2015	Updated Section 5.3: Interfaces to Vital Orchestrator
V07	Angelos Lenis	Silo	11/09/2015	Removed an operation from SD that is not supported in Section 5.3
V08	Elisa Herrmann	ATOS	08/10/2015	Updated Section 5.3
V09	Salvatore Guzzo Bonifacio	INRIA	08/10/2015	Updated section 4, Updated section 6, Minor Fixes
V10	Katerina Roukounaki	AIT	09/10/2015	Review
V11	Salvatore Guzzo Bonifacio	INRIA	09/10/2015	Addressing comments
V13	Aqeel H. Kazmi	NUIG	09/10/2015	Comments addressed in DMS section.
V14	Katerina Roukounaki	AIT	09/10/2015	Addressed comments
V15	Angelos Lenis	SiLo	12/10/2015	Updated Section 5.4: Interface to Vital Management
V16	Salvatore Guzzo Bonifacio	INRIA	12/10/2015	Minor fixes and Technical Review
V17	Aqeel Kazmi	NUIG	22/10/2015	Quality Reviews
V18	Salvatore Guzzo Bonifacio	INRIA	23/10/2015	Final Edits
V19	Aqeel H. Kasmi	NUIG	23/10/2015	Circulated for Approval
Draft	Martin Serrano	NUIG	23/10/2015	EC Submitted

OVERVIEW OF UPDATES/ENHANCEMENTS OVER D4.1.1

Section	Description
Section 1	Updates to introduction
Section 4	Updates to Discoverer implementation
Section 5	Updates to interaction with other VITAL components
Section 6	Updates to conclusion and further directions

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TERMS AND ACRONYMS

API	Application Programming Interface
BPM	Business Process Management
CEP	Complex Event Processing
DMS	Data Management Service
HTTP	Hypertext Transfer Protocol
ICO	Internet-Connected Object
IoT	Internet of Things
JAVA EE	Java Enterprise Edition
JSON	JavaScript Object Notation
JSON-LD	JSON for Linking Data
OWL	Web Ontology Language
RDF	Resource Description Framework
REST	REpresentational State Transfer
SOAP	Simple Object Access Protocol
SPARQL	SPARWL Protocol and RDF Query Language
URI	Uniform Resource Identifier
URN	Uniform Resource Name
VOB	VITAL Orchestration / BPM Module
W3C	World Wide Web Consortium

1 INTRODUCTION

This deliverable specifies the architecture and features of the Discoverer module and its positions it in the VITAL framework. It also gives details about its implementation.

The main task of this component is to provide the means for discovering ICOs, IoT services and IoT systems that are virtualized in the VITAL platform. The VITAL Discoverer module is additionally enhanced through the use of smart predictive mechanisms to provide more sophisticated functionalities. Other VITAL modules (e.g., CEP, Orchestration) will use the functionalities of the Discoverer to operate on the IoT resources that are needed for their particular business context.

A key element of the Discoverer module will be the discovery of ICOs. It will provide tools for discovering ICOs using inputs such as location and type. The Discoverer plays an important part in ensuring that the data sets which are accessed, processed and visualized are restricted to those that are relevant to the application context, in compliance with the policies of every physical network.

In the final version of the document we finalized the structure of the operations that the Discoverer, as a sub component of VITAL system, will provide to all other components. Such components are mainly other elements of the added value layer, but the Discoverer can also be accessed, if needed, by components residing further up in the architecture (e.g., the Development Tools).

The deliverable is produced with different audiences in mind: first, VITAL consortium members, notably researchers and engineers, who require the functionalities of the Discoverer module in order to develop their components; and second, external researchers and developers who want to use VITAL IoT resources in the development of their own applications.

The document is structured as follows. We first introduce the Discoverer component and its features; we present the technologies we use in development; we show how the Discoverer is implemented, designed, finalised and connected to the other modules of the VITAL architecture. We conclude the deliverable with a summary and an outline of the next steps.

2 DISCOVERER MODULE

The Discoverer module is responsible for discovering ICOs, systems, and services and is horizontally integrated in the VITAL platform as shown in Fig. 1. Furthermore, it will deal with IoT resources requested by other modules without regard to the underlying platforms, focusing only on the meaning of the information.

Fig. 1 also shows how the Discoverer interacts directly with the Data Management Service (DMS) which has storage capabilities, and where the various data streams (and their metadata) are modelled and formatted according to the VITAL ontology – see [Vital-D3.1.1-2014].

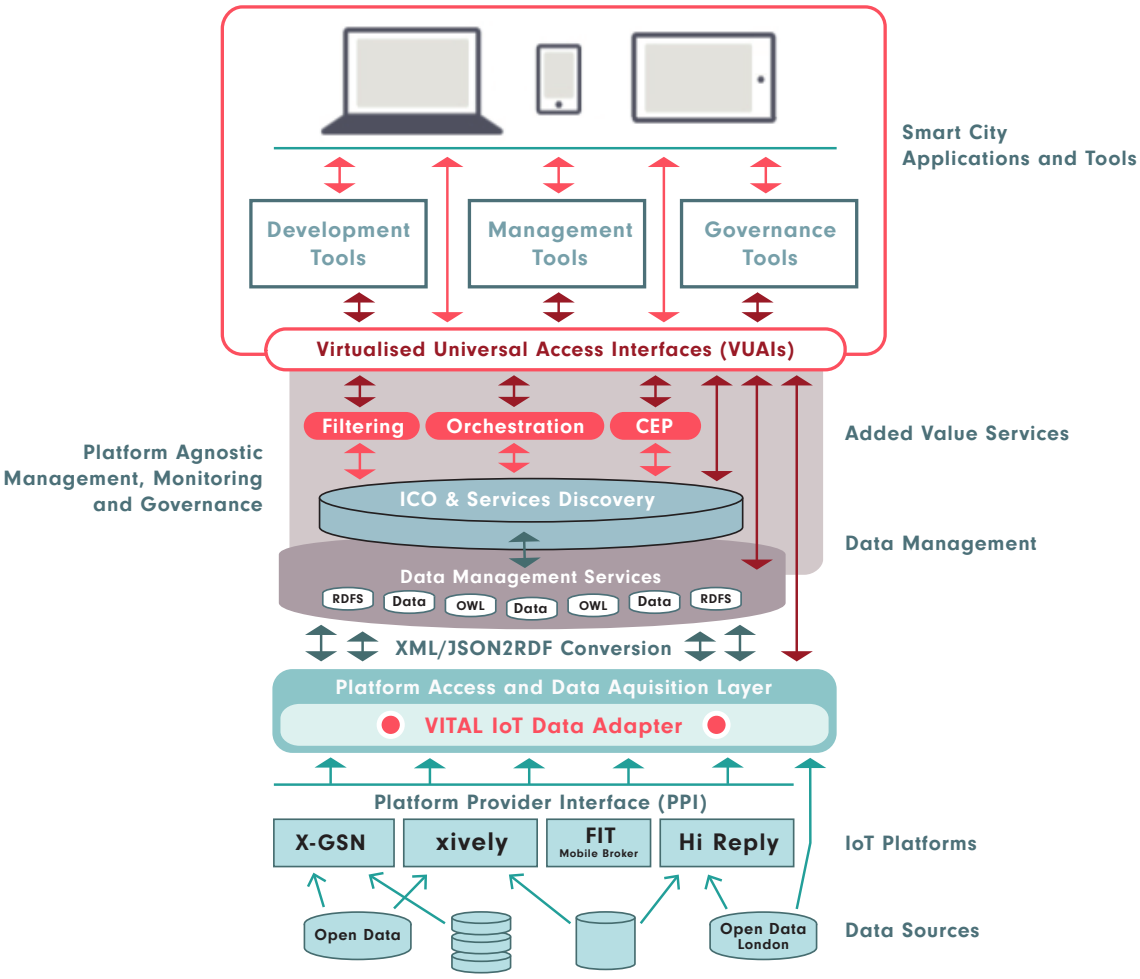

Figure 1. VITAL Architecture

The other modules in the architecture (CEP, Orchestration) will use the functionalities of the Discoverer to access IoT resources according to their particular business context. For example, to validate the Use Case (UC07) described in [Vital-2.2-2014], the CEP Module will access the Discoverer in order to retrieve the ICOs available in a defined location.

2.1 ICO classification and Intelligent Discovery

The Discoverer has been implemented incrementally, starting with a basic version. It relies on a classification of the ICO such as that shown in (Table 1). In this example, an ICO is classified by reference to its motion and further qualified with information about the connectivity and localization capabilities of the device itself. This will later allow the Discoverer to include in its results information regarding the data availability of mobile devices in a defined area. In this version Discoverer will only produce URIs acting as IDs.

Table 1. Events Categorization

ICOs	Localization	Connection
Fixed	continue/ intermittent	continue/ intermittent
Mobile - predictive pattern	continue/ intermittent	continue/ intermittent
Mobile - random pattern	continue/ intermittent	continue/ intermittent

To enhance the service and make the discovery intelligent, the VITAL module will be equipped with smart predictive mechanisms through which it will be able to forecast the connectivity or data availability of a specific device based on its classification.

3 TECHNOLOGIES

In this chapter, we offer a short review of the technologies that could potentially be used for the implementation of the Discoverer. The details (heuristic details on data model and data format for Linked Data) are in deliverable Vital-D3.1.1-2014. We introduce the REST architecture and its basic concept. Then we briefly discuss JAVA EE as a programming technology for developing the REST interfaces and manipulating JSON data.

3.1 REST

Representational State Transfer (REST) is a term coined by Roy Fielding in 2000 [RFielding00] to refer a software architectural style.

Before the introduction of REST, the first generation web services relied on exchanging XML packets conforming to SOAP (Simple Object Access Protocol) specification using HTTP protocol.

SOAP and XML are now considered too heavy, and HTTP itself has sufficient capabilities to allow applications to communicate over the network.

REST ignores the details of the components' implementation and protocol syntax in favour of focusing on the roles of components, the constraints on their interaction with other components, and their interpretation of significant data elements. It

encompasses the fundamental constraints on *components*, *connectors*, and *data* that define the basis of the Web architecture, and thus the essence of its behaviour as a network-based application.

REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the resource. REST's data elements are summarized in Table 2.

Table 2. REST Data Elements

Data Element	Modern Web Examples
resource	the intended conceptual target of a hypertext reference
resource identifier	url, urn
representation	html document, jpeg image
representation metadata	media type, last-modified time
resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control

A resource represents the key abstraction of information in REST. Any information that can be named can be a resource: a document or image, a temporal service, a collection of other resources, and so on.

Web services APIs that adhere to the architectural constraints are called RESTful. HTTP-based RESTful APIs are defined with these aspects:

- base URI, such as http://example.com/resources/
- an Internet media type for the data. This is often JSON but can be any other valid Internet media type (e.g. XML, Atom, microformats, images, etc.)
- standard HTTP methods (e.g., GET, PUT, POST, or DELETE)
- hypertext links to reference state
- hypertext links to reference related resources

3.2 Java EE

Originally specified by Sun in 1999, Java Enterprise Edition (formerly J2EE) is currently the most widely used web programming technology.

Java EE offers many advantages: independence, portability, multi-layer structure system, efficient development, scalability and stable usability to name but a few, making it a platform of first choice for building web-based application systems. Many implementations are available, both commercial (IBM, HP, Sun, Oracle, etc.) and open-source (WildFly, Apache Geronimo, etc.).

For the development of the Discoverer module, we have used WildFly (http://www.wildfly.org), formerly known as JBoss AS, not least because it supports the latest standards for REST-based data access including JSON.

3.3 Conclusion

The ability of REST to fully leverage the protocols and standards that power the World Wide Web, combined with its simplicity relative to SOAP-based approaches, have contributed to the position where it is now the preferred technology for building web services, including services from large vendors such as Google, Yahoo, Amazon and Microsoft.

In addition, the considerations dealt with in Deliverable [Vital-D3.1.1-2014] (Virtual Models, Data and Metadata for ICOs, V1), also point to REST as the technology that better suits the communications between the different VITAL modules.

Within the VITAL context, we therefore decided to adopt REST for the communication between the different modules of the architecture.

4 DISCOVERER IMPLEMENTATION

4.1 Global implementation

The Discoverer is accessible through a RESTful web service, which exposes information like the context, a description, its status and the operations it offers. Table 3 shows a résumé of the operations available. The `ConnDMS`, for example, gives information about the status of the connection with the Data Management Service. Operations will be further analysed in **Error! Reference source not found.**

Table 3. Discoverer description

```
{
  "@context":"http://vital-iot.org/contexts/service.jsonld",
  "type":"vital:Discoverer",
  "description":"This is the VITAL Discoverer module.",
  "status":"running",
  "msm:hasOperation":
  [
    {
      "type":"ConnDMS",
      "hrest:hasAddress":"DISCOVERER_BASE_URL/ConnDMS",
      "hrest:hasMethod":"hrest:GET"
    },
    {
      "type":"GetICOs",
      "hrest:hasAddress":"DISCOVERER_BASE_URL/ico",
      "hrest:hasMethod":"hrest:POST"
    },
    {
      "type":"GetSystems",
      "hrest:hasAddress":"DISCOVERER_BASE_URL/system",
      "hrest:hasMethod":"hrest:POST"
    },
    {
      "type":"GetServices",
      "hrest:hasAddress":"DISCOVERER_BASE_URL/service",
      "hrest:hasMethod":"hrest:POST"
    }
  ]
}
```


To be compliant with the structure defined for VITAL components in D3.2.2, Discoverer exposes his PPI through a set of web services. Required services are available at the addresses as described in Table 4.

Table 4. Discoverer PPI addresses

Service	***Address***
Get IoT system metadata	`DISCOVERER_BASE_URL/ppi/metadata`
Get IoT system status	`DISCOVERER_BASE_URL/ppi/status`
Get IoT service metadata	`DISCOVERER_BASE_URL/ppi/service/metadata`
Get IoT sensor metadata	`DISCOVERER_BASE_URL/ppi/sensor/metadata`

Discovery operations require to access DMS and request the execution of a SPARQL query. The query to be executed is prepared according to discovery options received in the request. Data obtained by the DMS are then elaborated to produce the output of the discovery request. This interaction is illustrated in Figure 2 for the case of discovery of ICOs.

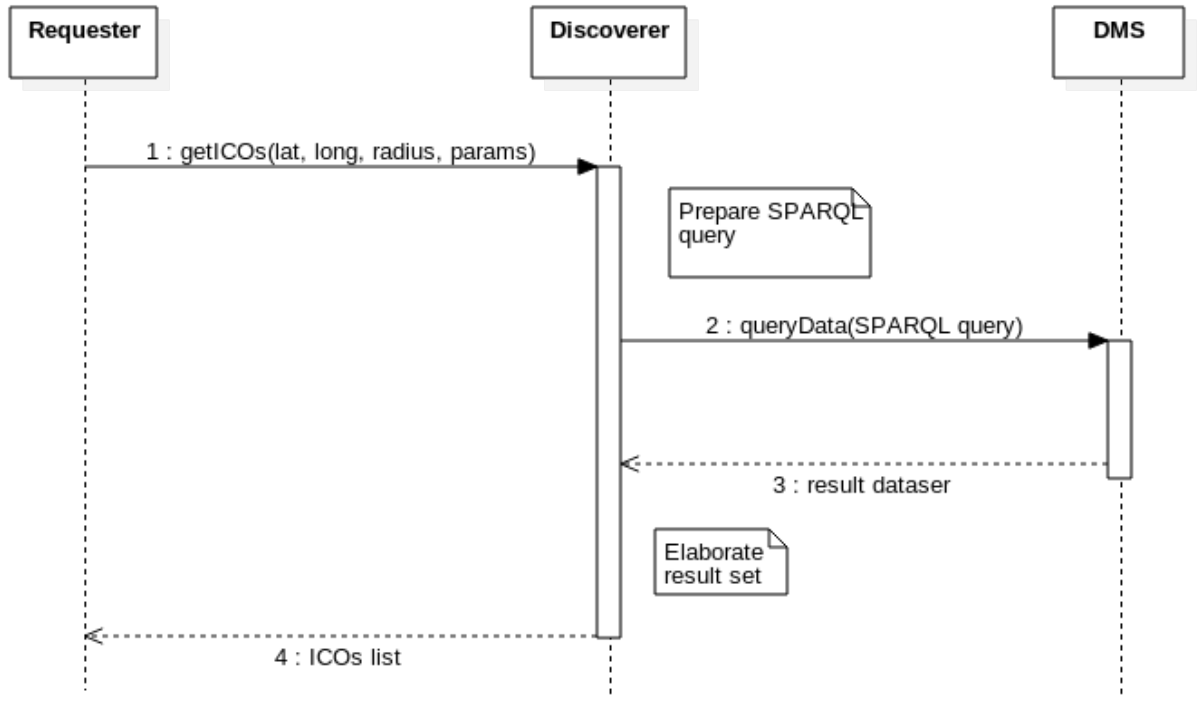

Figure 2. Interaction with DMS

4.2 Implementation of enhanced mechanisms

In order to support the above functionalities, the device description is enriched with the following properties that are defined in Deliverable [Vital-D3.1.1-2014]:

- `hasMovementPattern`, a mandatory property that links to an instance of MovementPattern;

- `hasLocalizer`, an optional property that links to an IoT service specification that provides access to the current location of the sensor.
- `hasNetworkConnection`, an optional property that links to an instance of NetworkConnection.

The property of the ICO hasMovementPattern gives information related to the motion of a device. In the basic version of the Discoverer module, the ICO movement pattern is defined as `Stationary`, indicating that the ICO is not moving at all and the device can be characterized with information about its location (`hasLastKnownLocation`). If this property is not defined, there is no information about the sensor location.

In a more "advanced" version of the Discoverer module, the mobility of a device can be defined by two possible types of `MovementPattern`**:** Predictive and Random. The first provides additional information that allows future movement and locations to be "predicted" (e.g. by defining the direction and the velocity). Knowledge of movement details can be exploited to deduce the area where a moving ICO can be placed in a future moment in time.

Second version of Discoverer takes into account ICOs with predicted mobility pattern to perform a more dynamic discovery. It performs position estimation for nodes with Predicted pattern considering a specified time window. Depending on the result of such a prediction it is possible di identify three different ICO types:

Prediction result	***Description***
Entering ICOs	Connected object that at request time are outside the requested area but, within the considered time window, are likely to move inside the region of interest
Exiting ICOs	Connected object that at request time are inside the requested area but, within the considered time window, are likely to move outside the region of interest
Persisting ICOs	Connected that are currently in the requested area and that, within the considered time window, are likely to remain in the same region.

Following the classification of nodes by their predicted position, we can define the structure of result set considering the mobility of nodes. Subsequently to a discovery request for ICOs, the result set will be composed of the following elements:

- `Stationary` connected objects residing in the requested region
- `Predicted` connected objects belonging to prediction category "*Entering ICOs"* and "*Persisting ICOs*".
- `Mobile` connected objects residing in the requested area

The management of general `Mobile` ICOs to be handled in a more dynamic way during the discovery process could be introduced in a following version of the Discoverer.

In Deliverable [Vital-D3.1.1-2014], we specified that a mobile sensor in a system that supports a localization service should include the property **hasLocalizer**. The localizer service can then rely on information transmitted by the sensor itself (e.g. through a GPS receiver), or alternatively it can use an external tracking system. In

either case, the Discovery service will provide a list of objects that support the localizer service.

In this respect, the VITAL concept moves beyond the Internet of Things by referencing Cloud computing associated with the Internet of Things; we refer to this as “the Cloud of Things” (CoT) [PLM+14] [PLM2+14].

4.3 Web Service definition

4.3.1 Overview

Deliverable [Vital-D3.1.1-2014] provides information on the data models and ontologies used by VITAL to describe the sensor/ICO.

In the short example in Table 5 we can observe the parameters used to represent a `VitalSensor`, i.e. name, type, URI and description. Meaningful information used by our ICOs Discoverer, such as the location and the observed properties, is stored in this last container.

Table 5. Example of a VITAL sensor

```
{
  "@context": "http://vital-iot.org/contexts/sensor.jsonld",

  "name": "TemperatureSensor No.123",
  "type": "VitalSensor",
  "description": "This is an example sensor",
  "uri": "http://www.example.com/vital/sensor/123",
  "hasLastKnownLocation": {
    "type": "geo:Point",
    "geo:lat": "53.2719",
    "geo:long": "-9.0489"
  },
  "ssn:observes": [
    {
      "type": "http://lsm.deri.ie/OpenIoT/Light",
      "uri": "http://www.example.com/vital/sensor/123/light"
    },
    {
      "type": "http://lsm.deri.ie/OpenIoT/Temperature",
      "uri": "http://www.example.com/vital/sensor/123/temperature"
    }
  ],
  "ssn:madeObservation":
    "http://www.example.com/vital/sensor/123/obsvn/1"
}
```

We present below a synopsis of the ideal interfaces that should be supported by the Discoverer in order to satisfy requests from other modules of the architecture (e.g. CEP, Orchestration).

To retrieve ICO information, the Discoverer presents different functions, such as:

- `ConnDMS` (Table 6) is a service used to test if connection between Discoverer and DMS is properly working.
- `getICOs` (Table 9) is a service to discover Internet Connected Objects

`getIoTSystems (`

- Table 11) is a service to discover IoT systems connected to VITAL, as well as VITAL sub-systems

- `getIoTSevices` (Table 13) is a function dedicated to the discovery of available services.

We introduced the service `ConnDMS` to have a mechanism to test the connection status between the Discoverer and the DMS. This is both useful for testing during development and for monitoring purposes. In case of malfunction the response to this service can provide information useful to isolate possible problems.

Table 6. Connection to DMS

	Connection to DMS
Description	It gives information about the status of the connection with the DMS.
URL	BASE_URL/discoverer/ConnDMS
Method	GET
Input	-

Output	{
	"@context": "http://vital-iot.org/contexts/service.jsonld",
	"type": "ServiceDiscovery/ConnDMS",
	"hrest:hasAddress": "BASE_URL/ConnDMS",
	"hrest:hasMethod": "hrest:GET",
	"hrest:status": "OFF "
	}

4.3.2 Discovery of ICOs

The discovery of Internet connected objects is an essential service for other components of VITAL system. This is necessary to have a mechanism able to retrieve in a dynamic and transparent way identifiers for ICOs according to a set of characteristics. The domain of these properties is widely defined to allow different types of searches over the set of available entities registered in the system. The service accepts HTTP POST requests containing a JSON object with all the parameters used to perform the search. Table 7 contains a list of the properties that can be defined in the ICOs discovery request:

Table 7. ICOs discovery request's JSON object

Name	***Type***	***Description***
`type`	String	Represents the category assigned to ICOs
`position`	Object	Object containing keys to perform a discovery over a spatial region
`latitude`	Number	Value of latitude as expressed in WGS84 standard using degree representation
`longitude`	Number	Value of longitude as expressed in WGS84 standard using degree representation
`radius`	Number	Distance expressed in kilometres. Represents the distance from the centre of the region of interest
`observes`	String	Indicates the measuring properties that ICOs have to provide
`movementPattern`	String	Used to select ICOs registered with a

		specific movement pattern
`connestionStability`	String	Allows to select ICOs with a specific level of connection stability
`hasLocalizer`	Boolean	Parameter to identify whether ICOs must provide a localizer service
`timeWindow`	Number	Time expressed in minutes. Identifies the time window taken into account for position estimation (default 15 minutes)

The key `position` has been included to allow the search to focus in a specific spatial region and is optional. The aforementioned key is associated to a JSON object with the triple `latitude`, `longitude` and `radius`. Even though `position` is optional, the triple is mandatory. If a user wants to restrict the search over a specific area, all the elements of the triple must be provided. If one of the element is missing the geographical search can't take place.

To perform this geographical search, it is necessary to compute latitude and longitude boundaries. Given a centre position and a distance in kilometres the conversion process aims to compute minimum and maximum values for both latitude and longitude. This computation is performed using the law of haversines [HF], defined in spherical trigonometry, which relates sides and angles of spherical triangles. This is mainly used in navigation to take into account Earth geometry. The formula can be used to obtain, starting from a couple of coordinates, a distance and a bearing angle, the pair of coordinates of a destination point. Hence, received `latitude` and `longitude` defines the starting point, whereas half `radius` is used as distance. Changing bearing angle, from 0 to 270 degrees clockwise, boundaries are measured. Table 8 summarizes parameters used.

Table 8. Bearing angle for latitude and longitude

Boundary	Bearing angle
Maximum Latitude	0°
Minimum Latitude	180°
Maximum Longitude	90°
Minimum Longitude	270°

A graphical representation of the geometry related to the boundaries definition for spatial search is presented in Figure 3.

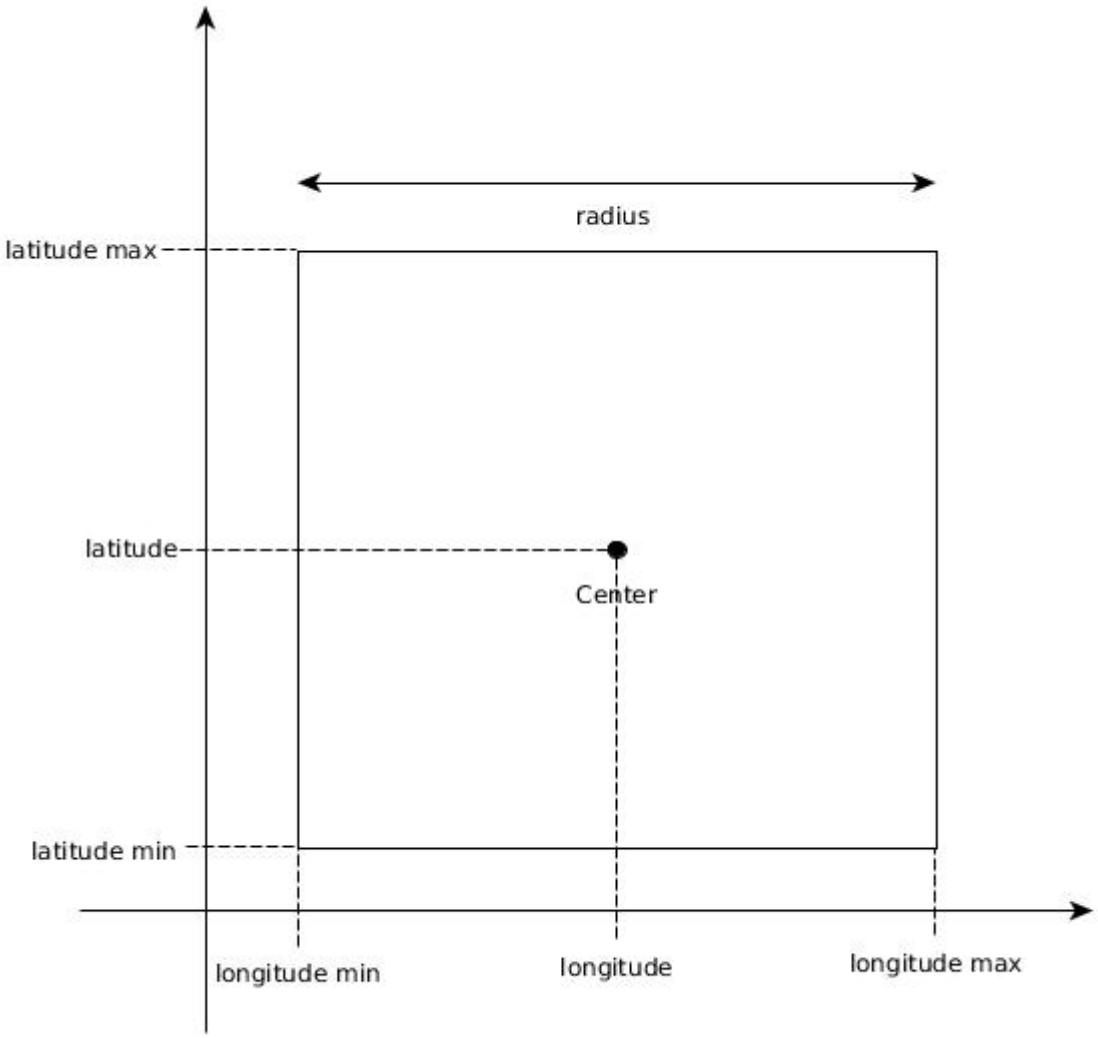

Figure 3. Latitude and longitude boundaries

To support the discovery based on the movement pattern the key `movementPattern` accepts one of the following values:

- `"http://vital-iot.eu/ontology/ns/Stationary"`
- `"http://vital-iot.eu/ontology/ns/Mobile"`
- `"http://vital-iot.eu/ontology/ns/Predicted"`

To support the discovery based on the connection stability the key `connectionStability` accepts one of the following values:

- `"http://vital-iot.eu/ontology/ns/Continuous"`
- `"http://vital-iot.eu/ontology/ns/Intermittent"`
- `"http://vital-iot.eu/ontology/ns/Disconnected"`

To implement prediction on mobile nodes position an optional parameter `timeWindow` is defined. This parameter expresses a time in minutes and, if not otherwise defined, a default value of 15 minutes is taken into account.

A brief recap of input and output for the discovery of internet connected objects in presented in Table 9.

Table 9. Get ICOs

	Get ICOs
Description	Returns the identifiers of the ICOs that match the search criteria.
URL	`DISCOVERY_BASE_URL/ico`
Method	`POST`
Request headers	`Content-Type` `application/json`

Request body	**Example**
	`{`
	`  "position":`
	`  {`
	`  "latitude": 40.83452,`
	`  "longitude": 5.04737,`
	`  "radius": 1`
	`  },`
	`  "observes": "http://lsm.deri.ie/OpenIoT/Temperature",`
	`  "type": "http://vital-iot.eu/ontology/ns/VitalSensor",`
	`  "movementPattern": "http://vital-iot.eu/ontology/ns/Mobile",`
	`  "connectionStability":"http://vital-iot.eu/ontology/ns/Continuous",`
	`  "hasLocalizer": true,`
	`  "timeWindow": 10`
	`  }`

Response headers	`Content-Type` `application/ld+json` or `application/json`

Response body	**Example**
	`[`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/sensor.jsonld",`
	`  "uri": "http://www.example.org/vital/sensor/123"`
	`  },`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/sensor.jsonld",`
	`  "uri": "http://www.example.org/vital/sensor/125"`
	`  },`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/sensor.jsonld",`
	`  "uri": "http://www.example.org/vital/sensor/127"`
	`  }`
	`]`

Notes	• Request body is a JSON object. The object should contain one or more keys defined in Table 7. Empty objects are not accepted.

4.3.3 Discovery of Systems

The discovery of registered systems is a service that produces an output containing a list of URIs for systems registered in VITAL. The request receives a JSON object containing keys to specify the discovery parameters.

Table 10 contains a list of the properties that can be defined in the request for systems discovery.

Table 10. Systems discovery request's JSON object

Name	*Type*	*Description*
`type`	String	Represents the category assigned to the systems when registered to VITAL
`serviceArea`	String	The field is optional and can be used to detail the service area the system has been registered with

A brief recap of input and output for the discovery of systems in presented in Table 11.

Table 11. Get IoT Systems

	Get IoT systems
Description	Returns the identifiers of the IoT Systems that match the search criteria.
URL	`DISCOVERER_BASE_URL/system`
Method	`POST`
Request headers	`Content-Type` `application/json`

Request body	**Example**
	`{`
	`  "type": "http://vital-iot.eu/ontology/ns/VitalSystem"`
	`  "serviceArea": "http://example.com/service_area_1"`
	`}`

Response headers	`Content-Type` `application/ld+json` or `application/json`

Response body	**Example**
	`[`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/system.jsonld",`
	`  "uri": "http://www.example.org/vital/system/sys1"`
	`  },`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/system.jsonld",`
	`  "uri": "http://www.example.org/vital/system/sys2"`
	`  },`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/system.jsonld",`
	`  "uri": "http://www.example.org/vital/system/sys3"`
	`  }`
	`]`

Notes	• Request body is a JSON object. The object should contain one or more keys defined in • Table 10. Empty objects are not accepted.

4.3.4 Discovery of Services

The discovery of services registered in VITAL system if performed though the Discoverer function here described. The purpose is to provide a collection of identifying URIs for registered services, according to the parameters defined in the request.

Table 12. Discoverer request's JSON object

Name	***Type***	***Description***
`type`	String	Represents the category assigned to the service when registered to VITAL
`system`	String	An optional key to specify the URI of the system providing the service

A brief recap of input and output for the discovery of services in presented in Table 13

Table 13. Get IoT Services

	Get IoT services	
Description	Produces a list of identifiers of the services that match the search criteria.	
URL	`DISCOVERER_BASE_URL/service`	
Method	`POST`	
Request headers	`Content-Type`	`application/json`

Request body	**Example**
	
	`{`
	`  "type": "http://vital-iot.eu/ontology/ns/VitalService"`
	`  "system": "http://example.com"`
	`}`	

Response headers	`Content-Type`	`application/ld+json` or `application/json`

Response body	**Example**
	
	`[`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/service.jsonld",`
	`  "uri": http://www.example.org/service/service1`
	`  },`
	`  {`
	`  "@context":"http://vital-iot.org/contexts/service.jsonld",`
	`  "uri": "http://www.example.org/service/service2"`
	`  },`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/service.jsonld",`
	`  "uri": http://www.example.org/service/service3`
	`  },`
	`  {`
	`  "@context": "http://vital-iot.org/contexts/service.jsonld",`
	`  "uri": "http://www.example.org/service/service4"`
	`  }`
	`]`	

Notes	• Request body is a JSON object. The object should contain one or more keys defined in Table 12. Empty objects are not accepted.

5 INTERACTION WITH OTHER VITAL COMPONENTS

In this section we will introduce and provide description of other services, defined in VITAL architecture, which interact with Discoverer. The involved services are Data Management Service (DMS), Complex Event Processing (CEP) and Orchestration, each detailed in the dedicated following paragraphs.

5.1 Interfaces to Data Management Service

Discoverer uses two different interfaces offered by Data Management Service (DMS) to access/get metadata and observations. Table 14 provides an example of the metadata interface. This interface can be used in order to obtain metadata (e.g. of an IoT system) stored in DMS. The second interface is the SPARQL endpoint that the discovery module can use to query observations stored in DMS. An illustration of this interface is provided in Table 14.

Table 14. DMS – Access metadata

	Pull metadata from DMS	
Description	This interface can be used to retrieve metadata from DMS.	
Method	POST	
URL	DMS_BASE_URL/metadata	
Request headers	Content-Type	application/json

Request body	**Example**
	`{`
	`  "id":"",`
	`  "returnType" : "JSON-LD"`
	`}`	

Response headers	Content-Type	application/json or application/ld+json or application/rdf+xml

Response body	**Example**
	`{`
	`  "@context": "http://vital-iot.eu/contexts/system.jsonld",`
	`  "id": "http://example.com",`
	`  "type": "vital:VitalSystem",`
	`  "name": "Sample IoT system",`
	`  "description": "This is a VITAL compliant IoT system.",`
	`  "operator": "http://example.com/people#john_doe",`
	`  "serviceArea": "http://dbpedia.org/page/Camden_Town",`
	`  "sensors":`
	`  [`
	`  "http://example.com/sensor/1",`
	`  "http://example.com/sensor/2"`
	`  ],`
	`  "services":`
	`  [`
	`  "http://example.com/service/1",`
	`  "http://example.com/service/2",`	

	<pre>"http://example.com/service/3"
	],
	"status": "vital:Running"
	}</pre>

Notes	• The request body is a JSON object with the following fields: **id**, whose value is the URI of an entity (of an IoT system, an ICO, or an IoT service), and **returnType**, which is a flag whose value is RDF/XML, JSON or JSON-LD.

Table 15. DMS – Access measurements

	Pull data from DMS	
Description	**This interface can be used to pull data from DMS using the SPARQL query language.**	
Method	**POST**	
URL	**DMS_BASE_URL/sparql**	
Request headers	Content-Type	application/json

Request body	**Example**
	<pre>{
	"query": " prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> SELECT ?uri WHERE { ?uri rdf:type <http://vital-iot.eu/ontology/ns/IotSystem> } ",
	"returnType": "JSON-LD"
	}</pre>	

Response headers	Content-Type	application/json or application/ld+json or application/rdf+xml

Response body	**Example**
	<pre>{
	"@context": "http://vital-iot.eu/contexts/system.jsonld",
	"id": "http://example.com",
	"type": "vital:VitalSystem",
	"name": "Sample IoT system",
	"description": "This is a VITAL compliant IoT system.",
	"operator": "http://example.com/people#john_doe",
	"serviceArea": "http://dbpedia.org/page/Camden_Town",
	"sensors":
	[
	"http://example.com/sensor/1",
	"http://example.com/sensor/2"
	],
	"services":
	[
	"http://example.com/service/1",
	"http://example.com/service/2",
	"http://example.com/service/3"
	],
	"status": "vital:Running"
	}</pre>	

Notes	• The request body is a JSON object with the following fields: **query**, whose value is a SPARQL query, and **returnType**, which us a flag whose value is either RDF/XML, JSON or JSON-LD.	

Additional details regarding the above interfaces can be found in deliverable D3.2.2.

5.2 Interfaces to CEP

The CEP module may use the Discoverer in the future to provide an adaptive mechanism to find different sources for the filtering service and CEPICOs.

This interaction could be used in order to provide enhanced CEP functionality, for instance if a data source or ICO suddenly stop providing data, the CEP could query to the Discoverer in order to find a new source of data which could provide the information and avoid service unavailability in VITAL platform. The operations provided by the Discoverer that will be used by CEP are Get IoT Systems in section 4.3.3 and Get ICOs in section 4.3.2.

5.3 Interfaces to Orchestration

The goal of VITAL's Orchestration is to achieve cross-platform and cross-business-context process integration, supporting VITAL's overall goal of providing an abstract digital layer over the application silos of the modern smart city. To this end, the Orchestrator module allows for definition of higher-level services that combine multiple underlying services and processes (as homogenized by the VITAL platform) and these can be called by the applications.

5.4 Interfaces to Management Platform

The VITAL platform is a complex IoT system that consists of multiple loosely coupled subcomponents (via RESTful APIs), and it by design integrates with multiple and heterogeneous underlying IoT Platforms / Products. VITAL provides to its connected applications a common view of the data & services provided by these underlying systems, unified by a common semantic data model. In a similar fashion the VITAL management layer provides a common management plane that unifies all the management information of all components and systems as well as management functionality that can be performed in a unified way to all parts of the VITAL ecosystem [Vital-D5.1.2.-2015].

The management platform, depends on the Service Discovery module for it's Dynamic Topology functionality. With its support for finding existing systems, services and sensors, the Management Platform can dynamically fill and display the dashboard view with the necessary topology information.

5.5 Interfaces to Development Tools

The Development and Deployment Environment serves as a single entry point to the added-value (filtering, complex event processing, and business process management) and management functionalities provided by VITAL. Although Service Discovery resides at a lower level, the development tools expose all discovery

functionalities to the application users, by allowing them to use directly the VUAIs that the Service Discovery component offers.

6 CONCLUSION AND NEXT STEPS

In the final release of this document, we have introduced the Discoverer module in the VITAL architecture. We have positioned it within the global architecture, presented the requirements and expectations of the module and explained our reasoning for the technical implementation choices we have made.

This document also describes the main interfaces, the current status of the service, and the interaction between Discoverer and other components of the system.

We have chosen an iterative implementation, progressively adding more services and allowing the discovery of more item types based on classification of ICOs, flows and services. In order to enhance the service that can be provided by the VITAL framework, the Discoverer will embed different predictive model mechanisms, miming approaches such as the ones in [BJR+08], [MTBSS+12], and [MSTS+13].

We defined a set of operations to provide the discovery of ICOs, Services and Systems. In the discovery of ICOs we introduce a set of input parameters that allow to refine the search through a variety of options. This includes, but is not restricted to, the discovery over a spatial region. Moreover, in the discovery of ICOs, we proposed a solution to include in the result set nodes based on their mobility.

Discoverer interacts with other components of VITAL system, such as Orchestration and CEP as a support for their needs. Hence, the proposed component acts as a waypoint for all the requests, coming from previously mentioned units, using DMS functionalities to retrieve information and produce a result.

7 REFERENCES

[BJR+08] G. Box, G.M. Jenkins, and G.C. Reinsel, “Time Series Analysis: Forecasting and Control”, fourth edition, Wiley, 2008.

[HF] Haversine formula, https://en.wikipedia.org/wiki/Haversine_formula, last visited 30 July 2015

[MTBSS+12] M. Marchini, M. Tortonesi, G. Benincasa, N. Suri, C. Stefanelli, “Prediction Peer Interactions for Opportunistic Information Dissemination Protocols”, in IEEE Symposium on Computers and Communications (ISCC), July 2012.

[MSTS+13] A. Morelli, C. Stefanelli, M. Tortonesi, N. Suri, “Mobility Pattern prediction to Support Opportunistic Networking in Smart Cities”, in International Conference on Mobile Wireless Middleware, Operating Systems and Applications (MOBILWARE), November 2013.

[PLM+14] R. Petrolo, V. Loscrí, N. Mitton, “Towards a Smart City based on Cloud of Things,” in ACM International MobiHoc Workshop on Wireless and Mobile Technologies for Smart Cities (WiMobCity), August 2014.

[PLM2+14] R. Petrolo, V. Loscrí, N. Mitton, “Towards a Smart City based on Cloud of Things,” in IEEE Multimedia Communications Technical Committee (MMTC), vol 9., Number 5, Special Issue on: Technologies, Services and Applications for Smart Cities, September 2014.

[RFielding00] R. Thomas Fielding, “Architectural Styles and the Design of Network-Based Software Architectures”, Ph.D. Dissertation, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[Vital-2.2-2014] J. Soldatos, J. Kaldis et al. Vital Project, Deliverable D2.2, “Reference and Validation Scenarios for IoT virtualization”, April 2014.

[Vital-D2.3-2014] J. Soldatos, J. Kaldis et al. Vital Project, Deliverable D2.3, “Virtualization Architecture and Technical Specifications”, June 2014.

[Vital-D3.1.1-2014] G. Schiele, T. Geraghty et al. Vital Project, Deliverable D3.1.1, “Virtual Models, Data and Metadata for ICOs V1”, September 2014.

[Vital-D3.2.2-2015] J. Soldatos, K. Roukounaki et al. Vital Project, Deliverable D3.2.2, “Specification and Implementation of Virtualized Unified Access Interfaces V2”, June 2015

[Vital-D5.1.2-2015] A.Lenis, F. Stamatelopoulos et al. Vital Project, Deliverable D5.1.2, “D5.1.2 Management services over federated IoT platforms V2”, June 2015

