Project Acronym:	**VICINITY**
Project Full Title:	**Open virtual neighbourhood network to connect intelligent buildings and smart objects**
Grant Agreement:	**688467**
Project Duration:	**48 months (01/01/2016 - 31/12/2019)**

Deliverable D6.3

Auto-Discovery space deployment validation report

Work Package:	WP6 – VICINITY Framework Integration & Lab Testing
Task(s):	T6.3 – VICINITY auto-discovery space deployment & validation
Lead Beneficiary:	IS
Due Date:	31st December 2018 (M36)
Submission Date:	21st December 2018 (M36)
Deliverable Status:	Final
Deliverable Type:	R
Dissemination Level:	PU
File Name:	VICINITY_D6.3_Auto-Discovery space deployment validation report_v1.0.pdf

This project has received funding from the European Union's Horizon 2020 Research and innovation programme under Grant Agreement n°688467

VICINITY Consortium

No	Beneficiary		Country
1.	TU Kaiserslautern (Coordinator)	UNIKL	Germany
2.	ATOS SPAIN SA	ATOS	Spain
3.	Centre for Research and Technology Hellas	CERTH	Greece
4.	Aalborg University	AAU	Denmark
5.	GORENJE GOSPODINJSKI APARATI D.D.	GRN	Slovenia
6.	Hellenic Telecommunications Organization S.A.	OTE	Greece
7.	bAvenir s.r.o.	BVR	Slovakia
8.	Climate Associates Ltd	CAL	United Kingdom
9.	InterSoft A.S.	IS	Slovakia
10.	Universidad Politécnica de Madrid	UPM	Spain
11.	Gnomon Informatics S.A.	GNOMON	Greece
12.	Tiny Mesh AS	TINYM	Norway
13.	HAFENSTROM AS	HITS	Norway
14.	Enercoutim – Associação Empresarial de Energia Solar de Alcoutim	ENERC	Portugal
15.	Municipality of Pylaia-Hortiatis	MPH	Greece

[1] ***Deliverable Type:***
R: Document, report (excluding the periodic and final reports)
DEM: Demonstrator, pilot, prototype, plan designs
DEC: Websites, patents filing, press & media actions, videos, etc.
OTHER: Software, technical diagram, etc.

[2] ***Dissemination level:***
PU: Public, fully open, e.g. web
CO: Confidential, restricted under conditions set out in Model Grant Agreement
CI: Classified, information as referred to in Commission Decision 2001/844/EC.

Authors List

Leading Author (Editor)				
Surname		**First Name**	**Beneficiary**	**Contact email**
Paralič		Marek	IS	marek.paralic@intersoft.sk
Co-authors (in alphabetic order)				
No	**Surname**	**First Name**	**Beneficiary**	**Contact email**
1.	Bednár	Peter	IS	peter.bednar@intersoft.sk
2.	Guan	Yajuan	AAU	ygu@et.aau.dk

Reviewers List

List of Reviewers (in alphabetic order)				
No	**Surname**	**First Name**	**Beneficiary**	**Contact email**
1.	Cimmino	Andrea	UPM	cimmino@fi.upm.es
2.	Koutli	Mary	CERTH	mkoutli@iti.gr
3.	Zivkovic	Carna	UNIKL	zivkovic@cs.uni-kl.de

Revision Control

Version	Date	Status	Modifications made by
0.1	31 May 2018 (M29)	Initial Draft	Paralič (IS)
0.2	20 July 2018 (M31)	Draft version of brief test cases description	Paralič (IS)
0.3	24 September 2018 (M33)	Draft description of test case 1	Bednár (IS)
0.4	4 October 2018 (M24)	Draft description of test cases 2 and 3	Bednár (IS)
0.5	19 November 2018 (M35)	Results of test case 2	Paralič (IS)
0.6	30 November 2018 (M35)	Results of test case 1	Paralič (IS)
0.7	3 December 2018 (M36)	Introduction, Approach	Guan (AAU), Paralič (IS)
0.8	9 December 2018 (M36)	Results of test case 3	Bednár (IS)
0.9	10 December 2018 (M36)	Consolidated version of D6.3	Paralič (IS), Bednár (IS)
0.9.1	11 December 2018 (M36)	First version for QAR	Paralič (IS), Bednár (IS)
0.9.2	20 December 2018 (M36)	Final Draft reviewed	Paralič (IS), Bednár (IS)
1.0	21 December 2018 (M36)	Submission to the EC	Zivkovic (UNIKL)

Executive Summary

The presented document is the deliverable "D6.3 – Auto-Discovery space deployment validation report" of the VICINITY project (Grant Agreement No.: 688467), funded by the European Commission's Directorate-General for Research and Innovation (DG RTD), under its Horizon 2020 Research and Innovation Programme (H2020). This deliverable contributes to reach Milestone 7 (MS7 - First integrated system prototype available) by validation of the Auto-Discovery platform deployment within the task T6.3.

The document presents the process, the results and the quality and performance feedback for the validation of the auto-discovery functionality of the VICINITY platform. The key enabler for achieving smooth integration of heterogeneous IoT systems, platforms, and devices is the interoperability at the semantic level. In the VICINITY platform the semantic interoperability relies on the VICINITY Agents at the client/node side and the Semantic Discovery and the Agent Configuration Platform (SDACP) at the server/cloud side. SDACP in turn consists of a semantic triplestore and the service that provides the API for manipulating IoT object descriptions stored in the triplestore.

During the validation process we proved that:

- Auto-discovery process of the devices described in the VICINITY ontology is fully functional and accessible for the IoT platform adapters connected to the VICINITY Agent.
- Current implementation of the SDACP platform uses a Semantic graph database GraphDB[1] for storing semantic triplestores. The GraphDB is able to manage the appropriate level of load generated by dozens of IoT objects per adapter and dozens of adapters per one VICINITY Agent.
- The core functionality of the Agent, which is responsible for updating the semantic repository, can perform more than 10.000 DIFF statements per second and more than 15.000 IoT objects per second.
- The Semantic repository that is available via the Semantic discovery, and configuration service, offers an average processing time for one insert operation of 0,09s, for one update operation 0,79s, and one delete operation 0.02s[2]. However, the current development and testing installation of SDACP is based on the free licensed version of the GraphDB store that is limited to two parallel client sessions. This setup can dramatically reduce the scalability, which can be further enhanced by unlimited commercial version of GraphDB.

This deliverable covers all individual modules that have been developed in WP3 - "VICINITY Server Implementation" and WP4 - "VICINITY Client Infrastructures Implementation", with focus on the server and client side discovery components designed and implemented in the tasks T3.2 and T4.2. Problems identified are timely reported and solved.

The quality and performance feedback of this task's tests is used in order to improve the auto-discovery functionality – its availability and ability to be also deployed to potential middle and large-scale IoT platforms.

[1] http://graphdb.ontotext.com/

[2] More details including the relation to the number of IoT objects can be found in section 5.4.

Table of Contents

List of Tables

List of Figures

List of Definitions & Abbreviations

Abbreviation	Definition
EC	European Commission
CFP	Call For Paper
D.A.R.	Dissemination Activity Report
DC	Direct Current
DG RTD	Directorate-General for Research and Innovation
DIFF	Difference (used by Agent's state update operation)
DoA	Description of Actions
EU	European Union
H2020	Horizon 2020 Research and Innovation Programme
H-EMS	Home Energy Management System
IoT	Internet of Things
KPIs	key performance indicators
NM	Neighbourhood Manager
OPGW	Optimal Power Ground Wire
P2P	Peer to Peer
PMU	Phasor Measurement Units
SDACP	Semantic Discovery and Agent Configuration Platform
TD	Thing Description
WP	Work-Package

1. Introduction

The deliverable describes the validation process and the results for the Auto-Discovery platform deployment. Auto-discovery platform includes the Semantic Discovery and Agent Configuration Platform at the server side, as well as, the VICINITY Agent at the client side of the VICINITY platform. Validation process relies on data provided by the all the pilot cases, which were also used during the integration testing in the task T6.2 "Lab setup, Testing & Validation". It includes full functional test of the VICINITY platform discovery process initiated by a locally deployed VICINITY node connected to the VICINITY development environment[3], performance of the Agent's DIFF operation and the write/delete/query over the semantic model performance. Functional test of the discovery process includes also scenarios with unusual circumstances in order to detect performance limits of the current VICINITY prototype as it is described in the deliverable D6.2[4].

1.1. Context within VICINITY

Figure 1 gives an overview of the context of D6.3 within VICINITY. D6.3 together with D6.1 and D6.2 contributes to reach Milestone 7 (MS7). MS7 - First integrated system prototype available - marks the conduction of intensive integrated Lab testing for VICINITY prototype, with the use of the VICINITY server components/services, client infrastructures and value-added services that were made available by the previous milestones.

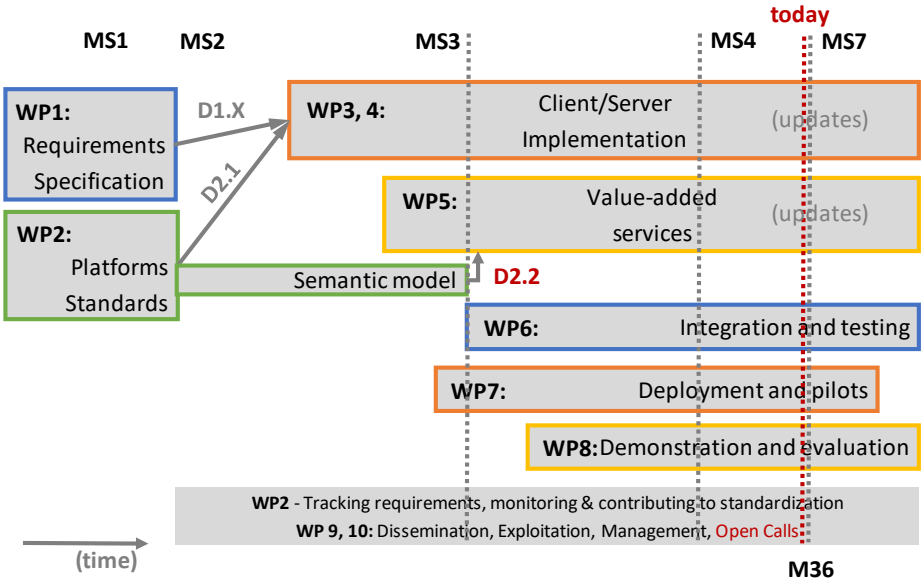

Figure 1 Work Package Architecture

Regarding the relation to other WPs, the current document builds on the results of previous WPs and tasks, specifically:

[3] Development environment is used specifically for testing, while production environment is used for deployment purposes.

[4] VICINITY test-bed deployment, including Validation, Parameterization and Testing

- WP1 – VICINITY Concept Requirements, Barriers, Specification, and Architecture
- WP3 – VICINITY Server Implementation (Task 3.2)
- WP4 – VICINITY Client Infrastructures Implementation (Task 4.2)
- WP5 – Value-Added Services implementation

The outcome of this deliverable will form the basis of work for the following WPs and tasks:

- WP3 - VICINITY Operation and continuous upgrades of core components (Task 3.3)
- WP4 - VICINITY Clients Operation and continuous upgrades (Task 4.4)
- WP7 – On-site Deployment and Pilot Installations (Tasks 7.2 – 7.5)
- WP8 – Pilot Demonstration and Overall Evaluation (Tasks 8.2 – 8.5)

1.2. Objectives in Work Package 6 and Task 6.3

The purpose of WP6 "VICINITY Framework Integration and Lab Testing" is to ensure that the VICINITY platform operates correctly from a technical perspective, prior to its deployment at the pilot sites in WP7.

T6.1 "Integration of VICINITY Components" focusses on integrating the components that form server and client infrastructures, along with the related value-added services to form the first version of the VICINITY prototype. The layout and scope of the tests in T6.1 were decided based on pilot site definitions, functional requirements, operational requirements and the VICINITY architecture as defined by WP1 "Requirements Specification" and the value-added services as defined by WP5 "Value-Added Services Implementation". The issues that were uncovered during the process are documented in the Issue Tracking System that is used for this project-the Open Project- which is available to all partners, with the status and context of individual issues. Evidence of the progress in solving these issues with cross-pilot cooperation can also be found on the internal project website. Resolved issues resulted in new versions of the software components, which were deployed after successful regression testing.

T6.2 "Lab setup, Testing & Validation" deals with two kinds of lab-testing. The first is Edge Cases Testing to validate the expected prototype performance when the prototype is pushed close to its edges/limits according to the requirements detailed in WP1. The second kind of lab testing focuses on functionality and performance, including cross-domain testing scenarios, in line with value-added services defined in WP5. The diagnosed problems during the lab-testing process were discussed and resolved by collaboration among partners to improve and enrich the VICINITY prototype functionality.

T6.3 "Auto-discovery space deployment and validation" validates the auto-discovery platform and gives the quality and performance feedback prior to deployment at the pilot sites. The auto-discovery platform includes the VICINITY Agent at the client side, as well as the Semantic discovery and dynamic configuration platform at the server side. The platform allows to perform the full semantic search of object descriptions, so that existing objects can be easily discovered. IoT objects descriptions are used as the core information for auto configuration of agents, through which these objects are available for interaction. The validation was split into three main parts:

a. Functional test of the auto-discovery process provided by VICINITY Agent
b. DIFF operation and the write/delete/query over the semantic model performance
c. Performance test of the core VICINITY Agent task

1.3. Structure of the Deliverable

Chapter 1: Introduction to the deliverable, and the context of the tasks in Vicinity. This section outlines the role this document plays in the development process.

Chapter 2: Overall approach to testing.

Chapter 3: Test case 1 - Functional test of the auto-discovery process provided by VICINITY Agent.

Chapter 4: Test case 2 - DIFF performance.

Chapter 5: Test case 3 - SDACP performance.

Chapter 6: Quality and performance feedback.

Chapter 7: Conclusion.

2. Approach

The test cases that are described in this deliverable come from the internal structure of the auto-discovery platform that consist from cloud part and client side. The cloud part - the semantic discovery and dynamic configuration platform is realized as two separate components - Semantic triplestore and Semantic discovery and configuration services. The client-side component - VICINITY Agent - implements the full discovery and configuration process.

The first test covers a functional test of the whole discovery process with utilization of all core components of the VICINITY platform – including Agent, Gateway, Communication server, Neighbourhood Manager and the Semantic discovery and dynamic configuration platform. The second test focuses on the core functionality of the Agent's discovery process itself – the DIFF operation. It is performed on the current state defined by Adapters and persistent state stored in the semantic store. The third test case then covers the cloud side of the auto-discovery platform and checks the performance of the write/delete/query operations applied on the semantic model.

Based on the Edge Case Testing Methodology described in D6.2 VICINITY test-bed deployment, including Validation, Parametrization and Testing (M36), additional edge testing cases are included - limit of registrations from one adapter (considered also the size of the thing descriptions) and limit of registrations from parallel adapters. They are designed to check some features of Gateway API by considering the requirements and installation specifications envisioned in WP1. This enables to define a stable and proper operating range for the VICINITY platform.

As in the testing process for D6.2, if either the testing results or the design behave unexpectedly, a bug and a trace that leads to it are reported through Open Project, emails, Skype and Slack. Iterative tests have been conducted to verify the solutions and solve the bugs.

The general structure for each testing case mainly includes a test scenario and goal, VICINITY components/functions involved, equipment and testing environments, expected results, test procedure, testing platforms, real results, deviations encountered from expected result and solutions.

2.1. Tested VICINITY Platform configuration

The tested VICINITY Platform configuration consists of the set of software components summarized in the following table (Table 1), which were integrated together and verified by execution of the identified test cases (Section 3 - 5).

Table 1 Tested VICINITY Platform configuration

Components	Version
VICINITY Gateway API	0.6.3.1
VICINITY Agent	0.6.3.1
Neighborhood Manager	0.6.3

2.2. Integration testing coverage

The VICINITY Platform interfaces have been directly and/or indirectly tested by individual testing cases (described in Section 3 - 5) that are summarized in the following Table 2.

Table 2 VICINITY Interface integration test coverage

Name of Interface	Used by	Covered by test
VICINITY Neighbourhood Manager		
Authentication service	VICINITY Communication Node, VICINITY Gateway API, VICINITY Agent	Test1
Neighbourhood discovery service	VICINITY Gateway API Services	Test1
Registry service	VICINITY Communication Server	Test1
Semantic model change notifications	Semantic Platform	Test1
Semantic discovery and dynamic configuration agent platform		
Semantic discovery service	VICINITY Neighbourhood Manager	Test1, Test3
Registry Service	VICINITY Neighbourhood Manager	Test1, Test3
VICINITY Gateway API Services		
VICINITY Communication server	Request/ Response	Test1
VICINITY Communication Server		
Discovery service	VICINITY Gateway API	Test1
VICINITY Node Configuration Service	VICINITY Neighbourhood Manager	Test1
Registry Service	VICINITY Communication Node	Test1
VICINITY Communication Node		
VICINITY Node Configuration Service	VICINITY Communication Server	Test1
Registry Service	VICINITY Communication Server	Test1
VICINITY Gateway API		
Discovery and query service	VICINITY Agent/Adapter	Test1
Consuming service	VICINITY Agent/Adapter	Test1

Name of Interface	Used by	Covered by test
VICINITY Node Configuration Service	VICINITY Agent/Adapter	Test1
Registry Service	VICINITY Agent/Adapter	Test1
VICINITY Agent/ Adapter		
VICINITY Node Configuration Service	VICINITY Gateway API	Test1
Exposing service	VICINITY Gateway API	Test1

2.3. Quality and performance measures

The validation will be based on a defined set of quality and performance measures. Quality is measured by the set of functional tests that can be evaluated by the number of passed tests and coverage of the tested functions. The performance is measured by means of metrics that are focused namely on the scalability, entailing that they cover the overall processing time and consumption of computation resources (memory, CPU).

Measured performance metrics for VICINITY Agent DIFF operation include:

- Number of generated DIFF IoT objects per second
- Number of generated DIFF statements per second
- Memory usage measured as the number of allocated objects of the given type and as the overall size of the allocated heap memory in bytes
- CPU usage measured as the % of the CPU computation time

Measured performance metrics for SDCAP write/delete/query operations over the semantic model include:

- Number of stored, updated or deleted IoT objects per second
- Number of stored, update or deleted triples per second
- Overall processing time for various type of SPARQL queries (normalized to the size of the result sets in the number of IoT objects and triples)
- Memory usage measured as the number of allocated objects of the given type and as the overall size of the allocated heap memory in bytes
- CPU usage measured as the % of the CPU computation time

All metrics were implemented and measured on one machine with the same configuration. Implemented tests also check variations by running tests and measuring metrics multiple times.

3. Test case 1 - Functional test of the auto-discovery process provided by the VICINITY Agent

3.1. Testing objective

The main goal of this test case is to test the integration between all main components of the VICINITY platform, from the VICINITY Agents in the changing environment up to the Neighbourhood Manager and Semantic Discovery and Agent Configuration Platform (SDACP) that manages distributed global state of the platform. All components involved in the process of auto-discovery should contribute to the successful discovery process issued by Agent or Adapters at the client side. The current state of the IoT platform connected to the VICINITY platform via Agent(s) & Adapter(s) should be correctly persisted in the semantic repository inside SDACP. Semantically annotated discovered IoT objects should be also visible via Neighbourhood Manager.

3.2. Testing scenario design

Table 3 Functional test of the auto-discovery process provided by the VICINITY Agent - testing scenario design

Test 1	**Functional test of the auto-discovery process provided by the VICINITY Agent**
Test ID	Test 1_IS
Test name	Functional test of the auto-discovery process provided by the VICINITY Agent
Test scenario and goal	To test how the process of discovery of IoT objects is provided by VICINITY Agent. Guarantee the discovery process has correct results accepted by the VICINITY Neighbourhood Manager (NM).
Performed by	Marek Paralič (IS)
Iterations	The test will be conducted repeatedly in different conditions in relation to the initial state of NM and VICINITY Adapters, which represent connected IoT platforms of individual pilot cases.
Equipment, environments and IoT Infrastructure involved	• Live connectivity to a running VICINITY solution that offers the VICINITY Cloud including the NM and the Semantic Discovery and Agent Configuration Platform (SDACP). • One VICINITY Client Node that contains VICINITY Gateway (GW), VICINITY Agent and VICINITY Adapters. • VICINITY Adapters expose stubbed IoT objects.
Expected results	• Query of NM state after the test returns result that contains expected changes defined by the current state of the exposed IoT objects by the Adapter. • Each of IoT object registered by Adapter is registered also by NM. • Each of by NM previously registered IoT object that is not anymore available is removed from NM state. • Each previously registered IoT object by NM that is still exposed by Adapter is available in NM in appropriate state.
Test procedure	Test procedure for one agent with 10 adapters:

- Adapters with representative Thing Descriptions (TDs) of individual pilot sites IoT objects are started as passive and make available their state to the VICINITY Agent via REST API.
- Agent retrieves current state of the Adapters and saves it for later comparison.
- Agent retrieves current state of the NM.
- Agent compares current state of the Adapters with the state of the NM and computes difference in form of list of CRUD operations over the NM state.
- Agent sends a list of CRUD operations to the NM.
- The NM updates its internal policy state and send updates to the semantic platform SDACP.
- The semantic platform SDACP commits the updates to the persistent storage.
- Current state of the NM is retrieved and compared with the expected state defined by the configuration of started Adapters.

Test procedure for getting the limit of max number of objects in one adapter:

- Configuration file for agent is generated with one adapter set to active discovery mode
- Agent with generated configuration file is started
- Active adapter is started and based on number of objects and template with TD to be used, generates POST request to agent with given number of TDs
- The success or failure of the test is decided based on the status code of the agent's response

Test procedure for getting the limit of max number of adapters issuing parallel requests to the agent:

- Configuration file for an agent is generated with given number of adapters set to active discovery mode
- Agent with generated configuration file is started
- Given number of active adapters with 5 objects are started in separated threads (in 1sec intervals) and generate POST request to the agent with 5 TDs
- Based on the status code of the agent's response for every adapter the success or failure of the test is decided

This test case emulates the changes in the environment where the IoT objects are added or removed, or their properties are changed. All these changes are reflected by the change of the state of the VICINITY adapters, which are then propagated in the platform by intermediate components such as VICINITY Agents, VICINITY Gateway, Neighbourhood Manager, and stored in the SDACP. The most important component in this scenario is the VICINITY Agent which detects the changes in the adapter's configuration, compares them with the current state stored in the platform and generates the updates which are then forwarded to the SDACP. The proposed test case will:

a) Create controlled environment with the set of IoT objects each represented by the stub implementation of the VICINITY adapter,
b) Emulate some structural changes in this environment (i.e. add/remove new adapter, change adapter properties, etc.) and

c) Query the state of the SDACP through the Neighbourhood Manager in order to check if the distributed state of the VICINITY platform was successfully updated.

3.3. Testing Platform

The test case is implemented relaying on a Java application, which setup the stub IoT objects and invoke the local instance of the VICINITY Agent. Agent is connected to the testing VICINITY installation through VICINITY Gateway that runs locally. The testing procedure covers the interaction between the components shown in the following sequence diagram:

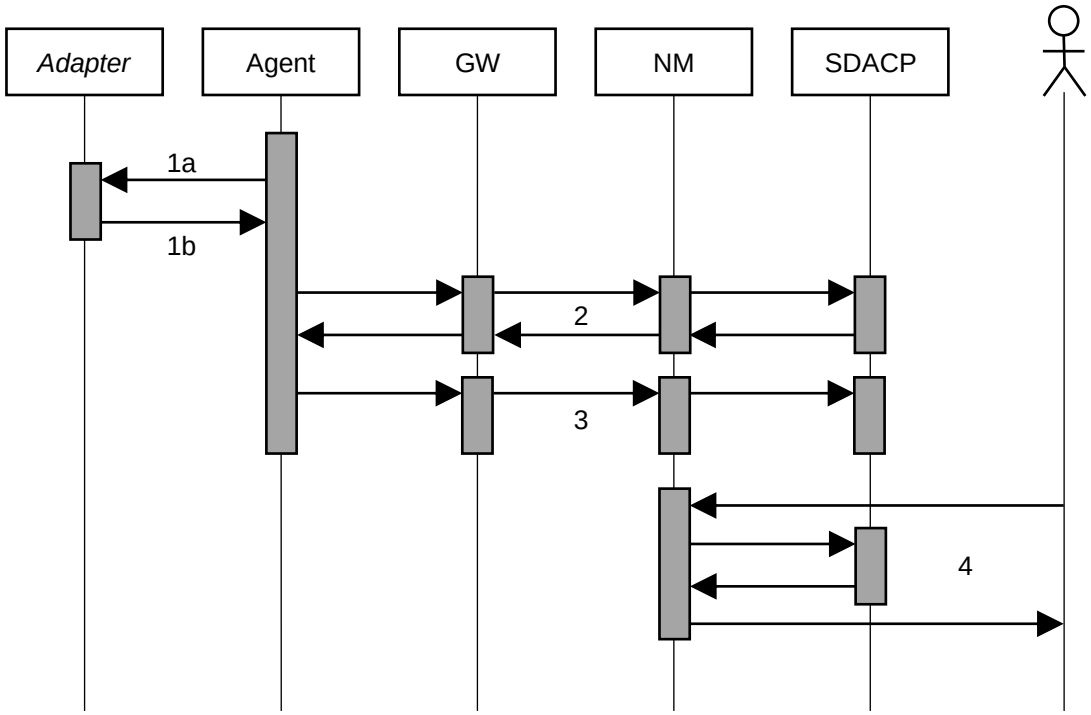

Figure 2 Interaction between the VICINITY components in the functional test of the auto-discovery process

1. Agent queries Adapter state (adapters are implemented as the stubs representing IoT objects and configured in passive mode), the Agent will receive current state of the IoT environment and persist it in json format.
2. Agent queries the current state of the VICINITY platform from the NM, which queries the state from the SDACP. Communication is routed by GW. The Agent compares the current state of the IoT environment and the VICINITY platform and generates the updates.
3. The updated state of the VICINITY platform is propagated through the GW and NM and stored into the SDACP.
4. Client of the platform query the updated state from the NM, which fetch the current state from the SDACP and persists it in JSON format.

Alternatively, the Adapter can initialize the communication asynchronously and notify Agent about the changes of its state (this is denoted as a phase 1b). The rest of the test procedure is unchanged. Note that only the Adapter is implemented as the mock-up that is not connected to the real IoT object. The rest of the components are deployed in the testing cloud environment with the same configuration as the production deployment of the platform.

The following table summarizes the covered interactions, interfaces and operations of the use case visible and used by the test. In the interactions, the first component initializes the communication.

Phase	Interaction	Interfaces	Operations
1a – Query of the adapter state	Agent – Adapter	Adapter REST	GET adapter_URL/objects
1b – Query of the adapter state	Adapter – Agent	Agent REST	POST agent_URL/
4 – Query of update state	Client – NM NM – SDACP	NM REST	GET NM_URL/agents/{agentID}/items

In order to test the limits of the current implementation of the VICINITY platform, the test case 1 was extended in the following way:

- Dynamic generation of Agent's configuration file with defined number of adapters set to active mode
- Dynamic generation of Adapter's configuration file with defined number of objects based on a given TD-template (pilot case specific)
- TD-templates for devices and value-added services specific for individual pilot cases

Tests are run directly from Java IDE, whereby type of test is defined by set of bool parameters of the main test implementation class (`AutoDiscoveryFunctionalTest.java`):

`updateTypeTest` - *if true, suppose previous run with value false - i.e. itemsFromNMwithOIDs.json exists*

`cleanAgentAfterTest` - *if true, delete all items from agent*

`WINDOWS` – if true, the *test is executed under Windows OS*

`registrationLimitsTest` – *if true, just edge case limit testing is executed*

The Test 1 is executed under assumptions that the VICINITY Gateway is up and running locally and the VICINITY Agent is installed and accessible locally. The config file of the agent is not important, as far as the test generates its config file according to the type of the test dynamically.

3.4. Testing Results

Table 4 Functional test of the auto-discovery process provided by the VICINITY Agent - testing results

Test 1	**Functional test of the auto-discovery process provided by the VICINITY Agent**
Real results (figures)	• Agent starts up without failing, and then it successfully registers IoT objects from 10 Adapters in the VICINITY. Adapters are configured in a passive mode and their stub implementations return to Agent between 20 and 30 thing descriptions of IoT objects, i.e., devices or value-added services. The NM is configured to have an empty list of devices for the given Agent. After the test the NM successfully

registers altogether 266 devices and VASs for the given Agent.

- Execution of the given test with non-empty state of the NM for the given Agent confirmed correct execution of the Agent's PATCH operation. If the starting state of the NM was 266 previously registered objects for the given Agent, execution of the test should preserve this state what was checked and confirmed by not-changed OIDs or registered objects.
- The result from series of testing the limits of successful registration of objects for one adapter with relation to the size of their TDs is given on the following Figure 3:

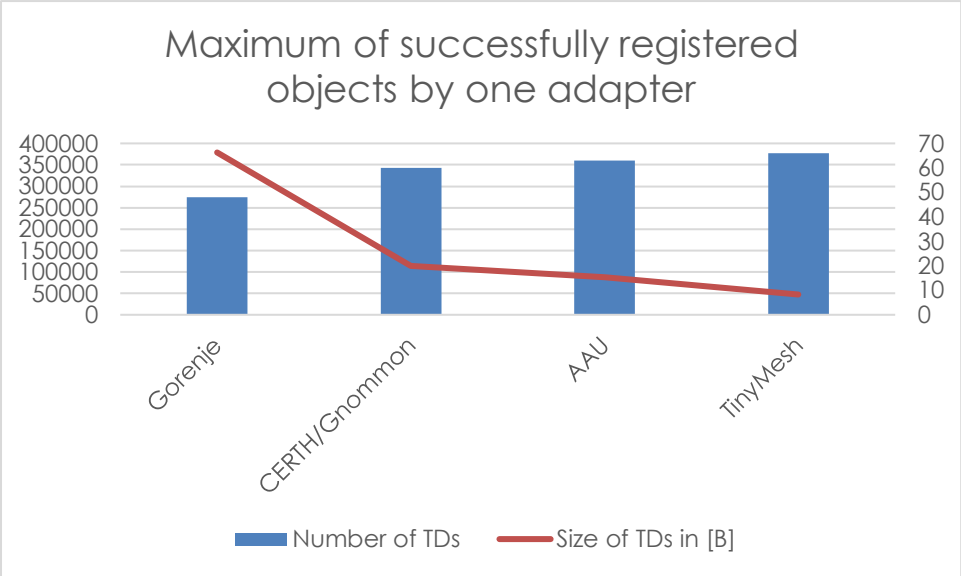

Figure 3 Maximum number of successful registered IoT objects and its relation to the size of their TDs

The tests proved that the current implementation with freely available semantic triplestore in SDACP manages tens of objects from one adapter. If the thing descriptions are a couple of kilobytes – like in the Gorenje devices oven and refrigerator (each 7,5 kB), then the max number was 48. However, if the size decreases to couple of hundreds of bytes – like in case of the door sensor from TinyMesh (700B) – the max number slightly increases to value 66.

- • The series of testing the limits of successful registration of objects (from the VICINITY Agent point of view) for adapters running in parallel gives as result 38 adapters with 5 objects each per one agent. However, this extreme testing case scenario caused problem with storing the results in the VICINITY platform and correct number of registered devices/VASs in the NM that will be further investigated.

Deviations	None.
Other technical issues	Testing scenario with parallel adapters caused problem with storing the results in the VICINITY platform and correct number of registered objects in the NM that will be further investigated.
Status	Passed.

Notes	None.

4. Test case 2 - DIFF performance

4.1. Testing objective

The goal of this test case is to measure the performance and scalability of the main VICINITY agent functionality, i.e. detecting of the changes in the configuration of VICINITY adapters and updating the global state of the platform propagated by the Neighbourhood Manager.

4.2. Testing scenario design

Table 5 DIFF performance - testing scenario design

Test 2	**DIFF performance**
Test ID	Test 2_IS
Test name	DIFF performance
Test scenario and goal	To test the performance of the DIFF operation – IoT objects received from Adapter versus IoT objects got from the NM.
Performed by	Marek Paralič (IS)
Iterations	Scenario is executed in iterations parametrized by the following parameters: • Number of added IoT objects • Number of removed IoT objects • Total number of DIFF statements (including the changes of object parameters)
Equipment, environments and IoT Infrastructure involved	• Stub connectivity to the VICINITY NM • Stub connectivity to one VICINITY Adapter exposing the changes in the configuration of IoT objects
Expected results	DIFF operations have expected performance and scalability measured by the specified measures. Measured performance metrics for different combination of scenario parameters include: • Number of generated DIFF IoT objects per seconds • Number of generated DIFF statements per seconds • Memory and CPU usage
Test procedure	The test procedure consists of the following steps: 1. Initialization of the NM and Adapter stubs 2. For each combination of scenario parameters: a. Prepare expectations based on known state of the NM and the Adapter b. Execute DIFF operation and measure time needed c. Compare results obtained by the DIFF operation with prepared expectations in terms of: i. DELETED objects

ii. CREATED objects
iii. UPDATED objects
iv. UNCHANGED objects

The update of the platform state can be initiated by the VICINITY agent (i.e. after its initialization – for passive adapters) or by an Adapter itself, which can notify the Agent about the changes of its internal state (active adapters). In both cases, the Agent follows the same steps in order to update the global state of the platform depicted in the following sequence diagram:

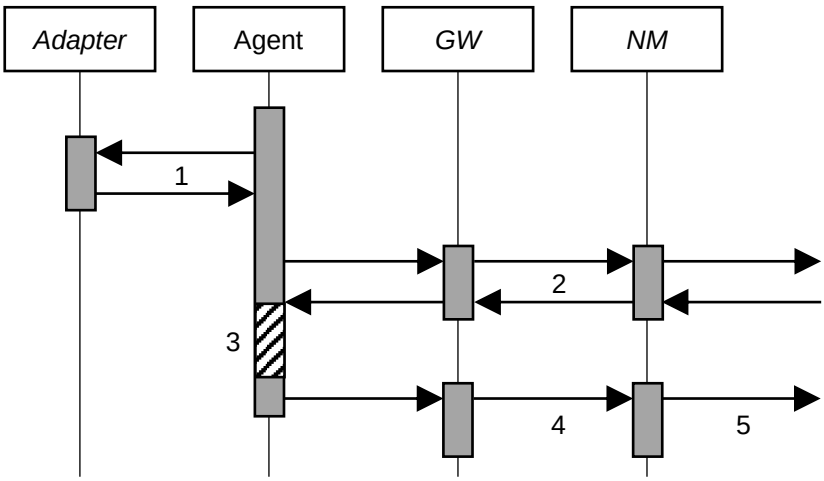

Figure 4 Agent's steps to update the global state of the VICINITY platform

1. The Agent queries the current Adapter state (Agent queries the state of all adapters during the startup. Alternatively, specific Adapter can notify the Agent about the changes sending its current state)
2. The Agent queries the current state of the VICINITY platform from the NM (which subsequently query state from the SDACP where the state is persistently stored)
3. The Agent generates updates of the state. Per each updated adapter, the Agent performs the following operations:
    a. Parse the global platform state and extract the persisted state of the adapter
    b. Compare the current state of the adapter with the persisted and generates the updates
4. The Agent sends detected updates of the global state to the NM through the GW
5. The NM propagates updates to the SDACP persistent storage

This test case measures directly the performance of the VICINITY Agent and it is implemented as a java application, which invokes internally the procedure described in the step 3. Particularly, the communication with the Adapter or with the NM, or querying or updating of the state from the SDACP is not covered by this test. The implementation uses generated inputs for updated states of the adapters and for the actual global state of the platform. Test is scaled by the following parameters:

1. Number of updated statements per one adapter
2. Number of all objects of all adapters of the given access point in the global state

The main performance metrics include:

1. Average time of execution
2. Memory consumption (average capacity, maximum peaks, garbage collector activity)

The results of the test are presented in the graphs shown below, which depicts the dependency between the test parameters (i.e. number of changes and size of the global state) and measured metrics (time and memory consumption). The expected scalability is linear with both parameters.

4.3. Testing Platform

As far as the test case focuses on the core functionality of the VICINITY Agent, the testing platform only consists of the VICINITY agent version 0.6.3.1 and the code of the test itself. In the test scenario the communication with the NM is stubbed as well as with the Adapter. The test is implemented as a java application and deployed as jar file including the library of the Agent. Command to run the test is the following:

```
java -Xmx6000m -jar diff-test.jar 300000 150000 150000 150000 > /dev/null
```

The first parameter after `diff-test.jar` defines the number of IoT objects in the current state of the NM. The following parameters define the current state of the adapter – so number of IoT objects to be created, updated and deleted, resp. Parameter `-Xmx` increases the heap size of the running java application – in order to manage creation of serialized form of the NM state (e.g. in this example the json version of 300000 thing descriptions).

Logs with the results of the test are automatically saved into the defined file (`DiffTestsLogs.txt`), so the standard output could be redirected in to `/dev/null` in order to shorten the test duration.

4.4. Testing Results

Table 6 DIFF performance – testing results

Test 2	**DIFF performance**
Real results (figures)	Charts showing the dependencies among the performance measures and the scenario parameters.

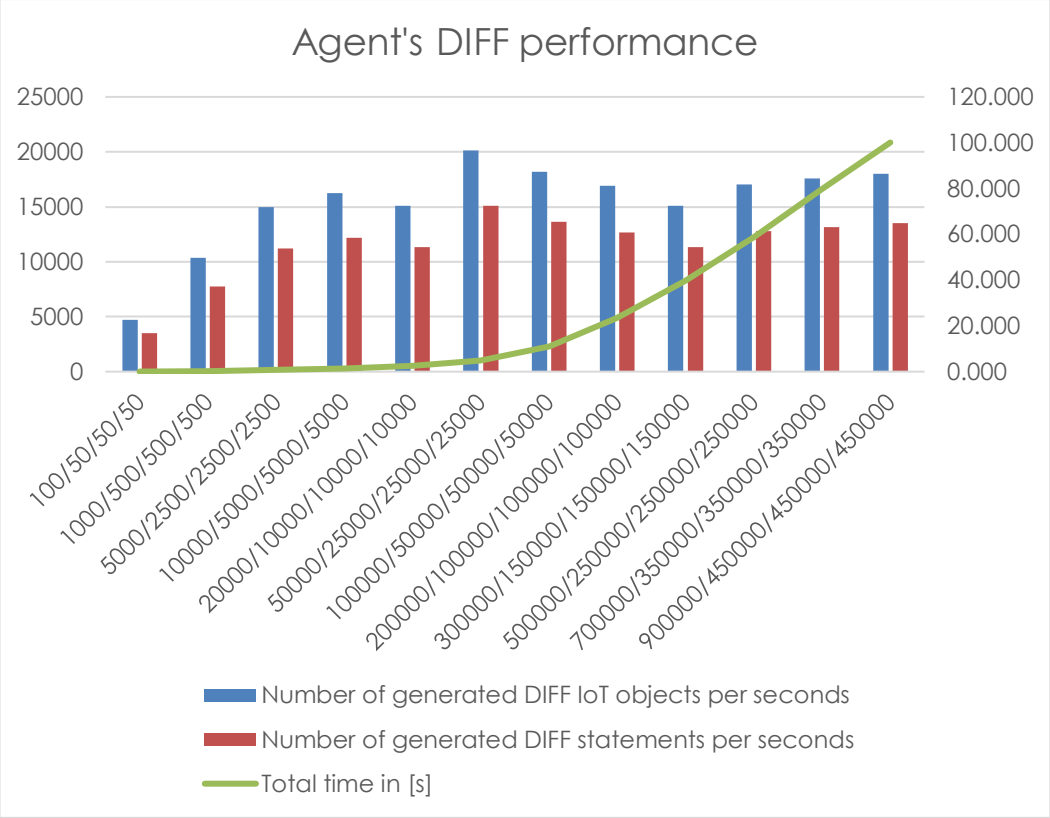

Figure 5 DIFF performance test results

The values at the x-axes in the Figure 5 in the form X/C/U/D have the following meaning – X is number of IoT objects in the NM for the given agent and its adapter. C is number of IoT objects to be created after the DIFF operation, U is number of the IoT objects to be updated and D is number of IoT objects to be deleted in the NM. The sum (C + U) defines the current state of the agent's adapter in terms of number of IoT objects.

For the given tests the average value of generated DIFF IoT objects per seconds is **15349** and the average value of generated DIFF statements per seconds is **11512**.

The key functionality of the agent's DIFF operation (methods in classes of the `sk.intersoft.vicinity.agent.service.config.processor` package) participated in the overall time of the test by approximately 26,5% as it can be observed from Figure 6.

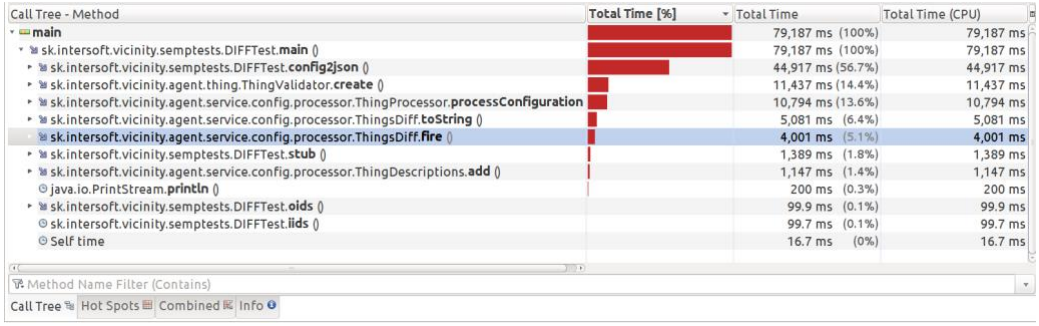

Figure 6 CPU consumption by methods in the DIFF performance test for 300000 objects

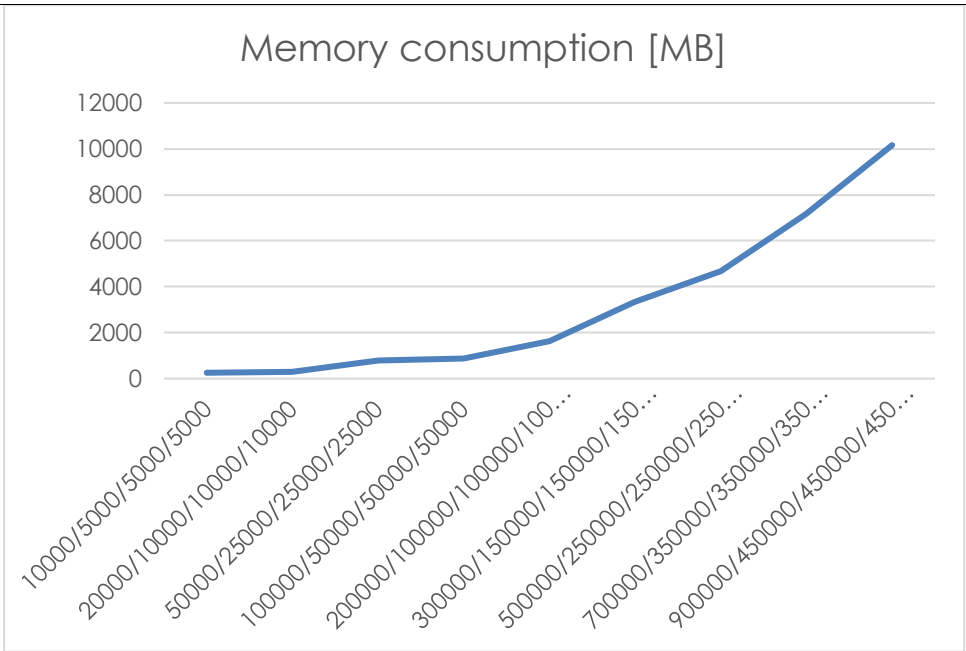

Figure 7 DIFF performance test results – memory consumption

The memory consumption in terms of the utilized heap size is shown in Figure 7. More details could be seen in Figure 8, where memory consumption of the DIFF performance test with 300000 objects is reported. As it can be seen, the used heap is 3GB, however 60% of that size are serialized form of the NM state in strings (char[]) used internally by the test application as a part of the stubbing functionality.

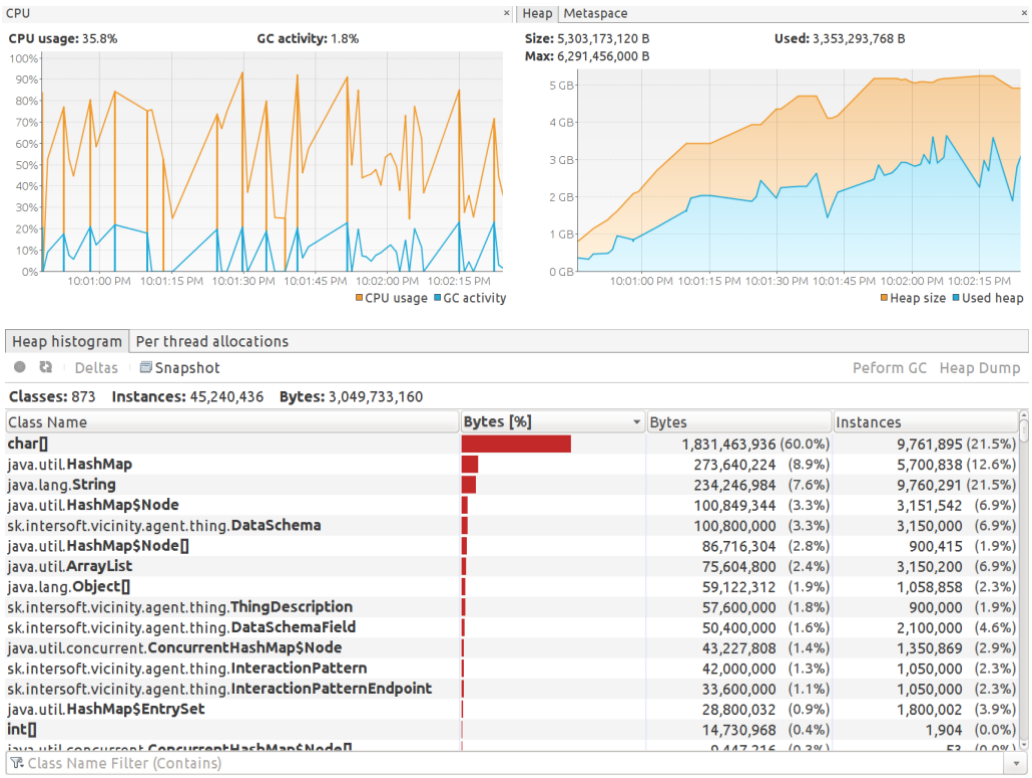

Figure 8 CPU usage, GC activity and Memory consumption of the DIFF performance test for 300000 objects

Deviations	None.
Other technical issues	None.
Status	Passed.
Notes	For every value at the x-axes in the Figure 5 the test for given combination X/C/U/D was run 10 times and total execution times for each case were computed as an average of execution times of single tests.

5. Test case 3 - SDACP performance

5.1. Testing objective

The goal of this test case is to measure performance and scalability of the SDACP component that provides the main persistence storage for the global state of the platform and semantic interoperability interfaces.

5.2. Testing scenario design

Table 7 SDACP performance – testing scenario design

Test 3	**SDACP performance**
Test ID	Test 1_IS
Test name	SDACP performance
Test scenario and goal	To test the performance of the SDACP – write/delete/query over the semantic model.
Performed by	Peter Bednar (IS)
Iterations	Scenario is executed in iterations parametrized by the following parameters: • Number of NM clients • Number of added statements • Number of removed statements • Set of test queries with the expected number of statements per results
Equipment, environments and IoT Infrastructure involved	• Stub connectivity to the VICINITY NM • Testing instance of the SDACP
Expected results	SDACP has expected performance and scalability measured by the specified measures (see the description of the Real results)
Test procedure	The test procedure consists of the following steps: 1. Initialization of the NM stubs and the SDACP instance 2. For each combination of scenario parameters execution of semantic operations

The clients of the SDACP are the platform clients, who can query semantic information about the IoT objects (things) on one side and the NM that integrates the running VICINITY agents on the other side. The state of the agents is propagated by the GW and the NM up to the interface provided by the SDACP, which then transforms the state updates into the triple store statements and update the global state of

the platform in the semantic repository. The context and the scope of the test case is presented in the following sequence diagram:

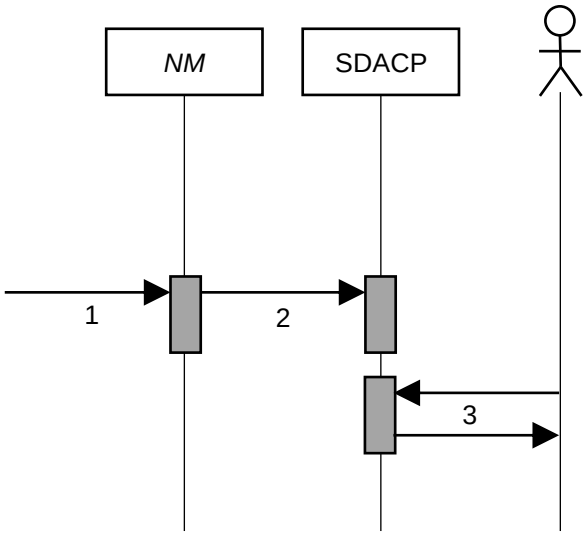

Figure 9 The context and the scope of the SDACP performance test case

1. Updates of the state of adapters are detected by the VICINITY Agents which generates updates
2. Updates are propagated to the SDACP through the GW and the NM. The NM is the only component interacting directly with the SDACP
3. Clients of the platform can query semantic information about the IoT environment through the standard semantic query interface (SPARQL entry point)

The case implements two scenarios:

1. It measures performance and scalability of update operations. This scenario is parametrized by the number of update statements of different operations (i.e. create, update, delete) and the size of the updated global state of the platform measured as the number of triples stored in the semantic repository.
2. It measures performance and scalability of the query evaluations. This scenario is parametrized by the set of test queries that can be received from the platform clients, the size of the global state of the platform measured as the number of the triples and the size of the result sets per each query.

In both scenarios, the main evaluation metric is the average execution time. Besides time, tests also report memory consumptions for the SDACP service and the semantic repository and disk usage for the semantic repository. The first scenario measures the time required for the batch uploading of the initial global state into the semantic repository.

The results of the test are presented in the graphs showing the dependency between the test parameters (i.e. number of changes and size of the global state) and measured metrics (i.e. time and memory consumption). The expected scalability is linear with the combination of all parameters.

In the first scenario, execution time covers transformation of the update operations into the semantic statements and updating of the semantic repository. The communication with the NM is not included, but test covers local communication between the SDACP client code and the semantic repository used as the

SDACP persistence layer. In the second scenario, the test includes local communication between the client and the SDACP semantic repository. Both scenarios are implemented as the unit tests connected to the locally deployed semantic repository.

5.3. Testing Platform

The test was implemented as a Java console application. The main procedure loads the data from the pre-generated JSON files and test of batch series of the insert, update, delete and query operations. The insert, update, and delete operations are implemented directly as HTTP requests to the SDACP REST service interface using the same client implemented in the NM. During the batch operations the main measured metric was the execution time, including also the parsing end validation of the responses with the status of the performed operations. Query operations were executed through the REST SPARQL interface. Measured time of the queries include the request time and parsing of the result set JSON into the object-oriented memory representation which allows the clients directly process the result data. All operations were performed on the SDACP testing installation.

5.4. Testing Results

Table 8 SDACP performance – testing results

Test 3	**SDACP performance**
Real results	The main metrics measured by the tests are the average processing time per IoT object for insert, update, and delete operations, as well as, the average processing time for queries. All tests were running locally on the testing instance of the SDACP prepopulated with the around 460 000 triples. The results for insert, update, and delete operations are presented in more details in the following part. The tests measure how the SDACP reacts to the increasing number of IoT objects in one batch, which is processed sequentially. The main result is that the processing time is not changing significantly up to hundreds IoT objects per batch. For testing we have used a set of IoT objects generated from two Gorenje devices (refrigerator and oven). Since the size of the description (i.e. number of properties and events, etc.) is roughly the same, average processing time per IoT object can be directly compared to the number of generated triples in the semantic repository (approximately 140 per object). The query testing was performed using the set of predefined queries in SPARQL. The average processing type per query was 0.058 s to 0.0655 s. The main deviations between queries were related to the number of triples in the result set. Complexity of the query doesn't have major impact to the performance. From the profiling analysis, most of the processing time is spend on communication over the HTTP and parsing of the results. The internal evaluation of the query by underlying storage (GraphDB) doesn't influence the query processing so much.
Real results (figures)	The following chart presents the dependency between the average times per IoT object for the insert operation. The number of IoT objects generated in the batch insert is given on the x axis. As it can be seen, the differences between the average processing times for increasing number of IoT objects in batch is negligible (less than 0,0002 s compared

to the previous case).

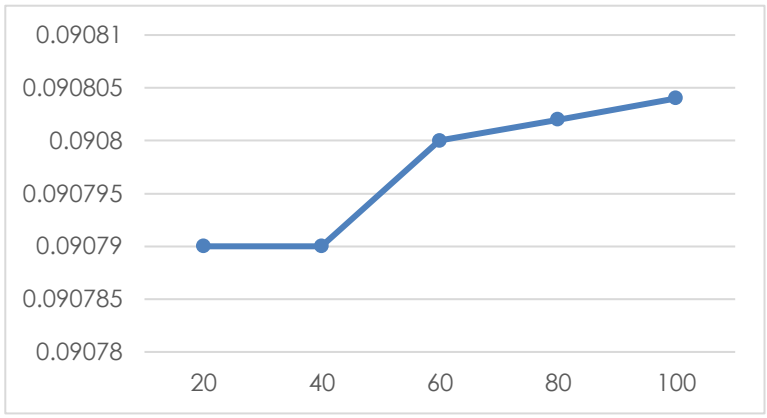

Figure 10 Average processing time for insert operations in relation to the number of IoT objects batch processed in parallel

Similarly to inserts, the following chart presents dependency between the average processing time and the number of IoT objects in batch operation for updates. The differences are again very small and there is just minor increase of processing time for larger batches.

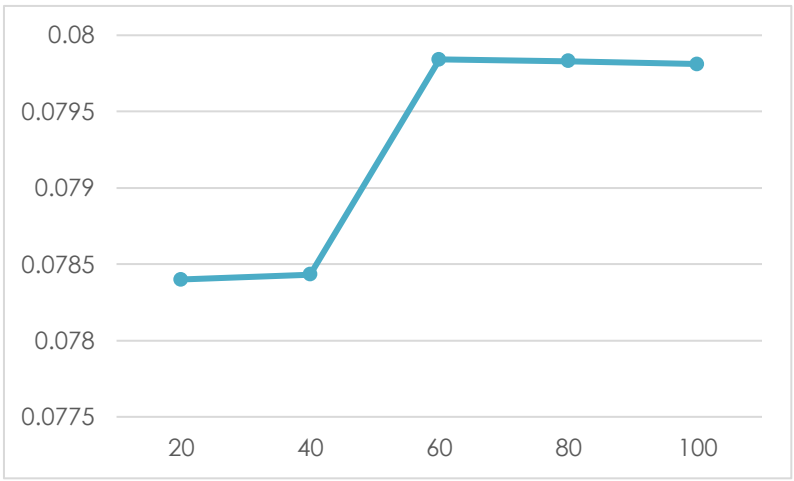

Figure 11 Average processing time for update operations as the relation to the number of IoT objects batch processed in parallel

The average time for the delete operations was around 0.02 seconds with the higher variances, but again, the processing time is not changing dramatically when more objects are deleted at the same time in one batch. However, although the processing time is quite short, the main problem with the current setup was mutual blocking of parallel delete operations, since the current installation of GraphDB can handle only two parallel workers processing the client requests.

Deviations	None.
Other technical	Current development and testing installation of the SDACP is based on the free licensed version of the GraphDB store, which is limited to two parallel client sessions. This setup

issues	can dramatically reduce scalability, which can be further enhanced by an unlimited commercial version of GraphDB.
Status	Passed.
Notes	None.

6. Quality and performance feedback

The proposed and realized test cases within the task T6.3 "VICINITY auto-discovery space deployment & validation" that are described in this document proved that the VICINITY Auto-discovery platform is deployable and fully functional. It provides necessary functionality required by pilot cases to achieve interoperability at the semantic level. The test executions were performed using the data from individual pilot cases in form of Thing Descriptions provided by VICINITY adapters.

Current implementation of the VICINITY Auto-Discovery platform suppose that all devices that should be discovered, and whose descriptions will be inserted into the semantic triplestore by the VICINITY Agent must be known by the Agent. It means that if a device should be part of the VICINITY platform, the Vicinity adapters model[5] should be extended. Technically speaking - core:Device subclass for device type, core:Service subclass for service type and necessary sosa:Property instances should be added. VICINITY Agent applies the IoT object description contract violation rules. Based on the experience from deployments of the pilot cases the services for dynamic introducing of new types and adding new instances should be considered.

Therefore, in this task we moved the focus from analysis of the behaviour of the Auto-Discovery platform in case of unknown devices towards validation of the full discovery & registration process. The process involves the VICINITY Agent at the client side and the Semantic Discovery and Agent Configuration Platform (SDACP) at the server/cloud side.

The first test case, particularly its part that tested limits of current implementation in terms of simultaneous registration of IoT objects by one or multiple adapters, opens a space for improvement of the auto-discovery functionality. Direct reason, why no more than dozens of IoT objects could be discovered and registered in single operation issued by one adapter or in parallel by multiple adapters was communication time-out between VICINITY Gateway and VICINITY Agent. These timeouts are caused by limitations of GraphDB Free store, enabling only two connections in parallel. This leads to the situation, that execution time of large numbers of create/update/delete operations in the GraphDB takes too long time. However, update to commercial, not limited GraphDB technology promises to improve the triplestore performance. Currently, the implemented solution enables the implementation of proposed pilot cases, but also represents a restriction in the scalability if large-scale pilots are considered. This feedback is in line with results of the third test case that focused directly on the Semantic Discovery and Agent Configuration Platform – services providing the functionality of the semantic store. Conveyed tests clearly proved that freely available edition of the utilized Semantic graph database GraphDB used as triplestore significantly restricts the performance of whole Auto-discovery platform. The limitation to two requests in parallel practically prevents tests with higher loads and enables to confirm functionality of the platform only at the size level of the VICINITY official pilot cases. If the precondition of free available software could be abandoned, commercially available editions promised to remove such a restriction[6]. Possibility to use one of the non-freely available editions in development and production environment of the VICINITY platform should be analysed.

The second test case proved that the core functionality of the VICINITY Agent can manage significantly higher load that it is generated by pilot cases of the VICINITY project. VICINITY Agent's implementation of the DIFF operation is ready to be used also by large-scale pilots without additional improvements.

[5] http://iot.linkeddata.es/def/adapters/
[6] https://www.ontotext.com/products/graphdb/editions/

7. Conclusion

In order to validate the Auto-Discovery space deployment of the VICINITY platform three test cases were designed, implemented, and repeatedly executed. On one hand the results of the tests clearly proved full functionality of the Semantic Discovery and Agent Configuration Platform (SDACP) at the server/cloud side together with the VICINITY Agent at the client/node side. On the other hand, they also made visible the performance restrictions that come mainly from the use of the free available edition of the GraphDB used as the semantic store in the SDACP.

Designed and implemented tests offer a framework for checking, if the future enhancement of the Auto-Discovery platform implementation will improve the proposed and measured quality and performance parameters.