Project Acronym	**Fed4FIRE**
Project Title	**Federation for FIRE**
Instrument	**Large scale integrating project (IP)**
Call identifier	**FP7-ICT-2011-8**
Project number	**318389**
Project website	**www.fed4fire.eu**

D7.6 – Report on Third cycle developments regarding trustworthiness

Work package	WP7
Task	Task 7.2, 7.3 and 7.4
Due date	31/01/2016
Submission date	11/02/2016
Deliverable lead	Steve Taylor (IT Innovation)
Version	2.0
Authors	Steve Taylor, Vadim Krivcov (IT Innovation) Javier García Lloreda, Elena Garrido Ostermann (ATOS) Chrysa Papagianni, Elena Stai, Vassilis Kariotis, Symeon Papavassiliou (NTUA) Thijs Walcarius, Wim Van de Meersche, Brecht Vermeulen (iMinds)
Reviewers	Mark Sawyer (UEDIN)

Abstract	This deliverable reports on experiences and developments in cycle 3 in three main areas, corresponding to the development tasks in WP7. These concern the access control Policy Decision Point (PDP) in T7.2, the SLA management framework in T7.3, and the Reputation Service in T7.4. Each task is discussed in a separate section of this deliverable.
Keywords	Trust, Reputation, SLA, Security, Certificate, Credential, Access Control

Nature of the deliverable	R	Report	X
	P	Prototype	
	D	Demonstrator	
	O	Other	
Dissemination level	PU	Public	X
	PP	Restricted to other programme participants (including the Commission)	
	RE	Restricted to a group specified by the consortium (including the Commission)	
	CO	Confidential, only for members of the consortium (including the Commission)	

Disclaimer#

The information, documentation and figures available in this deliverable, is written by the Fed4FIRE (Federation for FIRE) – *project consortium under EC co-financing contract FP7-ICT-318389 and does not necessarily reflect the views of the European Commission. The European Commission is not liable for any use that may be made of the information contained herein.*

Executive Summary

This deliverable concludes the work in WP7. WP7 has sought to promote trustworthiness in Fed4FIRE, i.e. to give the users and participants of Fed4FIRE greater confidence that Fed4FIRE will perform as expected and that their participation in Fed4FIRE will not endanger them (i.e. reduction of risks to them). The specific contributions made by the tasks in WP7 are listed as follows.

- The Access Control work in T7.2 has contributed trustworthiness mainly in the protection of testbeds, thus increasing their confidence in participating in the federation.
- The SLA work in T7.3 has contributed to the trustworthiness of both experimenters and testbeds. SLAs increase the confidence of both experimenters and testbeds because an SLA is an agreement between both parties that shows each side what is promised and what will happen should it not be delivered.
- The Reputation work in T7.4 contributes to the trustworthiness of testbeds and experimenters by providing an independent platform where experimenters can rate testbeds based on their experience of using the testbeds. The reputation service assists experimenters in selecting testbeds for resource provisioning, based on previous users' experience of the testbeds, helping the users find the most reliable testbeds.

Overall, WP7 has been about reduction of risks to the federation participants and enabling increased reliability, thus increasing the participants' confidence (therefore trust) in their participation in the federation. The mechanisms for this has taken different forms, from increased protection of federation resources to giving federation participants useful information to enable them to make informed trust judgements about their participation in the federation and the resources they want to use.

This deliverable has reported on experiences and developments in cycle 3 in three main areas, corresponding to the development tasks in WP7.

For T7.2, the PDP has been updated based on the analysis provided previously in D7.5 and also experience gained during testing within BonFIRE.

For T7.3, the work regarding SLA management during third cycle of the project has been dedicated to the improvement and bug-fixing of the modules that compose the SLA architecture, as well as the development of a new tool for testbed providers to easily manage their SLAs.

For T7.4, the Reputation component has been updated and integrated with other components of Fed4FIRE. These are the monitoring components (OML), which provide information on the resources used for an experiment, and experimenter tools, which enable the experimenter to query the reputation of a testbed service, or to provide feedback about a testbed's service.

Acronyms and Abbreviations

AM	Aggregate Manager
Authn	Authentication
Authz	Authorisation
CA	Certificate Authority
CMS	Content Management System
CRUD	Create, Read, Update, Delete
CSR	Certificate Signing Request
DN	Distinguished Name
FRCP	Federated Resource Control Protocol
FTUE	Federated Trust and User Experience
GID	Globally Unique Identifier
GUI	Graphical User Interface
HRN	Human-Readable Name
IdP	Identity Provider
MOS	Mean Opinion Score
OMF	Orbit Management Framework – a reference implementation of FRCP
OML	ORBIT Measurements framework and Library
PDP	Policy Decision Point
PEP	Policy Enforcement Point
QoE	Quality of Experience
QoS	Quality of Service
RAG	"Red-Amber-Green" – status indicators for components operational state for first level support
RC	Resource Controller (in OMF)
REST	Representational State Transfer
ROCQ	Reputation, Opinion, Credibility, Quality
SA	Slice Authority
SFA	Slice Federation Architecture
SLA	Service Level Agreement
URN	Uniform Resource Name
UUID	Universally Unique Identifier

Table of Contents

1 Introduction 7
2 Report on PDP Developments 8
2.1 Requirements from D7.5 8
2.1.1 PDP Generalisation 8
2.1.2 Installation 9
2.1.3 PDP Configuration 9
2.1.4 PDP Distribution 9
2.1.5 Administration of the PDP 10
2.1.6 PDP Error Handling 11
2.1.7 PDP Performance 12
2.1.8 More General Interfaces (Renamed from “Generalisation” in D7.4) 15
2.2 Requirements from BonFIRE Stress Testing 15
3 SLA Management 18
3.1 SLA architecture in Fed4FIRE 18
3.2 SLA Management module 19
3.3 SLA Collector 20
3.4 SLA Dashboard 21
3.4.1 Template-Agreement Lifecycle 22
4 Reputation Service 24
4.1 Introduction 24
4.2 Reputation Service Architecture in Fed4FIRE 25
4.2.1 External interfaces 26
Monitoring 26
4.3 Experimenter Tools 27
4.3.1 Reputation in jFed 29
5 Conclusions 31
6 References 33
Annex A: SLA Collector API 34
SLA Collector as a Proxy 35
Annex B – Reputation Service REST API 36

1 Introduction

This deliverable reports on experiences and developments in cycle 3 in three main areas, corresponding to the development tasks in WP7. These concern the access control Policy Decision Point (PDP) in T7.2, the SLA management framework in T7.3, and the Reputation Service in T7.4. Each task is discussed in a separate section of this deliverable.

2 Report on PDP Developments

The Fed4FIRE PDP's purpose is to provide an access control system that will protect testbeds who support OMF from unauthorised access, and to enable the access control decision to be grounded in the SFA concepts and tokens that are used in Fed4FIRE.

This section discusses updates to the PDP in cycle 3. Based on the specification described in D7.4 and additional stress testing in the BonFIRE testbed, the PDP has been updated, and this section describes these updates. Firstly, Section 2.1 discussed how the requirements from D7.5 (report on cycle 2 developments) were addressed, and following this, Section 2.2 describes how the additional specific requirements from the BonFIRE stress testing were addressed. Some of issues arising from the BonFIRE stress testing overlapped with the requirements from D7.5, and these are discussed in section 2.2.

The PDP has been updated as discussed in the text in the next sections, and a distribution of the updated version is being prepared for release within the project.

2.1 Requirements from D7.5

The sections following correspond to the sections in D7.4 [1], to describe how the specification has been implemented. The format chosen is to keep the requirements that originated in D7.5 [3] (in *italics*), where the evaluation of cycle 2's developments took place, and to discuss how they were addressed in comparison with the plans discussed in D7.4 [1].

2.1.1 PDP Generalisation

Requirement 12: *Message handling and policy rule configuration should be separated. Deployments are named by testbed (e.g. vwall or bonfire) in the configuration file, and the deployment name is used to determine the message handling (how to get facts, attributes, etc from message) and policies (how message outcomes relate to decision making). Coupling these two functions together in collections specific to one deployment is firstly architecturally problematic as the two functions are different aspects and secondly not scalable to further deployments.*

Requirement 21: *Remove any deployment-specific assertion message parsing. The existing PDP contains special bean objects corresponding to specific deployments, and these should be removed where they are not generally applicable to other deployments. Standardised assertions should be investigated to replace these specific objects.*

Requirement 22: *Identity standardised mappings for standard types of credentials. Parsing different credentials such as X.509 Certificates or Slice Credentials has revealed same patterns therefore providing standardised mappings will help to control what facts should be generated from different credential types and consequently used in rules.*

D7.4 stated that the approach chosen for generalisation was to employ as much standardisation as possible:

> "The approach chosen to address these requirements is to employ as much standardisation as possible. This standardisation is mainly in the vocabulary chosen for assertions and access policies, and how the assertions are parsed. If standard methods of parsing and standard vocabularies are adopted, then standardised (i.e. not deployment-specific) message and rule processing can be adopted, thus removing the need for deployment-specific processing and enabling them to be processed separately." [1]

The simplified parsing of message tokens and assertions has been implemented on the PDP server side, and example scripts have been prepared using the vocabulary described in appendix 1 of D7.4. In addition, the separation of slice credential and OMF request processing has been completed, thus paving the way for replacement of either section should another credential or message schema be required.

2.1.2 Installation

Requirement 10: *Currently the PDP installation is a complex and manual process, which is prone to errors and a barrier for adoption. In order to ease the PDP installation and initial PDP configuration procedure appropriate automated tools (such as Vagrant or Docker) should ideally be used to create fully automated PDP installation and initial configuration script(s).*

As described in D7.4, the PDP installations use Vagrant scripts. These scripts download and install all the required dependencies, and install the PDP with the appropriate configuration. There is some small configuration (e.g. installing certificates) but the major configuration is automatic. This has made it much easier for testbed operators to deploy the PDP because all the pre-requisites and major configuration aspects are automatically installed and configured.

2.1.3 PDP Configuration

Requirement 11: *The current PDP installations are hard coded to each deployment. This means that there are multiple versions of the PDP to maintain, which is not scalable.*

Requirement 13: *The policies that apply to an installation should not be distributed with the PDP – instead, they should be set by the systems administrator on installation. Example policies as templates should be included in the distribution so as to guide the testbed administrator in creating the policies for their testbed.*

The standardisation of the PDP's installation discussed previously has addressed the requirements regarding the PDP configuration. In addition, all hard-coded configuration specific to a deployment has been removed from the PDP codebase and re-located into the configuration. Given that the distribution now uses Vagrant scripts, most of the configuration parameters are now standardised, so only the bare minimum configuration is needed for an individual deployment.

2.1.4 PDP Distribution

Requirement 15: *The generic PDP distribution should not include specific testbed deployment scripts (i.e. Authorised Asserter CLI scripts etc.) – these should be distributed separately.*

This requirement has been addressed in the previous sections – each installation required for a PDP-enabled OMF environment is now provided as a distribution that includes a Vagrant script for the pre-requisite components and configuration. There are in total four VMs, corresponding to the installations:

- PDP installation
- PEP – The OMF RC that takes orders from PDP
- Authorised asserter
- User client – sends requests for resource usage to testbed along with access tokens.

2.1.5 Administration of the PDP

Requirement 16: *Investigate tools to support visualisation of policies and logs for systems administrators. Commands are provided to get the working memory state by listing asserted facts, but this is a complex low level view on the data and for any large deployment would be unreadable by humans. This is problematic when trying to investigate the current state of access to resources and the facts contributing to a security decision. Readability of policies and facts is important for trust in the system itself and knowing that the correct decisions are being made.*

This requirement has been addressed in a different manner to that described in D7.4. D7.4 recommended that a database was used to record logs, but the current logging remains in files, with greater control for the testbed administrator and less logging by default. The reasons for this are given below.

Testing within BonFIRE (reported in D4.6 [2]) showed excessive logging, meaning that many messages were written to log files with the consequence that it was very difficult to find a specific event because of the vast quantities of information. The cause was found to be that the logs were recording all events because many log messages were set to the wrong log-level. Typically a software system logs information at different log-levels:

1. Debug – debugging information, mostly used by the developers. This is the most detailed log level, and includes all information from the log levels below.
2. Info – important information, mainly again used by developers for important events. This is a moderate amount of logging information, and includes all information from the level below, meaning errors are also included.
3. Error – reporting when an error occurs. This is the sparsest logging level, used to indicate when an error has occurred.

At runtime, the testbed operator can tell the system which log-level they want to record, and the more detailed log-levels are suppressed. A developer would probably want to use DEBUG, but an operational system would probably want to use INFO or ERROR.

It was found that too many log messages were being set at INFO level, when they should be at DEBUG level, so when the deployment was started, excessive messages at INFO level were being recorded. This was rectified and the result was much cleaner (and easy-to-read) log files when the level was set to INFO or above.

It was felt that this solution was better than employing a database because it did not require the installation and management of a database dedicated to logging, and with the greater control and reduction of verbosity the update provided, the main objective of producing meaningful logs that could be easily read by testbed administrators was achieved. In addition, should a database be chosen for the logging store rather than simple files, this would involve the need for the testbed administrator to write queries, every time they wanted to look at the logs, which would be an additional overhead barrier. Given these reasons, it was felt that simple log files were more useful and given the changes made to the logging described above, easier to navigate.

Requirement 17: *Investigate tools to support analysis of message authorisation decision outcomes for system administrators. Currently all message authorisation requests are logged to one common log file which makes it difficult for system administrators to analyse message authorisation requests (i.e. system administrators will need to analyse a large common PDP log file if there are errors and problems need to be fixed).*

This requirement has not been addressed directly in cycle 3, because the main problem reported by the testing was excessive logging, and this has been resolved using the approach mentioned above.

The resulting log files should be easier to navigate without the need for an additional database installation. In addition, the use of log file rotation (e.g. standardising on a single log file per day) should also make navigation and analysis of the log files easier. Given all these reasons, the need for dedicated analysis tools was not regarded as a priority.

2.1.6 PDP Error Handling

Requirement 18: *Mechanisms for ensuring the consistency of the fact base (i.e. handle duplicate facts etc.) need to be investigated. In the current version of the PDP, it is possible to assert mutually inconsistent facts (for example, a fact saying a situation is true and another saying the same situation is false), leading to indeterminate decisions.*

This requirement was addressed in the manner proposed in D7.4, with some additional logic to provide flexibility. The original challenge D7.4 addressed was the prevention of mutually exclusive or contradictory facts, and the following rules were proposed:

1. A subject-predicate combination can only have *one value only* (that is one object). Duplicate values are not allowed. This removes the possibility of inconsistency in the facts store.
2. If a duplicate value for subject-predicate combination is asserted, the default processing is to throw an error.
3. The asserter may optionally specify a flag: *"force_update = true"* for a predicate assertion and if the same subject-predicate combination already exists, the new value will update the existing value in the working memory. It is important that the force update flag is set per predicate assertion, so the asserter is aware of the behaviour and understands what will happen if there is a duplicate. [1]

These rules are adequate if it can never be possible that a predicate can have more than one value. Another type of predicate could exist where the same predicate could legitimately have more than one value, such as where the predicate represents a set of properties. An example could be that a slice has a set of resources:

- *slice_1 contains resource_1*
- *slice_1 contains resource_2*

These are clearly not exclusive, so both are valid.

The addition to the logic is for the predicates to be classed as:

1. “exclusive”, meaning that a subject-predicate combination can have one value only, or
2. “non-exclusive”, meaning that a subject-predicate combination can have more than one value.

In the case of the “exclusive” predicate, the logic described in D7.4 applies. The updated logic is as follows:

1. The default behaviour is that a subject-predicate combination is “exclusive”, and therefore it can have one value only.
    a. If a duplicate value for subject-predicate combination is asserted, the default behaviour is to throw an error.
2. The asserter may optionally specify one of two flags:
    a. *"force_update = true"*, which will result in the update of an existing assertion, or
    b. *“non-exclusive = true”,* which means the subject-predicate combination is permitted to have multiple values, so the assertion is permitted.

This mechanism ensures that the asserter understands the exclusivity of the assertions they are making. It is felt that the default position of "exclusive" assertions is a fail-safe situation.

Requirement 19: *When a request is denied (e.g. not authorised) no error message or any feedback is sent to the client. This decreases the usability of the system as the user has no idea what is going on. A mechanism for reporting the outcome of security decisions to the client should be implemented.*

This requirement for error handling was additionally highlighted in the testing conducted by BonFIRE, and reported in D4.6. If an access request was denied, there was no response, so the user had no idea what was happening. This was very unhelpful to the users, as if their request was denied, the system appeared to hang. The solution was simply to send an INFORM message to the requester's topic if the access request was denied. Once this was implemented, the system was much more usable from the experimenter's point of view – there was always a response to their request: either they were able to access the resources they requested, or they received a message informing them that their request was denied.

2.1.7 PDP Performance

Requirement 20: *The primary performance of the PDP can be measured in terms of transactions a second (e.g. security decisions are second). The performance will be influenced by factors such as the size of the fact base, complexity of policy rules, arrival rates and available compute resources. Performance tests should be conducted to understand how the PDP will scale for expected deployment conditions.*

The PDP has been tested for performance, and the main conclusion is that the key factor in terms of the PDP's performance is the number of assertions that are in its working memory at one time. To get an idea of the response time, the PDP was deployed and primed with different numbers of assertions associated with reservations (the most common long-lived assertions), and the response times measured. These were recorded in a Ubuntu Virtual Machine running inside a Windows 7 host machine (a medium performance laptop), and the results are shown in Table 1.

Table 1: Response Time vs Number of Assertions in Working Memory

# Assertions	Average Response Time (s)
50	0.007541053
100	0.014321159
500	0.053509042
1000	0.136898317
2000	0.292698585
4000	0.90054175
6000	1.599012281
8000	3.986360146
10000	5.292229398

It can be seen that the response times increase with the number of assertions, so an assessment of the requirements of the Fed4FIRE federation in terms of reservation assertions is needed to determine whether the performance is likely to be acceptable. Because the intention for the PDP is that it be deployed at testbed sites, the key factor in terms of the PDP's performance is the number of assertions

that are needed for all reservations at any one given time at a single site. The following analysis aims to analyse typical assertion numbers for Fed4FIRE deployment cases.

For a typical reservation, the PDP requires a number of assertions:

1. The slice ID
2. The slice owner
3. The start time of the reservation
4. The end time of the reservation
5. 1 assertion per resource attached to the reservation

Given the assertion structure associated with a reservation described above, there can be two extreme cases:

1. One resource per reservation. In this case, there will be five assertions per reservation. The number of reservations is limited by the number of resources, and the maximum possible number of assertions will be **five times the number of resources** – e.g. for 200 resources, the number of assertions is maximum 1000.
2. All resources in one reservation. Here there will be four static assertions (1-4 inclusive above), plus one assertion for each resource attached to the reservation. Given that all resources are in the reservation, the maximum number of assertions required is **4 + the number of resources in the testbed**, e.g. if there are 200 resources, 204 will be required for one reservation with all 200 resources in it.

Clearly, the most costly of these two cases in terms of assertions is case (1), because the maximum of 5 assertions are required for every resource reserved. Hence, this will be regarded as the "worst case" in the discussion below.

Since the number of resources within a Fed4FIRE testbed is a key factor in the number of assertions at a testbed's PDP, it is important to understand what these numbers are, and to this end, a naïve survey was undertaken of the Fed4FIRE testbeds to get an indication of the number of distinct resources available at each testbed. These numbers were derived by reference to the testbed description pages at the Fed4FIRE website[1]. Due to the heterogeneous nature of the testbeds in Fed4FIRE, and also how the resources are described, it is impossible to be accurate in each case, but the aim of this exercise is to provide an estimation of the scale of the resources available at each testbed, so as to determine the orders of magnitude in terms of reservation-based assertions the PDP should support, and to find the maximum number of distinct resources available at a testbed. In the following bullet points, direct quotations are used where possible, but in other cases summarisation has been necessary.

- "**The PlanetLab Europe Consortium** has over 140 signed member institutions: mostly universities and industrial research laboratories, each of which hosts **two servers** that it makes available to the global system."
- "The **NORBIT** testbed is a Wi-Fi testbed located in Sydney, Australia. It belongs to the NICTA group. The testbed consists of **38 nodes**."
- **W-iLab.t** has "**80 Atom based embedded PCs** with 2 Wi-Fi a/b/g/n interfaces and 1 IEEE 802.15.1 (Bluetooth)".
- **NITOS** has an outdoor and an indoor testbed, each of which has 40 nodes, thus the total for NITOS is **80 nodes**.
- "The **NETMODE** testbed is a Wi-Fi testbed belonging to the National Technical University of Athens (NTUA). It consists of **20 x86 compatible nodes** positioned indoors in an office environment."

[1] http://www.fed4fire.eu/testbeds/

- For **FUSECO**, "In the context of Fed4FIRE, a **fixed number of instances of both OpenIMS and OpenEPC** will be provided as a federated service. Experimenters will be able to get exclusive access to such an instance for a given amount of time."
- **PLATFORM LTE** provides "In general terms LTE connectivity is provided through **three different solutions**".
- **PL-LAB** has a total of **65 distinct resources**.
- **10G TRACE TESTER**. From the description, it appears that the **whole testbed** can be reserved to perform traffic generation and observation of the device under test.
- "**Community-Lab** is an open, distributed infrastructure where researchers can select among **more than 125 nodes**, create a set of containers (a slice), deploy experimental services, perform experiments or access open data traces."
- **LOG-A-TEC** (JSI) "… consists of several clusters of wireless sensor nodes. Approximately **70 sensor nodes** are mounted in different out-door environments..."
- **BonFIRE** – across EPCC and INRIA, up to 400 dedicated cores – **the most is 170 at one site.**
- **Virtual Wall** - There are currently 2 deployments of the Virtual Wall at iMinds (Virtual Wall 1 with 190 nodes and Virtual Wall 2 with **134 nodes**). In Fed4FIRE the initial target is to allow access to the newest instance, the Virtual Wall 2.
- **UBRISTOL OFELIA ISLAND** provides 3 x OpenFlow-enabled switches, 4 x Campus Grade Ethernet switches, 3 x ADVA FSP 3000 DWDM ROADM nodes, total **13 distinct resources**.
- **I2CAT OFELIA TESTBED** offers 5x OpenFlow packet switches and 3x virtualization servers – total **8 resources**.

Based on the above survey, the maximum number of resources at one site is 170, from BonFIRE. As described above, the "worst case" in assertion terms is that all resources are reserved at the same time, under separate slices, meaning that all assertions above are required per resource. In this case the number of assertions is multiplied by the number of assertions for a complete reservation (five). The largest number of resources at one site in the naïve survey above is 170, so this results in a total of 170 * 5 = 850 assertions. Consulting the PDP performance table above, the response time for 2500 assertions is 0.14 seconds.

If the testbed supports advanced reservation, there may also be multiple reservations for the same resources, so a multiplying factor can apply to the above worst case reservations, assuming they are all reserved independently in advance. There are no numbers for this, so an estimate must be made. Assuming the testbed has an average of 3 advance reservations per resource, we can again multiply our worst case total from about by this to get 2550 assertions. Consulting the PDP performance table above, the response time for 2500 assertion is 0.45 seconds.

These figures are worst case, and in many other cases there will be fewer reservations active within the testbeds. A typical case could be where the Virtual Wall is half-loaded with 20 reservations. Given that VWall 2 has 134 nodes, we can divide this by two to get 67 nodes, and allocate these within 20 reservations, so the allocation could be as per the following Table.

Table 2: Example Resource Allocation for a Testbed

Number of Reservations	Nodes per reservation	Total Nodes	Total Assertions[2]
2	1	2	10
4	2	8	24
4	3	12	28
5	4	20	40
5	5	25	45
20		**67**	**147**

This gives 147 assertions, and from Table 1 we can infer that the response time will be 0.02 seconds.

The cases above are approximate, but should give an indication of the performance of the PDP compared to typical deployment situations. Given the typical cases presented in this section, the response time performance is deemed to be adequate. If the situation changes significantly (i.e. the number of resources in a testbed increases greatly), the performance should be re-evaluated and updates made as necessary.

2.1.8 More General Interfaces (Renamed from "Generalisation" in D7.4)

Requirement 23: *Provide a REST interface for the Authorised Asserter. This will be used by the Authorised Asserter to assert or retract facts to and from the PDP without the need to use OMF tools.*

So far in Fed4FIRE, this has not been required, so it has not yet been implemented. However, should the need arise, it can be easily implemented.

2.2 Requirements from BonFIRE Stress Testing

This section describes the requirements arising from the stress testing performed in BonFIRE. Some of the requirements have been addressed already, and references to the relevant sections are provided. Other requirements are discussed in this section. In D4.6 [2], the results of the BonFIRE stress testing were reported as per the following annotated quotation. The requirements that have already been addressed are italicised and a reference is provided to the relevant section in this report after the requirement.

> "During cycle 3, BonFIRE tested and evaluated the PDP implementation. A number of issues were found as a result of stress testing, and these were shared with WP7. These issues are listed as follows.
>
> 1. In BonFIRE, it is common for an OMF RC not to be ready immediately the resources are allocated. When the resources are allocated, the PDP is informed, and it immediately tries to subscribe to its AMQP subject. If the OMF RC is not available when the PDP tries to subscribe, the PDP "forgets" the OMF RC. The only solution is to restart the PDP.
> **2.** The PDP uses lots of RAM and crashes when the process memory limit is reached. The PDP does not release memory frequently enough, which

[2] The total assertions are calculated by the formula:
*(4 static assertions in a reservation + num resources per reservation) * num reservations*

causes the PDP process to request more memory until the limit is reached and the process terminates.

3. *In the event of an authorisation denial, the PDP does not inform the OMF RC. This makes OMF experiments very hard to run, because the experimenter does not know what has happened.* **(Discussed in Section 2.1.6 - PDP Error Handling)**
4. *Excessive logging of the PDP. The log files contain everything that takes place. This makes it hard to see if something is working or not.* **(Discussed in Section 2.1.5 - Administration of the PDP)**

These issues were reported to the PDP developers and the updates made to the PDP to address them are described in D7.6." [2]

Two of these points (items 3 and 4) have already been discussed in previous sections, and they are highlighted above. The two remaining points are discussed here.

Item (2) is discussed first, as its solution has provided a mechanism to also address point (1), as will be discussed later.

Stress testing in BonFIRE were conducted with the following results (reported in D4.6):

> The PDP uses lots of RAM and crashes when the process memory limit is reached. The PDP does not release memory frequently enough, which causes the PDP process to request more memory until the limit is reached and the process terminates. [2]

This was clearly a bottleneck and its cause was investigated. The initial hypothesis for this bottleneck was that assertions were not being retracted, and as more were added, the PDP's memory image would grow. The PDP has two types of assertions – those associated with reservations, and those associated with a single access request. D7.4 discussed that the facts associated with an access request are placed within a transaction, where they are asserted, the authorisation test conducted, and the facts retracted. The other type of assertion, related to reservations, has a longer lifecycle than a single access request, and is not automatically removed from the working memory. Because access request assertions are always retracted immediately after the test, this led us to suspect that the reservation assertions made by the authorised asserter were not being retracted when they became invalid, thus causing an ever-growing database of facts, with impacts on performance and memory.

The PDP was deployed internally at IT Innovation, and this hypothesis was tested. The results showed that the PDP's memory image indeed grew with the number of reservation assertions in its working memory. A crash was not experienced in this testing, but this was attributed to an assumption that BonFIRE used smaller memory limits in VMs. In any case, there were clearly memory leaks in the PDP because its heap size increased in direct proportion to the number of assertions, and this needed to be addressed whether a crash was found or not.

Having confirmed the hypothesis, the solution was to provide automatic retraction of expired assertions. This had the aim of keeping the number of assertions in the memory of the PDP to a level that provides acceptable performance.

It was agreed that each reservation would have an expiry time associated with it, and once this time had passed, its assertions would be automatically retracted by the PDP, thus preventing the working memory expanding ad infinitum. The implementation mechanism for this automatic retraction was to create a separate thread dedicated to checking and retracting out of date reservation assertions. The thread sleeps most of the time, wakes periodically and queries for assertions that are expired given the current time. Any assertions that are out of date are retracted. The frequency the thread wakes

up is configurable by the testbed administrator, so can vary from deployment to deployment, but a likely starting point frequency is once per day.

At design time, the expected usage of the PDP from an authorised asserter point of view was that (e.g.) reservation assertions from authorised asserters would be infrequent and retracted by the authorised asserters once they were invalid. BonFIRE's use of the PDP involved a different case: there were frequent assertions made by the reservation components, and there were no retractions. Once the problem became apparent, it was decided to implement the solution above: to add expiry times to assertions and support automatic cleaning of those that were expired. It was quickly realised that adding this support would simplify the use of the PDP from the asserter's point of view - even though the PDP's API already supported retraction, it would require a significant of housekeeping effort on the part of the authorised asserter to remember to retract the assertions it had made. It is much simpler to specify an expiry time at assertion time, and the assertions are automatically retracted once they became invalid.

Item (1) is discussed next. For reference, the item is repeated below.

> In BonFIRE, it is common for an OMF RC not to be ready immediately the resources are allocated. When the resources are allocated, the PDP is informed, and it immediately tries to subscribe to its AMQP subject. If the OMF RC is not available when the PDP tries to subscribe, the PDP "forgets" the OMF RC. The only solution is to restart the PDP [2].

The requirement that the issue highlights is that there is a need for retries to subscribe to an OMF RC's topic, should the first attempt fail. The approach chosen to address this exploits the use of the additional thread that removes expired reservations. This thread sleeps most of the time but periodically wakes up and removes any expired reservations, so its property of periodically waking up is useful to attempt retries on any failed subscriptions. Hence, the thread was extended to also retry any failed subscriptions. Because the thread is already aimed at cleaning out any expired reservations, it knows about the set of reservations that are not expired, and this is the basis for the re-subscription. The overall algorithm for the thread is as follows:

1. Wake from sleep.
2. Get set of reservation assertions.
3. From this set, select the subset of reservation assertions whose expiry time is earlier than "now". The result is the set of expired assertions, and the remainder is the set of valid assertions.
4. Retract the assertions in the expired set.
5. For each assertion in the "valid" set, test its subscription status. For any that are not subscribed, re-attempt subscription.
6. Sleep for *n* minutes.

As a note on the selection of the time to sleep for, this is dependent on the nature of the deployment, so the sleep time is configurable by the testbed administrator. Some testbeds may prefer frequent cleaning and re-subscription attempts, so n may be small (e.g. 1, meaning 1 minute). Other testbeds may be happy with less frequent cleaning and re-subscription (e.g. 1440, meaning 1 day).

3 SLA Management

SLAs are offered by three testbed providers in Fed4FIRE: iMinds, Fraunhofer Fokus and NTUA. Since the end of previous cycle of the project, the main SLA component, the SLA Management module (SLA Core), has been deployed in those facilities.

The work regarding SLA management during third cycle of the project has been dedicated to the improvement and bug-fixing of the modules that compose the SLA architecture, as well as the development of a new tool for testbed providers to easily manage their SLAs. More specifically:

- **SLA Management module:** the key component for SLAs was introduced and improved during Cycle 2. The work during Cycle 3 includes integration with manifold to retrieve the metrics in order to calculate possible SLA violations.
- **SLA Collector**: similar to the SLA Management module, the work has been focused on adding some extra functionality to ease the integration with Reputation system and jFed tool.
- **SLA Dashboard:** the tool developed in Cycle 3 for infrastructure providers to create SLA templates for experimenters and visualize SLA agreements and their evaluation results.

To offer testbed providers complete support for SLAs, work towards integration between Fed4FIRE tools and monitoring mechanisms has also been carried out in Cycle 3.

3.1 SLA architecture in Fed4FIRE

The SLA architecture at the end of Cycle 3 is shown in Figure 1. It is composed by four elements:

- **SLA Management module:** this is the main component for SLAs. It is deployed in each testbed's facility and is in charge of managing the complete SLA lifecycle. This component was introduced in the federation during Cycle 2 and the work done on it during Cycle 3 has consisted on bug fixing and performance improvement adjustments.
- **SLA Collector:** this is the aggregation point for tools and services to access SLA information of all testbeds that support it. The SLA Collector can push and retrieve SLA information from the SLA Management modules of testbeds via REST API. Similar to the previous component, the development work of this tool during Cycle 3 has consisted of adding improvements and fixing bugs.
- **SLA Dashboard:** this is a tool developed during Cycle 3 for testbed owners to manage their SLAs from a web-based Graphical User Interface. With this tool, testbed owners can define SLA templates and visualize existing SLAs and their status. In the case that an SLA is not satisfied, this tool also shows information about the number of violations that had occurred.
- **SLA front-end tools**: these are extensions for federation tools that allow experimenters to make use of SLAs offered by testbeds. During Cycle 2, an SLA plugin for the MySlice Fed4FIRE portal was developed and during Cycle 3, jFed development team has also given it support for SLAs.

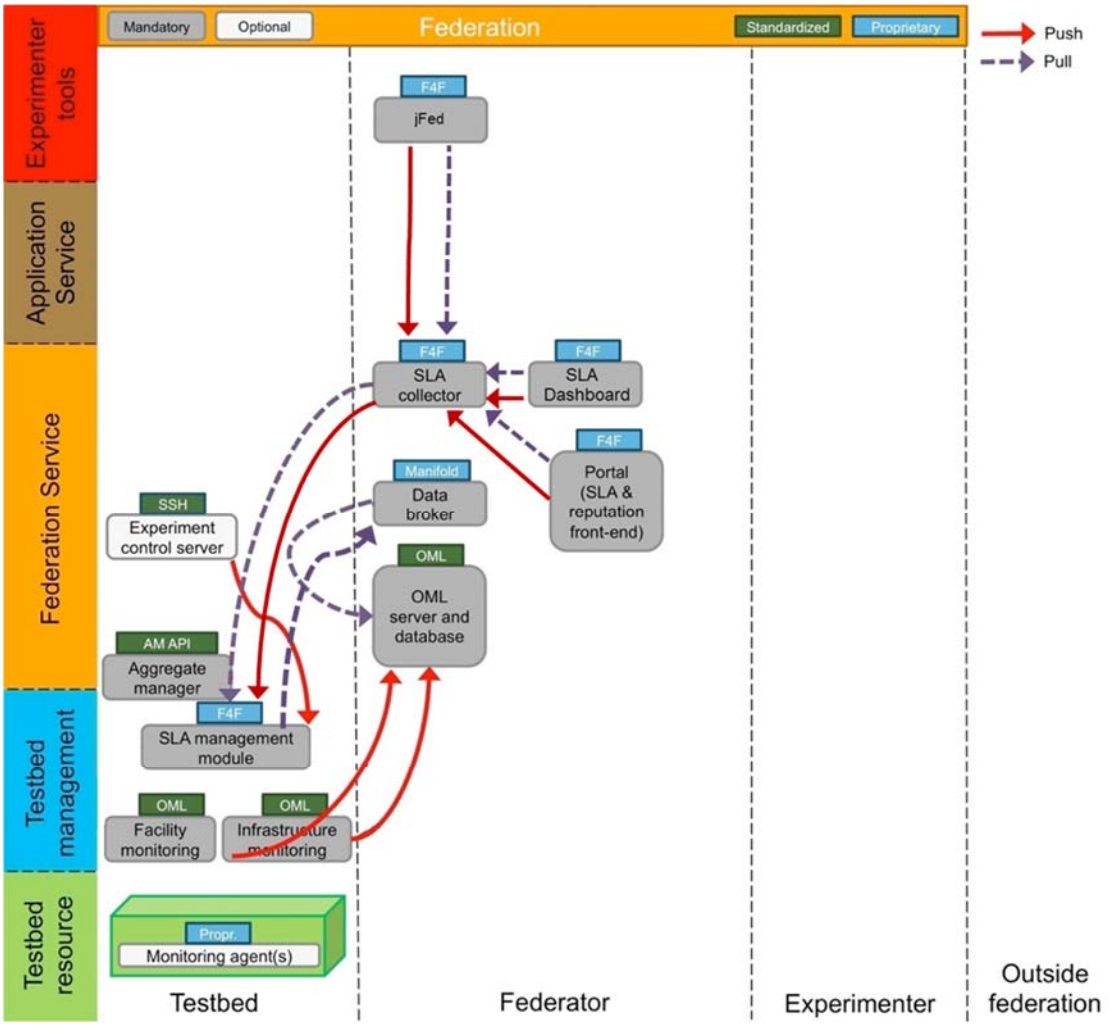

Figure 1. SLA architecture block diagram

3.2 SLA Management module

The SLA Management was introduced in the project in cycle 2. This component is located at testbed level and is the main element in the SLA management process. It is in charge of creating, storing, evaluating and destroying SLAs defined by testbeds.

During cycle 2 the metrics where retrieved from the monitoring database located at each testbed. This has changed during cycle 3, and the metrics are retrieved from a central database component of the federation called manifold.

Each agreement will have two guarantee terms. These guarantee terms have been mentioned in D5.6, they're the "Availability" and "Availability Compliance Ratio". "Availability" represents the percentage of time the sliver is available during the time the SLA is active. "Availability Compliance Ratio" is used for the percentage of slivers that will comply with the previous availability constraint. Both guarantee terms can have values between 0 and 1.

The SLA Management will evaluate these guarantee terms. The information of the slivers (or nodes) can be found in the field of the service scope from the SLA Agreement. This field has between square

brackets a list of coma separated values, where each name of the node has question marks at the beginning and at the end. An example of a Service Level Agreement in XML format is shown below.

```
<wsag:Agreement wsag:AgreementId="b5532b19-b882-11e5-9d53-001517becdc1" xmlns:sla="http://sla.atos.eu" xmlns:wsag="http://
www.ggf.org/namespaces/ws-agreement">
    <wsag:Name>Template for iMinds service</wsag:Name>
    <wsag:Context>
        <wsag:AgreementInitiator>urn:publicid:IDN+wall2.ilabt.iminds.be+user+bvermeul</wsag:AgreementInitiator>
        <wsag:AgreementResponder>wall1.ilabt.iminds.be</wsag:AgreementResponder>
        <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>
        <wsag:ExpirationTime>2016-01-11T19:45:33+01:00</wsag:ExpirationTime>
        <wsag:TemplateId>wall1.ilabt.iminds.be</wsag:TemplateId>
        <sla:Service>iMinds service</sla:Service>
    </wsag:Context>
    <wsag:Terms>
        <wsag:All>
            <wsag:ServiceDescriptionTerm wsag:Name="SDTName1" wsag:ServiceName="iMinds service"/>
            <wsag:ServiceProperties wsag:Name="NonFunctional" wsag:ServiceName="iMinds service">
                <wsag:VariableSet>
                    <wsag:Variable wsag:Metric="xs:double" wsag:Name="Availability">
                        <wsag:Location>iMinds/Availability</wsag:Location>
                    </wsag:Variable>
                    <wsag:Variable wsag:Metric="xs:decimal" wsag:Name="AvailabilityComplianceRatio">
                        <wsag:Location>iMinds/AvailabilityComplianceRatio</wsag:Location>
                    </wsag:Variable>
                </wsag:VariableSet>
            </wsag:ServiceProperties>
            <wsag:GuaranteeTerm wsag:Name="GT_Availability">
                <wsag:ServiceScope wsag:ServiceName="sla:iMinds">
                    ["urn:publicid:IDN+wall1.ilabt.iminds.be+sliver+102017",
                     "urn:publicid:IDN+wall1.ilabt.iminds.be+sliver+102018"]
                </wsag:ServiceScope>
                <wsag:ServiceLevelObjective>
                    <wsag:KPITarget>
                        <wsag:KPIName>Availability</wsag:KPIName>
                        <wsag:CustomServiceLevel>
                            {"constraint" : "Availability GT 0.99"}
                        </wsag:CustomServiceLevel>
                    </wsag:KPITarget>
                </wsag:ServiceLevelObjective>
            </wsag:GuaranteeTerm>
            <wsag:GuaranteeTerm wsag:Name="GT_AvailabilityComplianceRatio">
                <wsag:ServiceScope wsag:ServiceName="sla:iMinds">
                    ["urn:publicid:IDN+wall1.ilabt.iminds.be+sliver+102017",
                     "urn:publicid:IDN+wall1.ilabt.iminds.be+sliver+102018"]
                </wsag:ServiceScope>
                <wsag:ServiceLevelObjective>
                    <wsag:KPITarget>
                        <wsag:KPIName>AvailabilityComplianceRatio</wsag:KPIName>
                        <wsag:CustomServiceLevel>
                            {"constraint" : "AvailabilityComplianceRatio GT 0.99"}
                        </wsag:CustomServiceLevel>
                    </wsag:KPITarget>
                </wsag:ServiceLevelObjective>
            </wsag:GuaranteeTerm>
        </wsag:All>
    </wsag:Terms>
</wsag:Agreement>
```

Figure 2. SLA example in XML format

The SLA Management will request periodically the information about the availability of the nodes to the Manifold. The “Availability Compliance Ratio” guarantee term is a value that is calculated within the SLA Management module. As soon as any term from the SLA violated the information is recorded in the SLA Management database and can be retrieved using the RESTful services from the module.

3.3 SLA Collector

This module is the common entry point for tools and services to access SLAs across all testbeds. Introduced previously in Cycle 2, the current work around the SLA Collector has consisted on improving its performance by fixing bugs and refactoring the code base and adding extra features to provide better functionality for the tools that make use of it.

Since testbeds offering SLAs are now capable of defining their own SLA templates using the SLA Dashboard tool, one of the additional functionality in the SLA Collector has been the option of selecting on which SLA template the new SLA is going to be based. To do so, it is sufficient to simply specify the

template identifier in the corresponding field in the parameters for the SLA creation. See Annex A for more detailed information of SLA Collector API parameters.

The integration of SLAs with the Reputation system has also been done during Cycle 3. This has required that, in addition to retrieving all SLAs under a single slice, the option of further filtering SLAs by their expiration date needed to be developed. With this capability, tools can narrow the SLA result using the query parameter “expirationtime” in the GET call to retrieve SLAs from a specific slice.

More detailed information can be found in the API documentation of the SLA Collector in Annex A: SLA Collector API.

3.4 SLA Dashboard

The SLA Dashboard component has been introduced in Cycle 3. It is a centralized tool that will allow to each testbed provider (SLA-Administrator) creating its SLA templates. The testbed user, also named Experimenter, will be able to select (with the Fed4Fire portal) the template that best matches his needs. The testbed provider is able to see the agreements that have been created with his templates and, after they have been executed, whether they have been fulfilled.

Figure 3 shows how the SLA Dashboard interacts with the testbed. We take advantage of the SLA Collector, the component introduced in Cycle 2 that interacts with all SLA Management Module at each testbed. The SLA Collector has in its database the IP of each existing testbed. If a testbed provider has several testbeds in the federation (for example, iMinds has two testbeds), the tool will centralise the SLA template setup for all the involved testbeds, as shown in Figure 3.

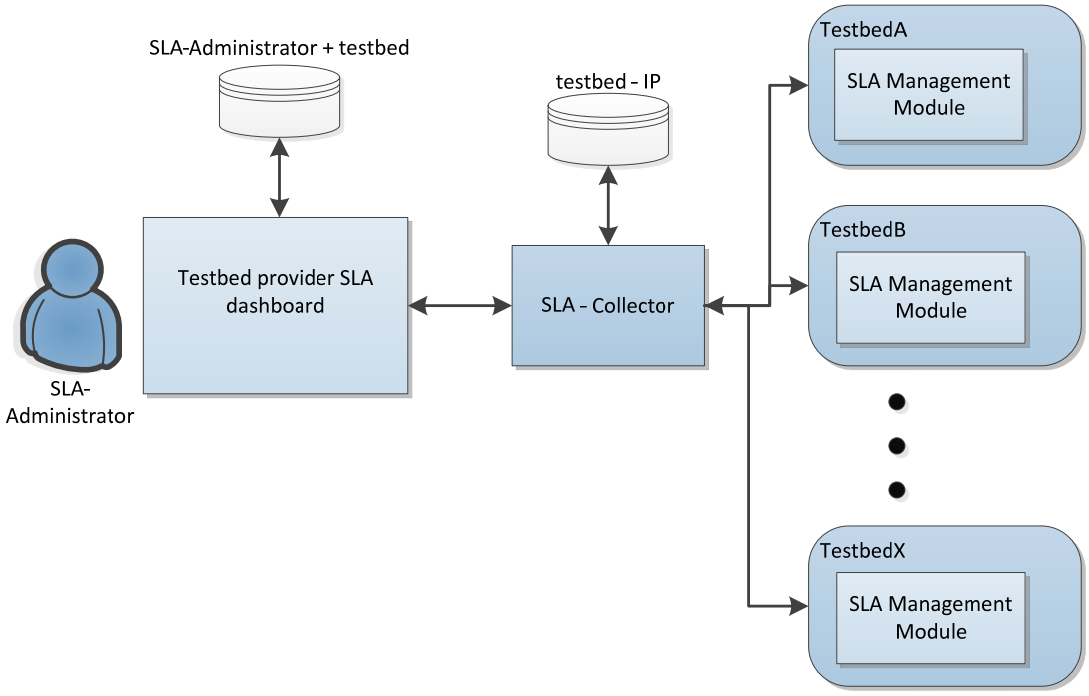

Figure 3: SLA-Dashboard in the Fed4Fire architecture

All the communication from the SLA Dashboard uses RESTful calls going out to the SLA-collector. These calls include information of the testbed to which it is directed. The SLA-collector distributes the call to the specific SLA Management Module installed in the testbed. A provider should never be able to

retrieve information from a SLA Management Module installed in a testbed not under their responsibility.

The SLA Dashboard has its own database of users, each associated to their corresponding testbed.

The SLA Dashboard is a web-based application. Any provider should be able to access from any computer that has a connection to the SLA Collector. An example of the GUI developed for the SLA Dashboard can be found in Figure 4.

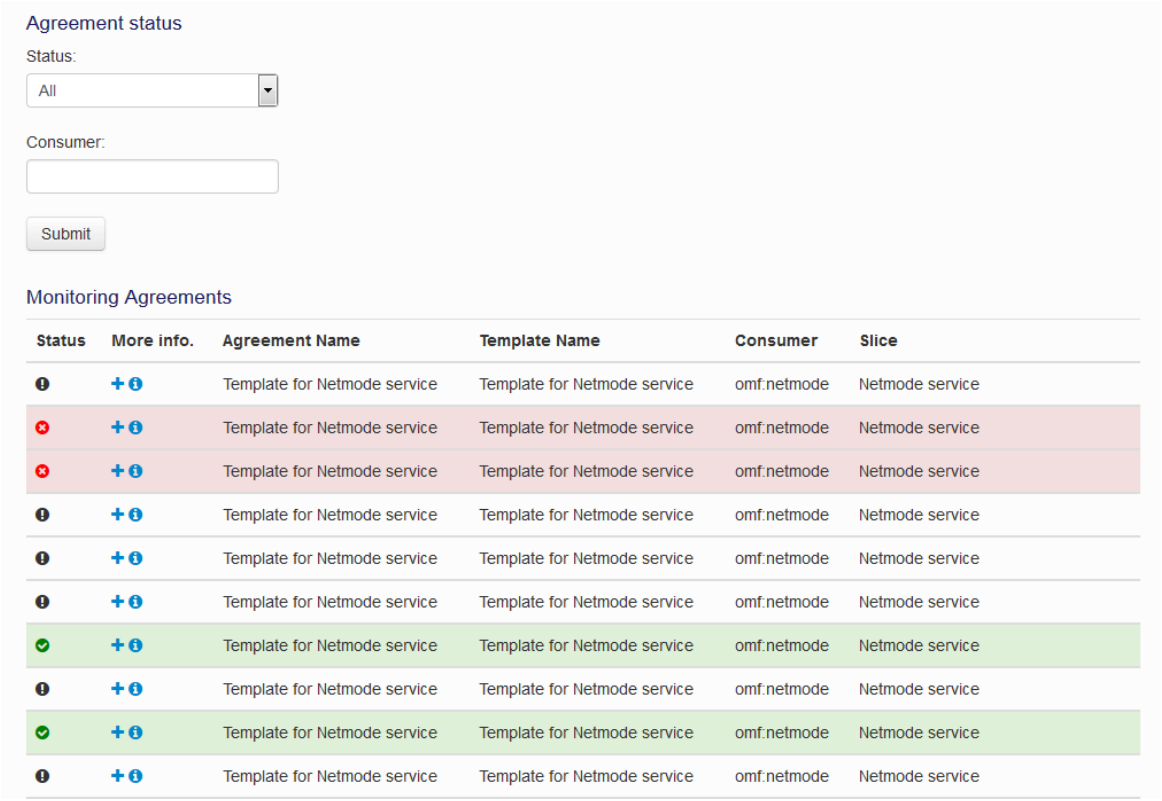

Figure 4: SLA Dashboard interface

3.4.1 Template-Agreement Lifecycle

The provider must generate the template according to his resources availability in the testbed (or in each testbed, in case there are several). All SLA templates define two metrics ("Availability" and "Availability Compliance Ratio") which are the guarantee terms and conditions to fulfil. The expiration date is also defined in the template. The SLA-Administrator can make use of the name to create the template versioning.

The tool doesn't implement any template-version management. If the SLA-Administrator wants to create another version from a previous template he will need to introduce it into the tool from scratch.

Once a template has been created, an Experimenter can use it (with other tools) and agree with the terms specified in the agreement, which will create the SLA.

The federation jFed tool will start the experiment and invoke the start of the enforcement job in the SLA Management Module. The enforcement job will retrieve the metrics from the monitoring and will evaluate if the guarantee terms indicated in the agreement are being fulfilled.

Once the job has been completed, both the experimenter (via the SLA Portal) and the testbed provider (from the SLA Dashboard) are able to check the result of the SLA evaluations that they have agreed upon.

More details on the SLA Dashboard's use as a tool can be found in the parallel deliverable D5.6.

4 Reputation Service

4.1 Introduction

The term trust has been used in various contexts in literature. However, one prevailing definition is that "*trust is the subjective probability by which an individual, A, expects that another individual, B, performs a given action on which its welfare depends"* [4]. The definition contributes to the notion of treating trust as an opinion, an evaluation and thus a belief. It includes the dependence on the trusted party and the reliability of the trusted party as seen by the trusting party [5]. The concept of *reputation* is closely linked to that of trustworthiness. Reputation can be considered as a collective measure of trustworthiness based on the ratings from members in a community [5]. Trust and Reputation have been the focus of research and have been widely used in P2P networks, web service and multi-agent systems, and more recently cloud environments [6].

In P2P networks, trust based on reputation was introduced to establish trust among peers, utilizing community-based feedback about past experiences of peers in order to assess their trustworthiness. In such systems, we assume service providers and provided services are not trusted. Service requesters select service providers based on their reputation values thus reputable service providers are selected to provide the service [7]. Transferring the paradigm to the domain of federated experimental infrastructures, a Reputation-based Trust Management system aims to assist experimenters (*service requesters*) in selecting among federated testbeds (*service providers*) for resource provisioning, by constructing a quantitative view of the trustworthiness of the *services* that each testbed provides.

The *Reputation Service (RS)* in Fed4FIRE implements the Reputation System, thus the mechanisms for building trustworthy testbed services. Services for testbeds are distinguished as:

- **Non-Technical Services** stemming from the non-quantifiable user's experience on conducting an experiment using resources from one or more testbeds in the federation. For example, the user's ***Overall Experience*** on conducting an experiment is considered a non-technical service. It is related not only to the technical characteristics of the testbed(s) involved but also factors like technical support, usability of tools etc.

- ***Technical Services*** emanating from the type of the testbed(s) involved in the experiment and its resources. For example, ***Availability*** of resources (up/down) throughout the duration of the experiment is a technical service for most types of testbed in the federation (e.g., wireless, wired, cloud).

The (experimenters) credibility and (testbed service) reputation models are described in Deliverable 7.5 [3], termed as Federated Trust and User Experience (FTUE) algorithm. The trust metrics (reputation scores) are based on the following parameters:

- The user's Quality of Experience that indicates the level of satisfaction for a service (e.g., resource **Availability**) that the experimenter receives from a testbed.

- Monitoring information with regards to the services provided by testbeds, assumed to be objective (thus used as the ground truth).

The reputation scores are available to experimenters and experimenters' tools (e.g., F4F portal, jFed) via standard API calls. Currently they are used to assist experimenters in selecting testbeds for resource provisioning, based on the trustworthiness of the "advertised" testbed services. Complementary they could be used by other federation services for performing their specific tasks (e.g., in resource

brokerage [Reservation Broker], using appropriate decision algorithms for testbed selection in experimenter's request partitioning among the testbeds).

4.2 Reputation Service Architecture in Fed4FIRE

The RS (also termed as FTUE framework) is a centralised reputation system. It consists of the (i) *Reputation computation Engine* used by the RS to derive reputation scores, (ii) the *Reputation Service Repository* and corresponding Database Management Functions and (iii) the Reputation Service REST API (Figure 5).

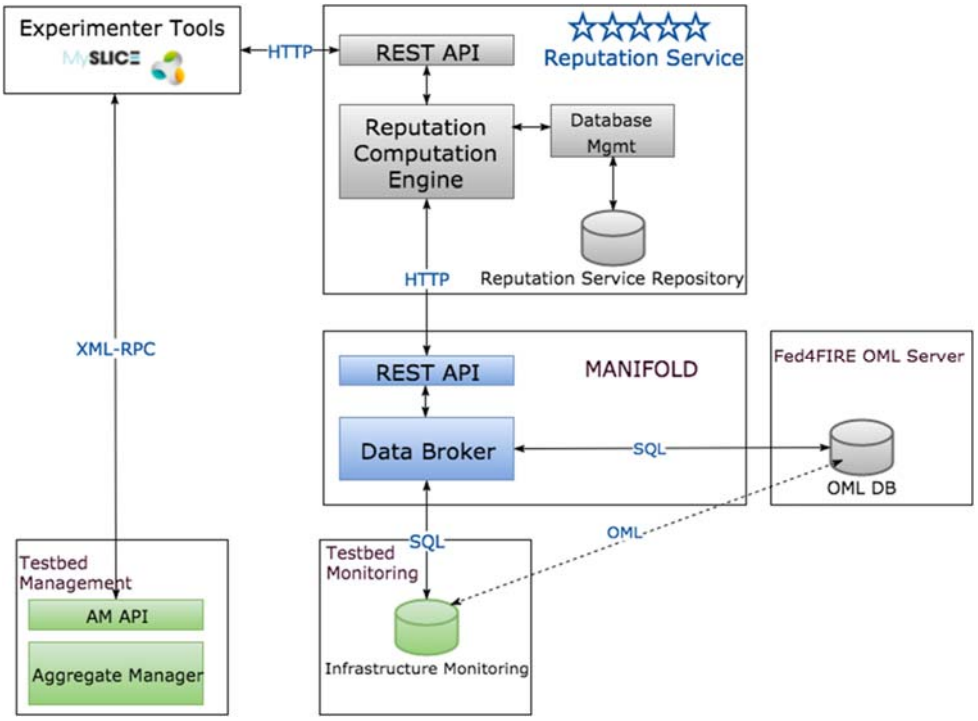

Figure 5: RS architecture and Interactions in Fed4FIRE ecosystem

Reputation Computation Engine

The *Reputation computation Engine* is a ruby-based implementation on the FTUE algorithm described in detail in Deliverable 7.5.

Reputation Service Repository

The *Reputation Service Repository* (PostgreSQL) stores information on the FTUE algorithm (e.g., testbeds, services, reputation scores etc.) as well as additional Information about the conducted experiments that have been rated by experimenters (e.g., resources, duration etc). The *Reputation Service Repository* data model has been *updated in Cycle 3* and the schema is provided in the following figure.

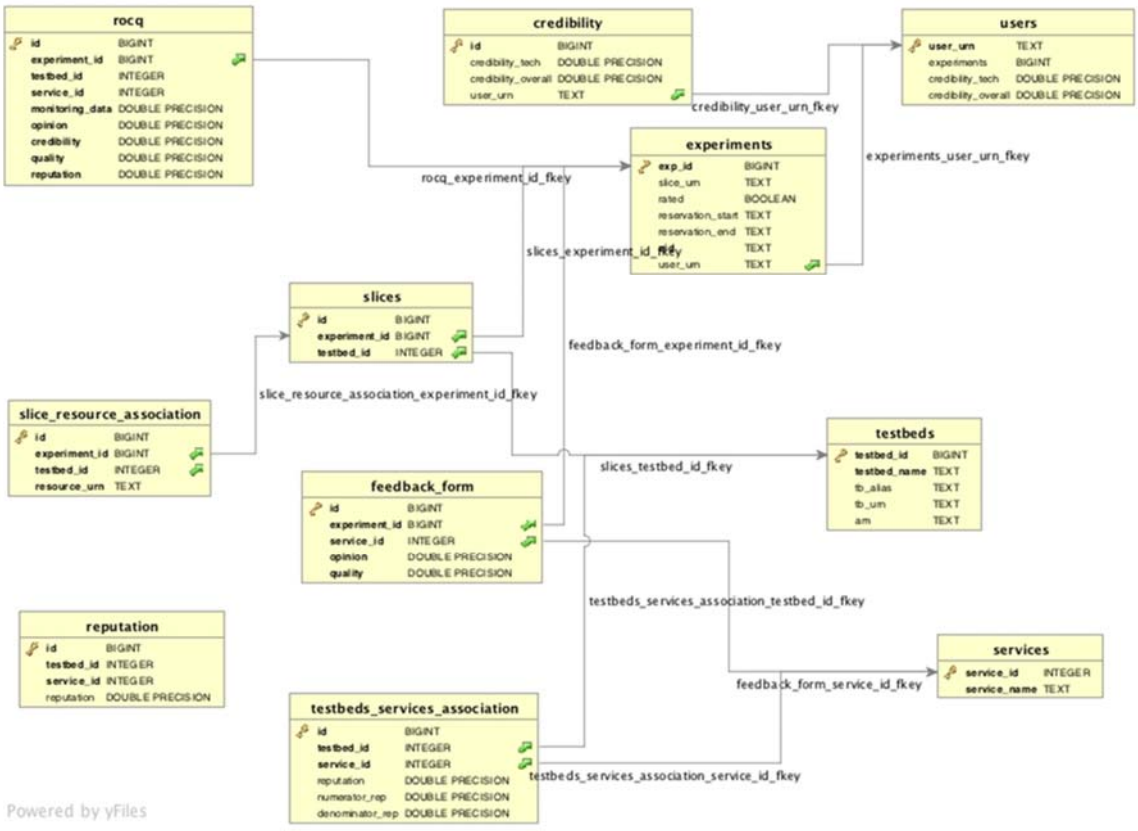

Figure 6: Reputation Service Repository Schema

4.2.1 External interfaces

REST API

The Reputation Service supports a REST API – this is described in detail in Annex B – Reputation Service REST API

Monitoring

Fed4FIRE has deployed an OML server that acts as an endpoint for receiving data according to the OML protocol. The OML server is deployed with a SQL database backend in which the **infrastructure monitoring data**[3] received by the server is stored. The OML server creates a new database in the SQL backend for each testbed reporting monitoring data; there is a table for each monitoring metric reported, including metrics used by the RS to monitor each testbed's technical services.

Testbeds are responsible for sending the corresponding monitoring data to the OML server of Fed4FIRE (using the appropriate OML client library [C, Python, Ruby]). Alternatively, the testbeds can deploy their own local OML server, so that they have a local database with their own monitoring data, and may report this data also to the Fed4FIRE OML server. In any case, as depicted in Figure 5, the RS retrieves the required monitoring data per experiment using Manifold, either from the Fed4FIRE OML

[3] Infrastructure monitoring: contains monitoring information about the infrastructure of the testbed, that is, the nodes and resources that integrate the testbed.

server or each testbed's local server. Specifically the Data Broker provides a REST API for accessing all OML streams. The RS has been fully integrated with MANIFOLD within the course of Cycle 3.

4.3 Experimenter Tools

Experimenters may use the *Reputation Service* via *Experimenter Tools* such as jFed or the Fed4FIRE portal (MySlice). Specifically through these tools they have the opportunity to:

- Provide ratings for technical and non-technical services of testbeds involved in the execution of an experiment.
- View reputation scores for the testbeds and their services.

The integration of RS with an *Experimenter Tool* requires that the latter implements the process of submitting user feedback and retrieving evaluation scores. The process has been updated in Cycle 3, with the integration of Manifold and the use of additional experimenters' tools (e.g., jFed). Indicative sequence diagrams are presented in the following.

Retrieve Reputation Scores for Testbed Services

List Reputation Scores: The tool must retrieve, parse and list the reputation scores for each testbed service. This is feasible using either the /reputation/showrep (Figure 7) or /reputation/show call on a per testbed basis.

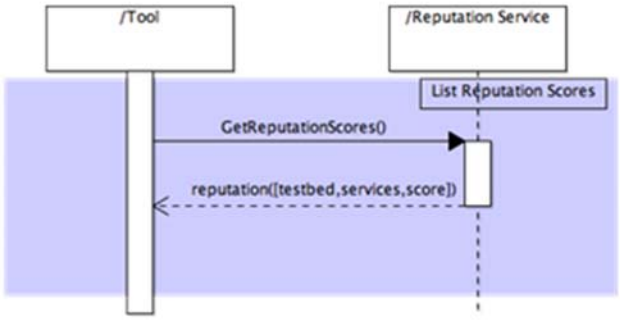

Figure 7: Retrieve Trust Metrics from the Reputation Service

Submit User Ratings

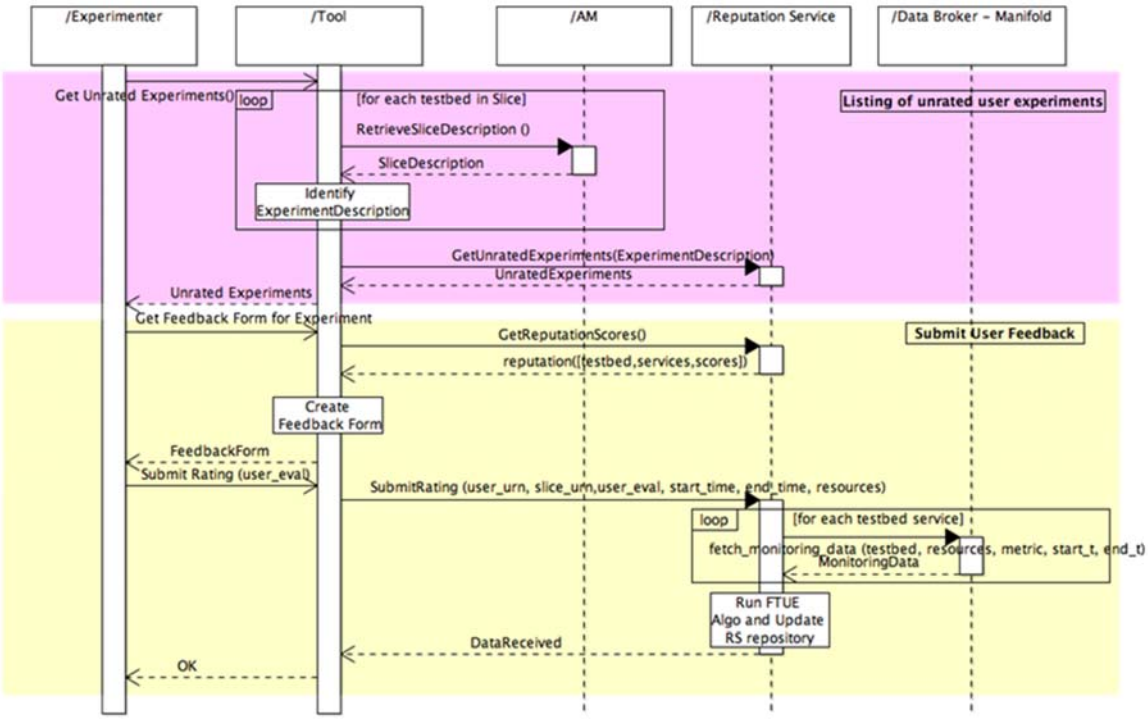

Figure 8: Submit Experimenter's Quality of Experience to the Reputation Service

Listing of unrated user experiments: In order to list unrated experiments the following steps should be followed:

- The tool must retrieve user slices (and related details such as resources etc.) based on the current logged-in user[4];
- The tool must identify experiment details (e.g., grouping the resources of a slice into experiments if needed [Deliverable 7.4 [1]), required in the RS API calls;
- The tool must construct the POST data, based on experiments description, and execute the "/reputation/exp" call;
- Eventually the tool must present the list with unrated experiments to the user.

Submit User Feedback: The tool must provide the user with the appropriate feedback form that will allow him/her to submit evaluation on an unrated experiment to the RS. For this purpose, apart from listing unrated experiments, the tool must implement the following:

- Retrieve the technical and non-technical services that will be evaluated for each testbed involved in the experiment (/reputation/show or /reputation/showrep call);
- Present the user with the appropriate feedback form;
- Construct the POST data including the evaluation and post the ratings to the RS (/reputation/userqoe call).

[4] For this purpose the tool must have access to the respective testbed AMs as depicted in the architecture.

During Cycle 3 jFed has been updated to support the RS. MySlice supports RS since Cycle 2 (Deliverable 7.5 [3]).

4.3.1 Reputation in jFed

In jFed, the experimenter can see the reputation score per testbed when he/she chooses resources. For each testbed (see Figure 9) we have multiple information icons:

- A wrench to indicate if a testbed has planned maintenance
- The star which indicates the reputation score (darker green is better, red is bad)
- The heart which indicates the health of a testbed (darker green is better, red is bad)
- An indication for the available resources in a bar

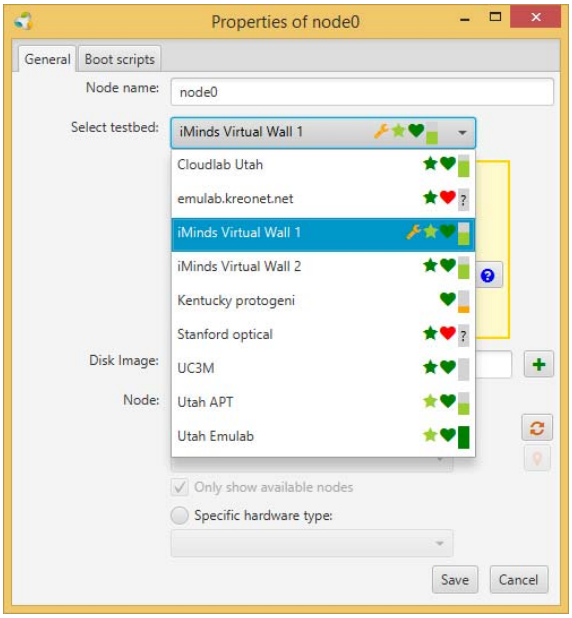

Figure 9: Information icons per testbed

Per testbed, the experiment sees also the detailed scores when a testbed is selected, as depicted in Figure 10.

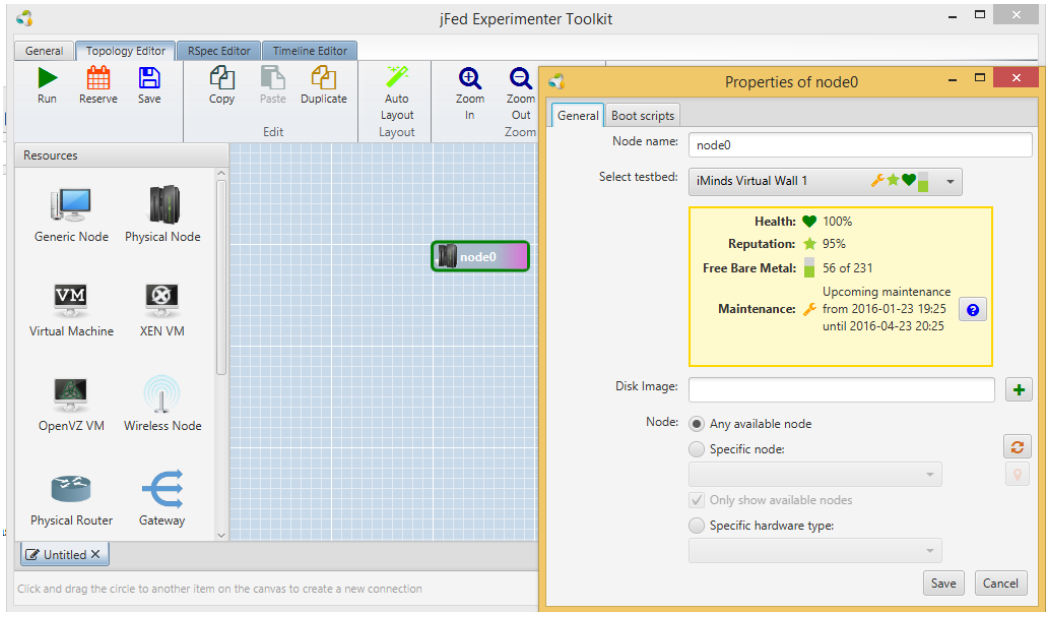

Figure 10: View Reputation Scores

At the end of an experiment, when the user terminates the experiment, he/she has the possibility to score the experiment either on per testbed basis (Figure 11) or not. This info is then sent to the reputation engine. Currently, via jFed we have not yet the possibility to rate older experiments.

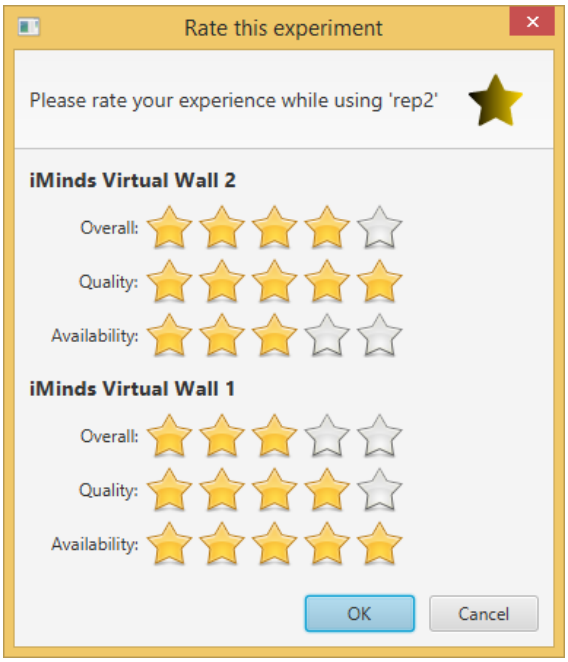

Figure 11: Provide Feedback

5 Conclusions

This deliverable concludes the work in WP7. WP7 has been concerned with the trustworthiness of the federation, and the work conducted within it has been independent but related efforts with the aim of increasing the trustworthiness of the environment offered by Fed4FIRE and the participants within it.

The first WP7 deliverable, D7.1 defined trust as follows:

> "The Oxford English Dictionary defines "Trust" as:
>
> > *"Confidence in or reliance on some quality or attribute of a person or thing, or the truth of a statement."*[5]
>
> The wording of the definition serves to determine our interpretation:
>
> - Firstly, there is the concept of *belief*, denoted by confidence.
> - Secondly there is the concept of *dependence* denoted by "reliance".
> - Thirdly there is the *subject* of the trust (here the "person", the "thing" or the "statement").
> - Finally, there is the concept of a *property*, *feature, behaviour* or *ability* of the subject (denoted by "quality or attribute").
>
> Trust is a belief held by a person or body of people (hereafter called the "trustor"), and is the outcome of a judgement made by them (called the "trust decision") about a property or attribute of another person or body of people (hereafter called the "trustee")."

Given this definition, WP7 has sought to promote trustworthiness in Fed4FIRE, i.e. to give the users and participants of Fed4FIRE greater confidence that Fed4FIRE will perform as expected and that their participation in Fed4FIRE will not endanger them (i.e. reduction of risks to them). The specific contributions made by the tasks in WP7 are listed as follows.

- The Access Control work in T7.2 has contributed trustworthiness mainly in the protection of testbeds, thus increasing their confidence in participating in the federation. Firstly, access to testbed resources is controlled based on Fed4FIRE tokens, and this means that only members of the federation, who have been subject to the checks in the registration process can access the federation's resources. Secondly, all entities within the Fed4FIRE federation are known, so all actions are done by a known entity – discouraging abuse because any perpetrators can be identified. Thirdly, users' access to a testbed is managed based on that testbed's own decisions. This gives the testbeds the final say in whether experimenters can access their resources.
- The SLA work in T7.3 has contributed to the trustworthiness of both experimenters and testbeds. SLAs increase the confidence of both experimenters and testbeds because an SLA is an agreement between both parties that shows each side what is promised and what will happen should it not be delivered. There is thus a clear statement of expectations and what will happen if the expectations are not met. The SLAs performance evaluation is supported by monitoring information that comes directly from the testbeds and compiled by the SLA Management tool located at each testbed's facility. This gives testbed owners important management information on how they are performing in terms of whether they are meeting

[5] http://www.oed.com

their SLA expectations, and if not why not, and this in turn helps them to provide better service to the experimenters.

- The Reputation work in T7.4 contributes to the trustworthiness of testbeds and experimenters by providing an independent platform where experimenters can rate testbeds based on their experience of using the testbeds. The reputation service assists experimenters in selecting testbeds for resource provisioning, based on previous users' experience of the testbeds, helping the users find the most reliable testbeds. Importantly, to avoid users abusing testbeds by providing false negative reports, each user has a "credibility" score, which is based on the truth of their experience reports. This is determined by measuring the testbeds' performance monitoring information, and the user's credibility scaled according to how close the user's report is to the measured performance. Users who make false reports will have their credibility reduced and their reports will have less impact on the testbed reputation scores.

Overall, WP7 has been about reduction of risks to the federation participants and enabling increased reliability, thus increasing the participants' confidence (therefore trust) in their participation in the federation. The mechanisms for this has taken different forms, from increased protection of federation resources to giving federation participants useful information to enable them to make informed trust judgements about their participation in the federation and the resources they want to use.

6 References

[1] Fed4FIRE D7.4, Specification for Third Cycle Developments
[2] Fed4FIRE D4.6, Report on Third Cycle Developments
[3] Fed4FIRE D7.5, Report on Second Cycle Developments
[4] D. Gambetta, ”Can we trust trust?”, in Trust: Making and Breaking Cooperative Relations, D. Gambetta, Basil Blackwell, 2000, pp. 213-237
[5] Jøsang, Audun, Roslan Ismail, and Colin Boyd. "A survey of trust and reputation systems for online service provision." Decision support systems 43.2 (2007): 618-644.
[6] Pawar, Pramod S., et al. "Trust model for optimized cloud services." Trust Management VI. Springer Berlin Heidelberg, 2012. 97-112.
[7] Shen, Xuemin Sherman, et al. Handbook of peer-to-peer networking. Vol. 34. Springer Science & Business Media, 2010.
[8] Fed4FIRE D7.1 – Detailed Specifications for First Cycle

Annex A: SLA Collector API

The SLA Collector is currently being located as a central component at iMinds. Similar to the REST API developed for the SLA Management module, this API will return 200 OK messages, with the requested information if any, when the query has been processed correctly. The responses are returned in JSON format.

GET /testbeds
Retrieves a list with all the testbeds that support SLAs

GET /sla/slice/{sliceId}/{?expirationtime}
Retrieves a list with the SLAs under a specific slice. **Parameters**: • sliceId: urn of a slice. • expirationtime: expiration time of an SLA to filter the output. The date has to be in ISO format, with the '+' sign encoded. Time zone is accepted with and without ':' as separator. Example: ?expirationtime=2016-01-18T11:30:00%2B0200 **Error message**: 404 when the requested slice does not contain any SLA.

POST /agreements/create{?testbed}
Creates a new agreement based on the {testbed} SLA template and redirects the call to its SLA Management module. The POST data have to be provided in JSON format and have to contain the following information: "SLIVER_INFO_AGGREGATE_URN": "testbed_urn", "SLIVER_INFO_EXPIRATION": "date in ISO format: 2014-10-29T16:17:55+01:00", "SLIVER_INFO_SLICE_URN": "slic_urn", "SLIVER_INFO_CREATOR_URN": "experimenter_urn", "SLIVER_INFO_URN": ["list with sliver Ids"], "SLIVER_INFO_SLA_ID": "SLA Id using UUID standard"

[Optional] “SLIVER_INFO_TEMPLATE_ID”: “SLA template Id using UUID standard”. This identifier can be obtained from the SLA Dashboard. If this field is not provided, the first template defined identified by the testbed name will be used.

Parameters:

- testbed: urn of the testbed.

Error message:

400 when the submitted parameters contain any error.

Errors returned from the SLA Management API.

SLA Collector as a Proxy

The following pattern is to use the SLA Collector as a proxy to make a request to a testbeds. This allows any REST operation that the SLA Management module supports.

GET | POST | PUT | DELETE /sla/{SLA Management API call}{?testbed}

Redirects the API call to the SLA Management module of the specified testbed.

Parameters:

- SLA Management API call: request following the described API of the SLA Management module.
- testbed: urn of the testbed.

Annex B – Reputation Service REST API

The RS REST API is implemented using Sinatra. The API, which has been updated in Cycle 3 and is provided in the following, implements the functions that are used by experimenters tools to invoke internal functionalities on the RS (Reputation Computation Engine).

GET /reputation/showrep

Name: Get Reputation Scores

Description:

Retrieves the reputation values for all testbed services.

Example request URI(s):

```
http://survivor.lab.netmode.ntua.gr:4567/reputation/showrep
```

Response Representations:

- 200 OK
- Content-type: application/json

```
Example
[
  {
    "testbed": "urn:publicid:IDN+omf:netmode+authority+am",
    "services": [
      { "overall": 1},
      { "availability": 1}
    ]
  },
(…)
]
```

GET /reputation/show?testbed=<TESTBED_URN>

Name: Get Reputation Scores Per Testbed

Description:

Retrieves the reputation values for all testbed services for testbed {testbed_urn}.

Example request URI(s):

```
http://survivor.lab.netmode.ntua.gr:4567/reputation/show?testbed=urn:publicid:IDN%2bwall2
.ilabt.iminds.be%2bauthority%2bcm
```

Parameters:

- <TESTBED_URN>: The URN of the testbed's component/aggregate manager

Response Representations:

- 200 OK
- Content-type: application/json

```
Example
[
  {
    "testbed": " wall2.ilabt.iminds.be",
    "services": [
      { "overall": 1},
      { "availability": 1}
    ]
  }
]
```

Error message:

400 Testbed not found: when the requested <testbed_urn> is not available in the RS

POST / reputation/exp

Name: Get Unrated Experiments

Description:

Posts a list of experiments and retrieves the unrated ones.

The POST data have to be provided in JSON format and have to contain the following information per experiment:

" user_urn ": The URN of the user conducting the experiment,

"slice_urn": The URN of the slice used to conduct the experiment,

"start_time ": Datetime the experiment started (rfc3339),

"end_time ": Datetime the experiment ended (rfc3339),

"resources": sliver / resource URNs in the slice (depending on the testbed involved).

Acceptable Request Representations:

- Content-type: application/json

```
Example
{
"experiments":[
     {
        "user_urn":"urn:publicid:IDN+fed4fire:global+user+chrisap",
         "slice_urn":"urn:publicid:IDN+fed4fire:global:review+slice+reservation",
         "start_time":"2015-12-07T15:00:00+02:00",
         "end_time":"2015-12-07T15:30:00+02:00",
         "resources":[
         "urn:publicid:IDN+omf:netmode+node+node7",
         "urn:publicid:IDN+omf:netmode+node+node4"
          ]
    },
    {
       "user_urn":"urn:publicid:IDN+fed4fire:global+user+chrisap",
        "slice_urn":"urn:publicid:IDN+fed4fire:global:review+slice+exp1",
        "start_time":"2015-12-06T17:00:00+02:00",
        "end_time":"2015-12-06T19:00:00+02:00",
```

```
        "resources":[
        "urn:publicid:IDN+omf:netmode+node+node9",
        "urn:publicid:IDN+omf:nitos.outdoor+node+node007",
        "urn:publicid:IDN+omf:netmode+node+node10"
         ]
    }
 ]
}
```

Response Representations:

- 200 OK
- Content-type: application/json

```
Example
[
  {
    "user_urn": "urn:publicid:IDN+fed4fire:global+user+chrisap",
    "slice_urn": "urn:publicid:IDN+fed4fire:global:review+slice+exp1",
    "start_time": "2015-12-06T17:00:00+02:00",
    "end_time": "2015-12-06T19:00:00+02:00",
    "resources":[
        "urn:publicid:IDN+omf:netmode+node+node9",
        "urn:publicid:IDN+omf:nitos.outdoor+node+node007",
        "urn:publicid:IDN+omf:netmode+node+node10"
         ]
  }
]
```

Error message:

500 Any error in the submission process.

POST / reputation/userqoe

Name: Submit Rating

Description:

Submits a new score for an unrated experiment.

The POST data have to be provided in JSON format and have to contain the following information per experiment:

"user_urn ": The URN of the user conducting the experiment,

"slice_urn": The URN of the slice used to conduct the experiment,

"start_time ": Datetime the experiment started (rfc3339),

"end_time ": Datetime the experiment ended (rfc3339),

"resources": sliver / resource URNs in the slice (depending on the testbed involved),

"user_eval ": ratings of all services regarding testbeds involved in the experiment

Acceptable Request Representations:

- Content-type: application/json

```
Example
{
"start_time":"2015-12-07T15:00:00+02:00",
"user_urn":"urn:publicid:IDN+fed4fire:global+user+chrisap",
"slice_urn":"urn:publicid:IDN+fed4fire:global:review+slice+reservation",
"end_time":"2015-12-07T15:30:00+02:00",
"resources":[
"urn:publicid:IDN+omf:netmode+node+node7",
"urn:publicid:IDN+omf:netmode+node+node4"],
"user_eval":{"overall":"5","quality":"5", "availability":"5"}
}
```

Response:

- 200 OK : "DATA RECEIVED!"
- Content-type: application/json

Error message:

400: Experiment exists in the RS

500 Any error in the submission process.

POST /testbed/

Name: Add Testbed in the RS

Description:

Registers a new testbed to the Reputation Service.

The POST data have to be provided in JSON format and have to contain the following information:

" testbed": The URN of the testbed's component/aggregate manager,

"alias": Alias for the testbed's component manager,

"services": Testbed's technical services,

"type ": AM or AM API implementation (e.g., EMULAB, NITOS Broker, Foam etc)

Acceptable Request Representations:

- Content-type: application/json
- Authorization: Basic <base64(username:password)>

```
Example
{
  "AM":"urn:publicid:IDN+omf:netmode+authority+am",
  "alias": "NETMODE",
  "services": ["Availability"],
  "type": "NITOS"
}
```

Response Representations:

- 200 OK: "The operation completed successfully."

Content-type: application/json

Error message:

400 Testbed exists or error in Testbed URN

401 Unauthorized

500 Any error in the registration process

PUT /testbed/

Name: Update Testbed in the RS

Description:

Updates testbed info to the Reputation Service.

The POST data have to be provided in JSON format and have to contain the following information:

" testbed": The URN of the testbed's component/aggregate manager,

"alias": Alias for the testbed's component manager,

"services": Testbed's technical services,

"type ": AM or AM API implementation (e.g., EMULAB, NITOS Broker, Foam etc)

Acceptable Request Representations:

- Content-type: application/json
- Authorization: Basic <base64(username:password)>

```
Example
{
  "AM":"urn:publicid:IDN+omf:netmode+authority+am",
  "services": ["link quality"]
}
```

Response Representations:

- 200 OK: "The operation completed successfully."

Content-type: application/json

Error message:

400 Testbed does not exist or error in Testbed URN

401 Unauthorized

500 Any error in the registration process

DELETE /testbed/<TESTBED_URN>

Name: Delete Testbed from the RS

Description:

Remove a testbed from the Reputation Service.

Example request URI(s):

```
http://survivor.lab.netmode.ntua.gr:4567/testbed/urn:publicid:IDN%2bwall2.ilabt.iminds.be%2bauthority%2bcm
```

Parameters:

- <TESTBED_URN>: The URN of the testbed's component/aggregate manager

Response Representations:

- 200 OK: "The operation completed successfully."

Content-type: application/json

Error message:

400 No appropriate testbed URN provided or the testbed is not registered in the reputation service

401 Unauthorized

500 Any error in the deletion process