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**

D6.2 - Detailed specifications regarding monitoring and measurement for second cycle

Work package	WP6
Task	Task 6.1, Task 6.2
Due date	31/01/2014
Submission date	30/04/2014
Deliverable lead	Yahya Al-Hazmi (TUB)
Version	1.0
Authors	Yahya Al-Hazmi (TUB) Wim Vandenberghe (iMinds) Thierry Rakotoarivelo (NICTA) Christoph Dwertmann (NICTA) Loïc Baron (UPMC) Alexander Willner (TUB) Pedro Rey (ATOS) Georgios Androulidakis (NTUA) Chrysa Papagianni (NTUA)
Reviewers	Brecht Vermeulen (iMinds) and Julien Lefeuvre (Inria)

Abstract	This document provides an overview of the measurement and monitoring services in Fed4FIRE and the respective stakeholders. Based on their commonalities, it presents implementations steps for the second development cycle of the project in terms of monitoring and measurement aspects of the Fed4FIRE federation.
Keywords	Measurement, Monitoring, Experimenter, SLA, Reputation, Reservation, OML, Data broker, Manifold, Portal

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 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.

The Fed4FIRE project received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no FP7-ICT-318389.

Executive Summary

This deliverable presents the second cycle implementation of the Fed4FIRE monitoring and measurement covered by Work Package 6. It focuses on infrastructure monitoring service as well as experiment measurements while facility monitoring service is already supported in the first cycle.

In the first cycle of the project, a survey and analysis of the state of the art in terms of i) tools for data acquisition, collection and reporting, ii) on the used tools by the involved testbeds, and iii) monitoring requirements and metrics were performed. This input was consolidated and considered together with all objectives set in multiple Fed4FIRE deliverables targeting the second cycle in order to facilitate building a detailed implementation design for the second cycle.

It was decided from the first cycle to use the OML as a common framework for data collection and reporting. It allows instrumenting any sort of system through the abstraction of a measurement point, describing a group of metrics. This abstraction allows more latitude in the choice of measurement tools: as long as they conform to the same measurement point for the same information, their output can be used indistinctly in further analysis. Selecting OML for reporting purposes therefore allows flexibility in the choice of measurement tools, both for monitoring and measurement tasks, as well as for a unified way to access the collected data.

OML is used only for data collection and storage but will not for direct access to the data or visualize it. Therefore another tool will be used for this purpose such as Manifold which is the core framework of the TopHat tool. It will be used for retrieving the data from the OML server together with its database backend and making it accessible to its corresponding user.

This deliverable covers the specification for the second cycle implementation of the infrastructure monitoring as well as experiment measurement for experimenters, and infrastructure monitoring service for multiple federation services such as SLA management, reputation and reservation.

Overall, infrastructure monitoring data will be collected through OML, persisted PostgreSQL databases, and then accessed by authorised experimenters tools and some federation services. In addition, experimenters can also route filtered copies of their measurement streams into their own OML endpoints, such as OML servers or adaptive experiment controllers. Federation services, in turn, will be able to access infrastructure monitoring data collected through OML at the federation level. It will also be possible for some services, such as the SLA service, to access data from distributed OML servers at the testbed level if testbeds optionally provide such capability.

All testbeds should provide infrastructure monitoring data as OML streams and are therefore requested to deploy the OML framework. Some of them could provide a collection resource (OML server together with PostgreSQL server) to be requested by experimenters as any normal experimental resource and to be used as a collection resource for their data. While the federation services, SLA, reputation and reservation, will use a central collection resource at the federation level. Federation wide solutions should limit the impact on the locally deployed solutions at the

testbeds. Thus monitoring and measurement tools already in use will not be superseded, but rather adapted to be included in the proposed architecture. In case the requirements are not met, default solutions are prescribed.

In order to implement this design, Table 1 presents the implementation steps that have been identified. Most steps in terms of software deployment and instrumentation should be undertaken independently by all participants. Where commonalities exist (e.g. the use of any of the recommended tools "Zabbix, Nagios or collectd" as well as the use of OML) instrumentation should be a common effort. Some implementation efforts are done at the federation level such as the data broker (Manifold) and the central data collection resource.

Functional element	Implementation strategy
Facility Monitoring	• Deploy Nagios and/or Zabbix and/or collectd or any equivalent tool for the same purpose if not yet available (all participants) • Deploy OML if not yet available (all participants with support from NICTA/UTH) • Instrument these relevant measurement systems (all participants, with support from WP6)
Fine-grained infrastructure monitoring for experimenters	• Deploy Nagios and/or Zabbix and/or collectd or any equivalent tool for the same purpose if not yet available (all participants) • Deploy OML if not yet available (all participants with support from NICTA/UTH) • Instrument these relevant measurement systems (all participants, with support from WP6) • Extend the RSpecs for advertising fine-grained infrastructure monitoring capabilities of the offered resources (all participants, with support from TUB) • Extend the Aggregate Manager for fine-grained infrastructure monitoring need (all participants)
Coarse-grained infrastructure monitoring for federation services	• Deploy Nagios and/or Zabbix and/or collectd or any equivalent tool for the same purpose if not yet available (all participants) • Deploy OML if not yet available (all participants with support from NICTA/UTH) • Instrument these relevant measurement systems (all participants, with support from WP6) • Extend the Aggregate Manager for coarse-grained infrastructure monitoring need (all participants)
Experiment measurement	• Deploy OML if not yet available (all participants with support from NICTA/UTH) • Instrument relevant measurement systems (all participants, with support from WP6) • Maintain clearinghouse of measurement points (NICTA)

Measurement service	• Provide a measurement service that is able to provide measurements data exported as OML streams without a need for the experimenter to set up the measurement framework (provided by participants as optional service)
Data collection	• Deploy a collection resource (OML server together with database) for infrastructure monitoring and measurements (optional for all participants, iMinds will provide a central one at the federation)
Data access for multiple stakeholders (experimenters and the reputation and reservation)	• Deploy Manifold at the federation level that acts as a data broker between users and their collection resources (supported by UPMC) • Make OML measurement databases accessible to Manifold (UPMC, NICTA, UTH)

Table 1: Implementation strategy of functional elements

Acronyms and Abbreviations

AM	Aggregate Manager
API	Application Programming Interface
ASN	Autonomous System Number
CM	Chassis Manager
CPU	Central Processing Unit
CSV	Comma-separated values
DB	Data base
DCCP	Datagram Congestion Control Protocol
FLS	First Level Support
FRCP	Federated Resource Control Protocol
GENI	Global Environment for Network Innovations
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
ID	identifier
IP	Internet Protocol
NIC	Network Interface Card
OEDL	OMF Experiment Description Language
OMF	cOntrol and Management Framework: a testbed management framework
OML	Measurement Library: an instrumentation system allowing for remote collection of any software-produced metrics, with in line filtering and multiple SQL back-ends.
OS	Operating System
QoS	Quality of Service
RAM	Random-access memory
RSpec	GENI Resource Specification
Rx	Receiver
SCTP	Stream Control Transmission Protocol
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
SSH	Secure Shell
TCP	Transmission Control Protocol
TDMI	TopHat Dedicated Measurement Infrastructure
Tx	Transmitter
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
xmlrpc	remote procedure call for extensible markup language

Table of Contents

List of Figures

List of Tables

1 Introduction

This deliverable presents the specifications regarding the second cycle development in WP6, based on the second cycle architecture described in D2.4 “Second Federation Architecture”. These specifications describe the actual implementations in terms of tools usage and deployment in order for providing measurement and monitoring services, and as well mechanisms for offering the collected data, for multiple stakeholders: First Level Support, experimenters, and federation services such as SLA, reputation, etc. This document is structured as follows. Section 2 presents specific constrains that WP6 should consider while preparing the specifications of the second cycle of the Fed4FIRE measurement and monitoring architecture. This includes input and requirements from other workpackages (deliverables) as a result of multiple discussions and across-workpackages work on specifications. In addition to this, specifications identified by WP6 are included. Note that Section 2 provides only a brief overview while in-details information is already reported in D6.1. Based on these requirements and input, Section 3 includes evaluations of possible approaches to use for implementation. The adopted ones are described beside a detailed design of the second cycle implementation of Fed4FIRE measurement and monitoring architecture. Note that Section 3 does not provide all evaluations since most of them have been already covered in D6.1. Finally, section 4 concludes this deliverable with an appropriate summary.

2 Inputs to this deliverable

This section revives specific information from some previously submitted Fed4FIRE deliverables. The input includes requirements and as well specifications in terms of monitoring and measurement that are considered when defining the second cycle development of the Fed4FIRE measurement and monitoring Architecture.

2.1 Architecture (D2.4)

Fed4FIRE identified the following types of monitoring and measurement (Figure 1) in D2.4 "Second Federation Architecture":

- **Facility monitoring**: this type of monitoring is used in the First Level Support (FLS) to see if the testbed facilities are still up and running. It is based on monitoring the key components of each testbed facility. The availability status of the key components of a particular facility are reported to a central location at the federation level that calculates the overall status of that testbed facility and represents its GAR (Green, Amber, or Red) status to the FLS.
- **Infrastructure monitoring:** this type of monitoring is about monitoring the facility infrastructure which is useful for stakeholders mainly the experimenters and some federation services such as Service Level Agreement (SLA) service, reputation service, and probably others. Examples of monitoring information of experimenters' interests could be: monitoring of switch traffic, wireless spectrum, and physical host performance if the experimenter uses virtual machines. For SLA management service, information about specific, predefined metrics required to validate SLAs is required. Compared to facility monitoring, infrastructure monitoring is characterised by a finer granularity, providing information about specific resources instead of the facility as a whole. Compared to experiment measuring, infrastructure monitoring is characterised by the fact that this concerns data that an experimenter or other federation service could not collect himself.
- **Experiment measuring:** this type represents measurements that are done through the use of measurement frameworks or tools that the experimenter uses, and can be deployed by the experimenter himself. Of course, a testbed provider can ease this by providing for instance deployable OS images with certain frameworks, but this is optional. The only hard requirement on testbeds supporting experiment measuring is the fact that they should support the export of such measurement data as an OML stream (as defined in D2.4 "Second Federation Architecture"). This information in D2.4 is considered to be sufficient to allow the implementation of this type of monitoring. No further specification seems to be needed; therefore experiment measuring is considered to be out of the scope of this deliverable.

This deliverable also stated that the infrastructure monitoring and experiment measuring will be provided in the second cycle of Fed4FIRE, besides the facility monitoring that was already rolled out on all testbeds in the first cycle. The corresponding architecture as defined in D2.4 is depicted in Figure 1 and Figure 2.

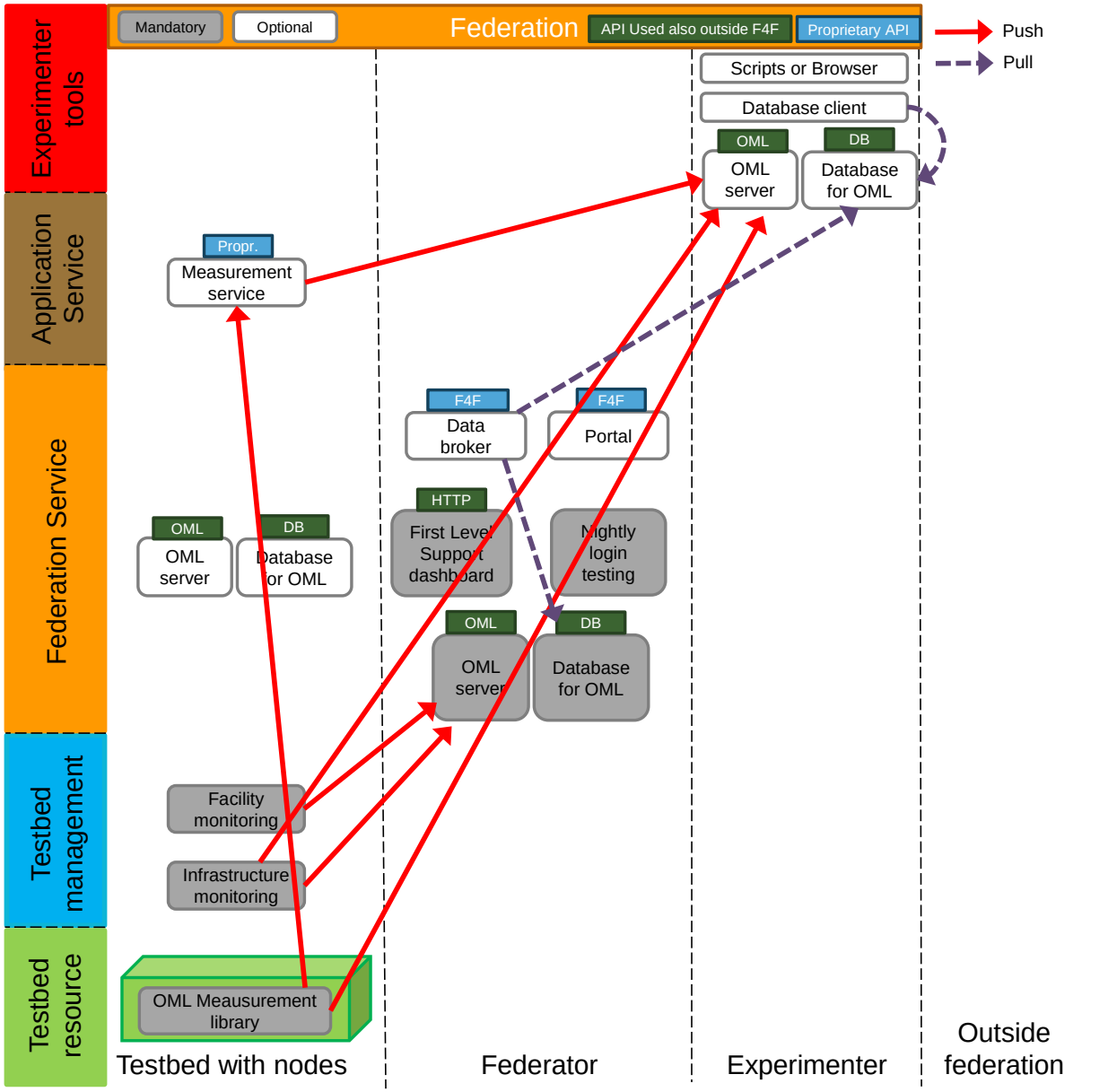

Figure 1: Monitoring and measurement architecture for cycle 2

Figure 1 illustrates the individual architectural components of the monitoring and measurement solution in the second cycle and their interactions. In the following we describe the individual architectural components at multiple levels that are required to realise the three types of monitoring and measurements discussed above.

At the **testbed** side, the following components are to be supported:

- **Facility monitoring**: The testbed has the freedom to adopt any solution to gather this type of monitoring data as it sees fit (e.g. an existing monitoring framework such as Zabbix, Nagios or similar), as long as it is able to export that data as an OML stream to the Federator's central OML server, which will store it in a database for FLS. The OML framework [9] is used as a common interface to provide monitoring data as OML streams. In the first cycle of Fed4FIRE, the facility monitoring was rolled out on all testbeds. More high-level details are

given in D2.4, section 3.4.2, and the corresponding specifications are included in Appendix C of this deliverable.

- **Infrastructure monitoring:** Instrumentation of infrastructure resources by the testbed provider itself to collect data on the behaviour and performance of resources, services, technologies, and protocols. This allows the stakeholder (mainly the experimenters and some federation services such as Service Level Agreement (SLA) service, reputation service, reservation system) to obtain monitoring information about the used resources that he could not collect himself.
    - **Infrastructure monitoring for federation services:** These services will get on the other hand coarse-grained infrastructure monitoring information as also illustrated in Figure 2. For SLA management service for instance, information about specific, predefined metrics required to validate SLAs is required.
    - **Infrastructure monitoring for experimenters:** Experimenters will get fine-grained infrastructure monitoring information. Examples of such infrastructure monitoring data of experimenters' concern during the course of the experiment are monitoring data of switch traffic, wireless spectrum, and the performance and NIC congestion of the physical host that hosts virtual machine resource.

  Experimenters should have the ability to get monitoring data through a common interface provided by all testbed facilities. The OML framework is used to provide a common interface for monitoring data. Monitoring data are then collected as OML streams. Infrastructure monitoring service will be provided in the second cycle of Fed4FIRE. More details on the implementations of this type of monitoring are discussed in section 3.
- **Experiment monitoring:** The **OML measurement library** is available to experimenters to instrument their own component or tools provided by the community. Control of what to measure and at what granularity is entirely in the hands of the experimenter. The OML library and supporting infrastructure cleanly separates the ability of the component or tool developer to provide numerous measurement points from the requirements of the experimenter for a particular experiment. Self-instrumentation of the OML framework will provide the experimenter with insight on the potential impact of collecting measurements. The experimenter is also free to choose the destination of the requested measurement streams. This could be an OML server (as discussed next) or other components which can use the measurement streams to observe and steer the experiment or to give visual feedback to the experimenter on the ongoing experiment.
- for experiment measuring**:** this component is intended for measurements which are done by a framework that the experimenter uses and which can be deployed by the experimenter itself on his testbed resources in his experiment. The experimenter can retrieve or calculate this data as he prefers, and can then use the OML measurement library (which should be available on every resource) to easily export it to an OML server with database for storage.
- **OML server**: this component can be configured to be the endpoint of a monitoring or measurement OML stream, and can persist the streams in several types of **databases** (PostgresQL, SQLite3). The deployment of an OML server by a testbed provider for the use of experimenters is optional. In the case of a 'public' measurement repository additional care needs to be taken to control access to the data. This can be achieved through the native access control of the database server, or through some proxy service, such as Manifold.
- The testbed provider that provided an OML server can decide to make this data easy to retrieve for the experimenter by exposing it as a **measurement service** using a proprietary interface. The deployment of such a measurement service is optional.

When focusing on the **Federator**, the following elements of Figure 1 require some further introduction:

- The **FLS dashboard** gives a real-time, comprehensive but also very compact overview of the health status of the different testbeds included in the Fed4FIRE federation. To determine this health status it combines facility monitoring information provided by the testbeds with specific measurements performed by the dashboard component itself. More details about this are given in D2.4 in section 3.4.2. The existence of the FLS dashboard at the Federator level is in line with the project's approach that Federator components should be put in place for convenience, but shouldn't be critical for the federation's operation. The dashboard is indeed a very useful tool to have, but if its operation would be disrupted or even discontinued, experimenters would still be just as able to get resources and work on them as when the dashboard was still supported. Also, since it is a stateless component (it combines data from the different testbeds in a single health overview), it can easily be duplicated or moved. This would only result in a loss of historical information about the federation's health, but no functionality loss would occur.
- The federator provides an **OML server and corresponding database for FLS data** to process and store facility monitoring data of the testbeds to be used by the First Level Support (FLS). These two Federator components are a necessity to implement the FLS dashboard, but since the dashboard itself is not critical to the federation, neither are the central OML server and corresponding database. This motivates that the Federation level is a suitable place for these components.
- The component for **nightly login testing** provides a second view on the operational status of the federation. Its information is not real-time (typically tests would automatically be performed once or twice a day), but the result of these tests is more thorough then those of the FLS dashboard. This is because in this case the testing module performs an actual experiment (including all steps of the experimental lifecycle related to resource discovery, reservation and provisioning, ending with an actual automatic SSH login on the provisioned resources) and tracks success or issues for any intermediary step of the experiment lifecycle. According to exactly the same principles as for the FLS dashboard, the component for nightly login testing is in line with the approach that Federator components should not be critical for the operation of the federation (disruption in the nightly login testing does not hinder the execution of experiments, this testing could be easily duplicated or moved).
- The **data broker** is an optional component that can be **accessed at the federation level**, and which makes it easier for novice experimenters to retrieve their experiment data from the different sources where it might reside (OML servers of the different testbeds that provided infrastructure monitoring, OML servers of the experimenter itself on which the experiment measurements were stored, etc.). According to exactly the same principles as for the FLS dashboard and the component for nightly login testing, the data broker is in line with the approach that Federator components should not be critical for the operation of the federation (disruption in the data broker service does not hinder the direct retrieval of experiment data, the broker service could be easily duplicated or moved). The data broker will be implemented by using the Manifold framework [7]. More details on requirements and implementations are discussed in sections 2.2.3 and 3.4.2 respectively.

The **experimenter** itself can also utilize different experimenter tools:

- A **data broker** provides an API, allowing users to query measurements and monitoring data that is already stored in their OML servers. The experimenters can then use any experimenter tools (Python **script**, ODEL **script**, **browser**, etc.) to send queries to that API.
- A **database client** allows the experimenter to retrieve his monitoring and measuring data from any OML server where it was stored

- For storing infrastructure and measurement data, the experimenter has the option to deploy his own **OML server and attached database server** in order to archive the data himself.

Figure 2 shows the relation between the monitoring and measurement architecture and the SLA, reputation and reservation services as defined in cycle 2. Facility monitoring data as well as infrastructure monitoring data are pushed by the testbed provider to both services SLA and reputation per experiment basis, while historical based data is pushed to the reservation service
The SLA management service will be deployed in a distributed manner in the Fed4FIRE architecture, in a matter of fact an **SLA management module** instance running on each testbed facility, will get monitoring data per experiment basis pushed as OML streams.
The reputation and reservation services will be deployed in a central manner at the federation level.
The **reputation engine** as well as the reservation broker will use the **Manifold** framework that acts as a **data broker** in this matter to retrieve monitoring data from a central collection point (**OML server and database**) located at the federation level that collects monitoring data pushed from the individual testbed providers as OML streams. The Manifold is used not only because of its effective role in facilitating the process but also due to its potential in allowing its user to fetch its data from multiple and distributed data sources. This feature is not tangible in the second implementation cycle since the concerned monitoring is collected in a central collection point, but will urgently be needed in case the data is not provided in a central location but rather distributed.

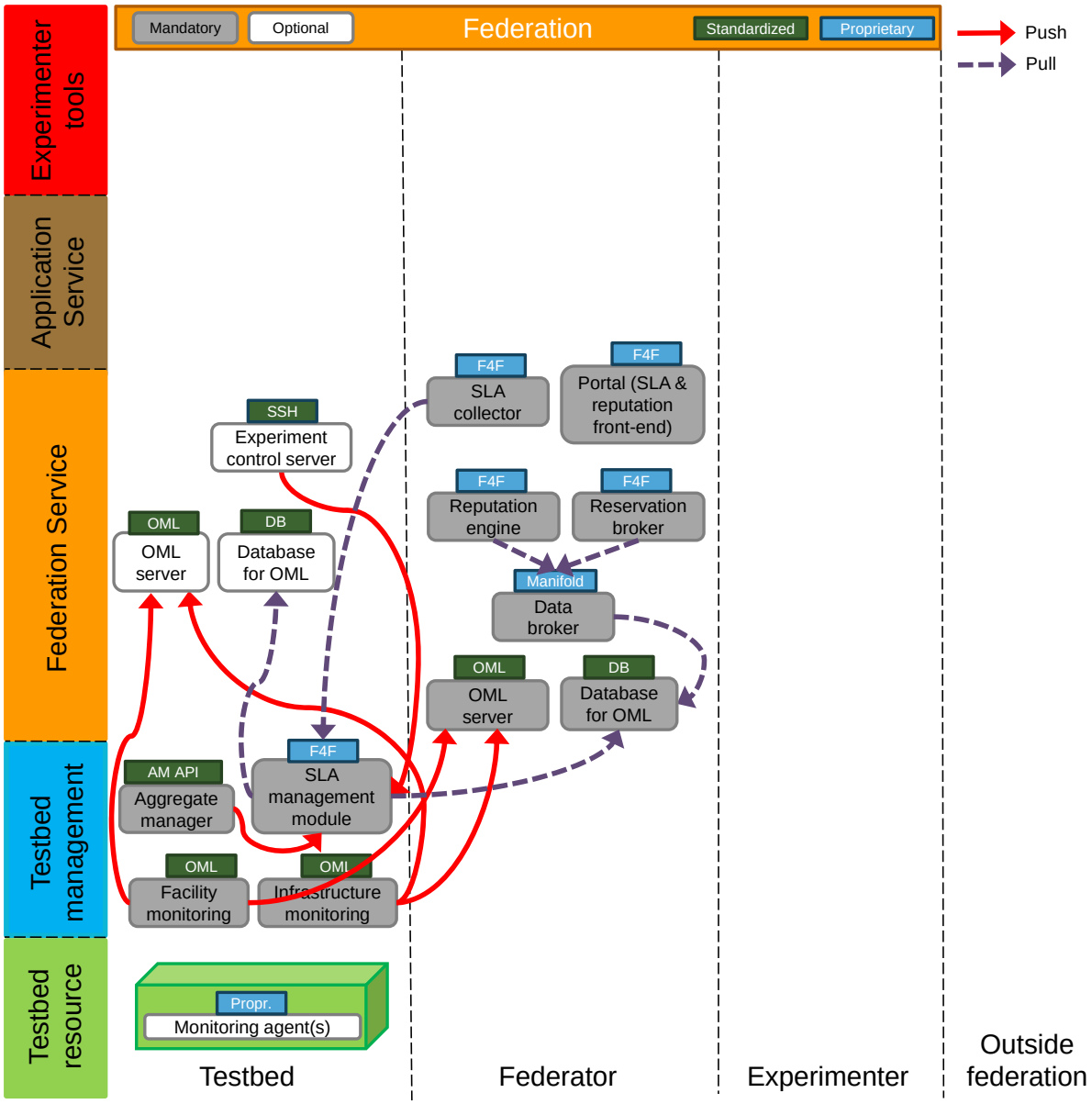

Figure 2: Relation between the monitoring and measurement architecture and the SLA, reputation and future reservation mechanisms as defined in cycle 2.

2.2 Requirements imposed by the other work packages

This section recalls the requirements relevant to WP6 set forth in D3.2, D4.2 and D8.4. It also lists specific new WP6 requirements originating from the planned cycle 2 developments in the other vertical work packages of the project, which are responsible for the development of the Fed4FIRE federation framework (i.e. WP5 and WP7)

2.2.1 High priority requirements of the infrastructure community (D3.2)

Req. id	Req. statement	Req. description	Comments
I.2.101	Measurement support framework	Fed4FIRE must provide an easy way for experimenters to store measures during the experiment runtime for later analysis. The data should be clearly correlated to the experiment ID.	Measurements can be related to common metrics for which existing tools such as ping or iperf can be used. However, they can also be very specific to the experiment, and hence calculated somewhere within the experimental software under test. It should be possible to take measurements on a large variety of resources: Linux servers/embedded devices, OpenFlow packet switches and optical devices, cellular base stations, etc.
I.2.102	Automatic measurement of common metrics	Common characteristics should be stored automatically during an experiment (CPU load, free RAM, Tx/Rx errors, etc.)	.
I.2.103	Wireless interference	Information about external wireless interference during the execution of the experiment should be provided.	The interference can be detected using monitor interfaces in dedicated nodes and/or spectrum analysers that offer more exact results. This functionality should be easily provided to every experimenter who is not “spectrum analysing” expert.
I.2.105	Monitoring resources for suitable resource selection and measurement interpretation	Fed4FIRE must also provide the monitoring info regarding the state of the resources to the experimenters. This way they can choose the best resources for their experiments. This information also provides the experimenters with the means to distinguish error introduced by the experiment from errors related to the infrastructure.	In this monitor view that an experimenter has on his/her resources, it could also be interesting to display some non-monitored background information, for instance the IP address of the control interface, the DNS name, etc.
I.2.106	Minimal impact of monitoring and measuring tools	As less overhead as possible should be expected from the monitoring and measurement support frameworks. The impact of the measurement tools over the experiment results should be negligible.	
I.2.107	On-demand measurements	The user must be able to request on-demand measurements. In order to do so, they will need to express that they want agents	The same information can be retrieved by looking into the output of the monitoring and measurement tools that will continuously provide

Req. id	Req. statement	Req. description	Comments
		with such on-demand polling capacities	measurements during the experiment run-time. However the on-demand measurement is more convenient during experiment development and debugging.
I.2.202	Data security	Access to the data should be properly secured	

2.2.2 High priority requirements of the services community (D4.2)

Req. id	Req. statement	Req. desc.	Comments	Justification of Requirement
ST.2.007	Real-time aggregated monitoring management control	Fed4FIRE must provide tools to create, view, update, and terminate monitoring configurations related to shared resource types or experiments in real time. Monitoring data should be reportable for visualisation and analysis purposes with several reporting strategies (only alarms, all data, filters, etc.) in real time in order to provide accurate information and ease the analysis process. The experimenter might create own aggregated / composite monitored elements out of the available ones when designing the experiment, deciding what to monitor, defining some filtering possibilities as well, and providing the destination endpoint to send the information to. Monitoring information must cover information from different facilities and services. Monitoring metrics should be compatible across different facilities. For	As most of the monitoring in cycle 1 is done by using OML streams, experimenters should be able to create and/or configure new/existing OML Measurement Points in real time (maybe via FRCP)	Access to experiment monitoring should be defined by the experimenter and not by the testbeds themselves. This could be extended to infrastructure monitoring in the context of a experiment. Experimenters should be able to select the metrics they need to monitor, so they can get information that is useful and meaningful to them. Give the experimenters the basic monitoring metrics and the ability to aggregate them. Particular resources may need specific control & configuration interfaces to unburden an experimenter from the need to know about detailed testbed infrastructure capacities and architecture. Additionally, due to differences in the equipment a resource having identical functional features may have different control&configuration capacities. Experimenters need to be informed quickly of any notifications - e.g. breakdowns or errors, so they can react to them

Req. id	Req. statement	Req. desc.	Comments	Justification of Requirement
		example, Monitoring computing & network resources' capacity. The monitored data during an experiment runtime will be available to the experimenter for all components involved in the experiment. If not aggregated, at least monitoring information from all involved testbeds must be provided. Testbeds must be able to publish infrastructure status through an API		

2.2.3 Portal requirements (WP5)

2.2.3.1 Portal requirements

The goal of the Fed4FIRE portal with respect to measurements and monitoring (M&M) is to allow experimenters to query data from various sources through a dedicated API in cycle 2. In further cycles, the Fed4Fire portal will offer to visualize measurements and navigate among them through a user-friendly Web GUI.

This brings in the federation a set of additional requirements:

- **Query language / API:** the federation needs to provide a rich API allowing to efficiently query a wide range of data sources described below. This query language needs to be fine grained enough so that only appropriate subsets of available data can be requested (for the sake of speed, scalability and reactivity of the service offered to the user). The SFA API is not fine-grained enough for the measurement aspects.

- **Data model**: SFA uses records to model static objects in the federation (resources, users, slices, authorities and eventually projects); they are stored in the registry and their number would typically be limited. For more interactive and heterogeneous data, the data model

proposed is based on the Resources Specifications (RSpecs). This format is not appropriate for our measurement context since it lacks many features such as links between objects or hierarchy, the use of XML makes it heavy to process and parse; hard to extend, aggregate or combine; and not machine-readable enough (due to coupled syntax and semantic, although this aspect might improve in the future with the planned support for ontologies).

For these reasons, we believe the Manifold framework, which is the native underlying architecture of the portal, represents a better alternative to provide a measurement plane on which to build portal services for example (and many others). It is in addition well integrated with technologies already in used in the Fed4FIRE architecture, such as SFA and associated authentication and authorisation functions.

2.2.3.2 Data sources

A wide range of data sources of interest for the Fed4FIRE federation have been identified, covering the facility, infrastructure and experimental aspects as described in Section 2.1. They range from dedicated measurement and monitoring platforms available in Fed4FIRE, such as:

1) **SFA:** provides data regarding users, authorities, slices and resources. These data objects identify experiments and their characteristics.
2) **One or several OML servers:** OML is typically used to instrument an experiment and collect measurements produced by it. It is used in the Fed4FIRE context to collect infrastructure monitoring (CPU load, memory, spectrum using tools such as Nagios and Zabbix) and facility monitoring (information for First Level Support about the availability of testbeds).
3) **Maxmind, and other webservices:** can be used to enrich the measurement data.

This list is voluntarily not exhaustive since information of interest (for example to select nodes of interest for building up a slice with originate both from these data sources, but also from other third party sources such as Internet web services (MaxMind, Team Cymru IP-to-ASN mapping service, etc.) or measurement infrastructures (ETOMIC, SONoMA, etc.). We focus on OML since this technology is at the core of the developments regarding measurements and monitoring performed within the project.

2.2.3.3 Combining multiple data sources

Since services such as measurable characteristics of resources, or usage monitoring require this M&M information to be mapped with such objects as user, slices, authorities or resources, which are available through SFA, we will in addition require a tight integration of all these technologies.

Manifold allows users to express queries over heterogeneous sources of data and aggregate results based on common key attributes.

While SFA is already mapped with Manifold for services already offered in the portal (user registration, slice request and resource allocation), mapping of OML DB will be developed in Cycle 2.

2.2.4 Service Level Agreement requirement (WP7)

The SLA service requires infrastructure and facility monitoring information about the provided resources on a per experiment basis in order to ensure that the agreements are met. There is no need for mechanisms which automatically announce infrastructure monitoring service capabilities. Predefined metrics are to be agreed between the testbed providers and the SLA Management. Based on the SLA options offered by the testbed providers and adopted by the SLA solution, testbed providers should measure appropriate metrics that represent information of the used resources for SLA demand. The SLA management service in Fed4FIRE is going to be deployed in a distributed manner. At each testbed, an SLA management instance will be deployed. It requires to access monitoring data locally from any database (e.g. SQL) that provides an API.

SLA management needs monitoring data regarding the period between the experiment's start and its end. Therefore, at the testbed level, the SLA management should be notified about the start and stop times by the AM. This is out of the scope of the WP6 activities, which only look into the needed mechanisms to provide the needed monitoring data to the SLA system. The remainder of the specifications regarding SLAs are provided by WP7.

2.2.5 Reputation service requirement (WP7)

The reputation service needs to collect infrastructure monitoring information per experiment/slice from a central location in order to compare this data with the feedback that the experimenter provides at the end of his/her experiment. The feedback form will include technical questions such as: were all the nodes available during the experiment, was the network ok, etc. In addition, specific questions can be asked for a particular set of services/resources in a testbed as long as the testbed can provide the corresponding measurements in order to evaluate the opinion of the experimenter (expressed by his/her answer in the question). Therefore, the reputation service needs from the testbed providers to provide the corresponding infrastructure monitoring data about the particular services/resources that were used during an experiment in a per experiment/slice basis.
With the use of the user feedback after the experiments and the corresponding monitoring data (in addition to the SLA information), a reputation score for each service is generated (and updated after every user feedback evaluation). Examples of these services can be the uptime of the requested nodes, wireless interference, etc. Services are defined by each testbed that should then provide the corresponding monitoring data for each service in a per experiment basis.
Some of the measured metrics might be the same as in the SLA service, but in general we cannot assume a one-to-one pairing. For example a wireless testbed may want to evaluate how users experience interference during their experiments, but it may not be willing to provide an SLA for that. Therefore, reputation service needs infrastructure monitoring data regarding interference, but SLA service does not.

Note that WP6 only looks into the needed mechanisms to provide the needed monitoring data to the reputation system. The remainder of the specifications regarding the reputation system are provided by WP7.

2.2.6 Reservation broker requirement (WP7)

The reservation brokering service is set out to provide experimenters with the ability to create slices of heterogeneous resources based on a multiplicity of selection criteria (time, type, etc.), via the Fed4FIRE portal, that belong to a wide and varied choice of testbeds within the Fed4FIRE federation environment. User requests for resources that contain full/partial or no mapping information between requested and physical resources must be supported by the particular service.
Towards this end, the reservation broker will leverage:

- Infrastructure monitoring information of physical resources on a per testbed basis, over a predefined time window. This information should be stored in a central location, to be retrieved by the data broker, as specified in D5.2. Examples of metrics that provide infrastructure monitoring information required by the reservation broker in order to efficiently map the resources requested by the experimenter include the number of free resources as well as physical machine specific metrics such as CPU load, RAM consumption, etc.
- Resource leases allowing for advance reservation of resources.

Information as such is deemed necessary to allocate/schedule resources in heterogeneous environments in an intelligent and efficient way that will ensure fairness among testbeds.
Required monitoring metrics are to be agreed between the testbed providers and the reservation broker developers. It is therefore out of the scope of this deliverable, since its focus lies on providing the solution but not concrete lists of metrics. Testbed providers should measure appropriate metrics that represent information regarding the availability and utilization of their resources made available within the Fed4FIRE federation.

2.3 Additional WP6 requirements

This section defines some additional requirements in the context of WP6. It first introduces generic requirements that are fundamental from WP6 viewpoint in the second cycle, and then information about the required tools at the testbed level that provides measurements and monitoring data.

2.3.1 Generic requirements

Infrastructure monitoring service will be provided in the second cycle. There are no additional requirements from WP6 to provide coarse -grained infrastructure monitoring information for the federation services. In contrast, in order to provide fine-grained infrastructure monitoring for experimenters there is still additional requirements from WP6. The infrastructure monitoring capabilities of each offered resource should be first announced to experimenters. To give an example

on this, given that a testbed offers a resource from type VM, the respective infrastructure monitoring capabilities such as fine-grained information about the physical machine hosting that VM should be made available to experimenters. An experimenter while requesting any kind of resource (e.g. VM), he should be able to see, and chose whether or not to have, the respective fine-grained infrastructure monitoring information. The experimenter should provide an endpoint where monitoring data should be then pushed to.

The experimenter's request is passed to the respective testbed that processes the request, and if infrastructure monitoring is required, it initiates a specific monitoring resource (we will refer to in this document with a Wrapper) that is in charge of collecting the respective infrastructure monitoring data from local monitoring tools responsible for monitoring the infrastructure resources, and push data to the experimenter.

Monitoring data should be collected from the individual testbeds on which the experimenter has resources and provided in a common representation, using Query language.

2.3.2 Data broker

The data broker is one of the important components at the federation level that facilitates several services such as reputation and future reservation. It will allow then to retrieve their data from the central collection resource (OML server together with its database). Furthermore, the data broker will be used by the portal to enable the experimenters to access their data from their own collection resources in a user-friendly manner. In Fed4FIRE, the Manifold [7] is used as a reference implementation for the data broker.

2.3.2.1 What is Manifold

Manifold aims at retrieving, combining, and presenting data queried by the user. A user query may involve several sources of data. To do this, Manifold requires to know what sources of data it can query, and what data is available (metadata). Then, Manifold computes a query plan serving the user query. So far, Manifold requires metadata related to each source of data, access information (url, port, etc.) and network connectivity to reach them.

2.3.2.2 Why do we use Manifold

Manifold provides a flexible and generic framework allowing to easily expose and combine heterogeneous data involving one or more sources. Data combination is deduced by Manifold according to the metadata it collects. For instance, Manifold could combine data provided by SFA and OML on a per-experimental basis.

By design, Manifold allows extensions (e.g. adding new data sources or provide new visualization plugins). It already supports various data format such as generic PostgreSQL and CSV data sources. One may develop additional Gateways to support its own data format. In the same way, one can

develop additional plugins to improve data visualization. This will be detailed in the implementation section.

2.3.2.3 Requirements

Manifold requires data connectivity to remote data sources and network information to connect to each data sources (url, port, etc.). We will detail in the implementation section those requirements. Moreover, a user can only access the data he/she owns. Manifold thus requires to transport or to manage user credential, used to interact with the data sources (platforms). In practice, Manifold can manage a set of users (in our case, it will correspond to the Fed4FIRE Web-portal users). Each user can configure a set of accounts, each of them allowing to query a given platform or testbed.

Note that Manifold does not manage grants regarding each data source, since each of them could be accessed in another way. In other words, each data source is responsible to accept or deny incoming query according to the authentication presented by Manifold.

Manifold can present to a data source a global account (if this data source can be accessed by using a global account, for example an anonymous account) or the credentials corresponding to the data source of the connected user. To achieve this, Manifold requires that experimenters configure their dedicated accounts (for instance through the WebGUI) for each data source involved in their queries.

2.3.3 Testbeds' measurement and monitoring tools

A complete and concrete list of monitoring and measurement tools and solutions is already covered in D6.1 in Section 2.3.3 Consolidated summary of testbeds' inputs.

3 Implementation of the architectural functional elements

3.1 Introduction

In this section we discuss every functional element of the monitoring and measurement architecture in the second cycle of Fed4FIRE represented in Figure 1. Facility monitoring service is already supported from the first implementation cycle. In this deliverable we will focus on the infrastructure monitoring service. However, the experiment measuring service is left to the experimenters to deploy whatever tools or framework themselves; some testbed providers could provide their own solutions to enable experiment measurements.

Some implementation decisions have been made in the first cycle implementation and reported in D6.1. Additional required implementation solutions are to be discussed in this document that is necessary for the second cycle. It is possible that an available piece of software or tool will be used as a starting point, or a combination of such software. It is also possible that some elements will be implemented from scratch.

3.2 Monitoring and measuring tools already selected from the first cycle of Fed4FIRE

Measurements usually go through multiple processes but the backbone of the three monitoring and measurement services provided by Fed4FIRE comprises two fundamental ones: (i) obtaining the readings, and (ii) making them accessible to the relevant stakeholders. A detailed review on the state of the art in these two areas has been reported in D6.1 “Detailed specifications for first cycle ready”. For obtaining the readings (also referred to with data acquisition), various tools are reported in the deliverable D6.1. Table 2 (Table 5 in D6.1) and Table 3 (Table 6 in D6.1) include consolidated data on facility and infrastructure monitoring tools, and as well measurement tools respectively. Moreover, Table 4 (Table 7 in D6.1) covers various systems for data collection and reporting.

Tool	Advantages	Disadvantages	Selected as initial approach
OMF/OCF, CM cards	• Facility monitoring	• Limited information	
Nagios	• Alert management • Plugin support • Plugin for historical infrastructure monitoring (but RRDTool) • Already deployed in 33 testbeds	• Ad hoc storage (but SQL export scripts available)	X
Zabbix	• Supports both Facility and Infrastructure Monitoring • Alert management • SQL storage • Plugin support • VM monitoring • Agent-less monitoring • SNMP support • Support for remote collection to centralised server • Already deployed in 44 testbeds	• SQL database can become huge and unresponsive in certain cases • Not always very intuitive	X
Zenoss	• Support Nagios plugins		
Munin, Cacti	• Good for infrastructure monitoring	• No facility monitoring • RRDTool backend (loss of resolution on old data)	
Collectd	• Plugin support • libvirt for VM monitoring • OML writer • SNMP support • Support for remote collection to centralised server	• Need local (or SNMP) agent	X
nmetrics	• OML-instrumented application available • Good for lightweight infrastructure monitoring	• Library, but not stand-alone application • Limited monitored metrics • No remote reporting	
TopHat / TDMI / MySlice	• Infrastructure (network, flows) monitoring • Federated	• Tophat: only running above TeamCumry, Maxmind, TDMI	X

Tool	Advantages	Disadvantages	Selected as initial approach
	• Support for external queries	• MySlice : only running above TopHat, SFA	
DIMES	• Allows measurement of the live Internet	• Not deployed in any of the involved testbeds	
PlanetFlow	• Infrastructure (flows) monitoring • Already deployed in 2 testbeds • Support for external queries	• Only deployed on PlanetLab	
CoMon	• Infrastructure monitoring • Support for external queries	• Currently not maintained	
Observium	• RRD • SNMP based • CollectD integration • IPMI integration • Minimal install effort • App. Monitoring (direct or via CollectD) • Ldap authentication	• Adding non-SNMP devices not supported • Monitoring data in RRD only	
dstat	• Low level info available • Flexible • Well-known tool	• Granularity over time (limited to 1/s)	

Table 2: Consolidated data on facility and infrastructure monitoring tools

Tool	Advantages	Disadvantages
Iperf	• Well known tool • OML instrumentation • TCP, UDP support • DCCP, SCTP in some flavours	• No unified output (by default) • No remote reporting (by default) • Segmented codebase
D-ITG	• OML instrumentation • Different traffic profiles • TCP, UDP, DCCP support	• Requires precise node synchronsation
OTG	• OML instrumentation • Different traffic profiles • Modular • TCP, UDP support	• No DCCP nor SCTP support

Tcpdump	• Well known tool	• No unified output, can differ strongly based on the chosen options. • No remote reporting
libtrace	• OML instrumentation available • Radiotap support	• Not standalone tool
DAG3	• Fast processing	• No unified output • No remote reporting
NetFPGA	• Fast processing	• No unified output • No remote reporting
Multihop Packet Tracking	• Detailed hop-by-hop metrics like delay and loss (traffic engineering) • Environment conditions like cross-traffic • Tracking single packets through the network • Hash-based packet selection technique • Export of measurement results with IPFIX. • Visualization of measurement results.	• Cannot measure passively if there is no traffic. • Hash calculation and measurement export requires resources • Time synchronisation of nodes for delay measurements required
PlanetFlow	• TCP, UDP, ICMP support • Well known data format (silk format) • Netflow query system • Fast and extensive querying faciliyes • Web GUI access.	• Only deployed on PlanetLab

Table 3: Measurement tools

Tool	Advantages	Disadvantages	Selected as final approach
SNMP	• Standard measurement systems • Unified reporting	• No remote reporting	
DTrace	• Dynamic instrumentation of live applications • Default installed on some OS	• Limited to aggregating functions • No remote reporting	
IPFIX	• Unified reporting • Remote reporting	• Limited representable information (but extensible)	

Netcat	• Simple to understand (text over tunnel) • Simple to use (can pipe output of other tools into it on the command line)	• Limited functionality (e.g. no persistence to a database as part of Netcat)	
OML	• Unified reporting • Centralized reporting • Already deployed or planned in 5 testbeds • In-line filtering	• Only reporting and collection: no measurement	X
MINER	• Similar in functionality as OML	• Not open source, hence less extensible	
TopHat	• Allows federation of data sources • Can be plugged into the Fed4FIRE portal (which uses MySlice technology)	• Not intended for collecting of data on the level of the actual resources.	X

Table 4: Collection and reporting systems

D6.1 includes evaluation of multiple solutions and as well those approaches that have been already selected from the first Fed4FIRE cycle. Some testbeds have their solutions that can either completely or partially support capabilities of the adopted approaches. They can retain their solution, but will need to ensure its integration into the rest of the Fed4FIRE federation architecture.

There is a wide variety in the tools deployed on the various testbeds. A few commonalities in the monitoring solutions were already identified, where Zabbix [1] and Nagios [2] are primarily used. However, Zabbix natively caters for both facility and infrastructure monitoring services, while Nagios only supports the former. Facility and infrastructure monitoring services require specific metrics to always be made available about the testbed and its resources. While some deployments already have solutions in place, the most indicated ones for others are, in order of preference, Zabbix, Nagios or Collectd [3]. In order for providing monitoring data for its stakeholders in a common way, it was identified that the most widespread commonality is the use of OML [4] as a collection and reporting framework. It allows instrumenting any sort of system through the abstraction of a measurement point (MP), describing a group of metrics. This abstraction allows more latitude in the choice of measurement tools: as long as they conform to the same measurement point for the same information, their output can be used indistinctly in further analysis. Selecting OML for reporting purposes therefore allows flexibility in the choice of measurement tools, both for monitoring and measurement services, as well as for a unified way to access the collected data. Note that as part of the implementation process of cycle 1, WP6 has provided example scripts to instrument both Zabbix and Nagios monitoring frameworks with OML support. These scripts are collected in Appendix B of this deliverable.

On the experiment measuring service the variability shows the most, with a lot of different and sometimes ad hoc tools. This disparity can be solved through the use of OML as a middleware measurement system in charge of reporting samples from heterogeneous distributed tools in a unified and centralised way.

OML is used for data collection and storage, but does not allow a direct data access or visualization. SQL database can be deployed as a backend for OML and thus allow direct access to the data. For data access, visualisation and analysis, TopHat [5] was already identified as the most suitable candidate, where Manifold is the core framework used by TopHat. It runs queries over distributed systems and data stores, and pre-existing deployments.

Figure 3 illustrates all these tools selected for the second implementation cycle and their interactions.

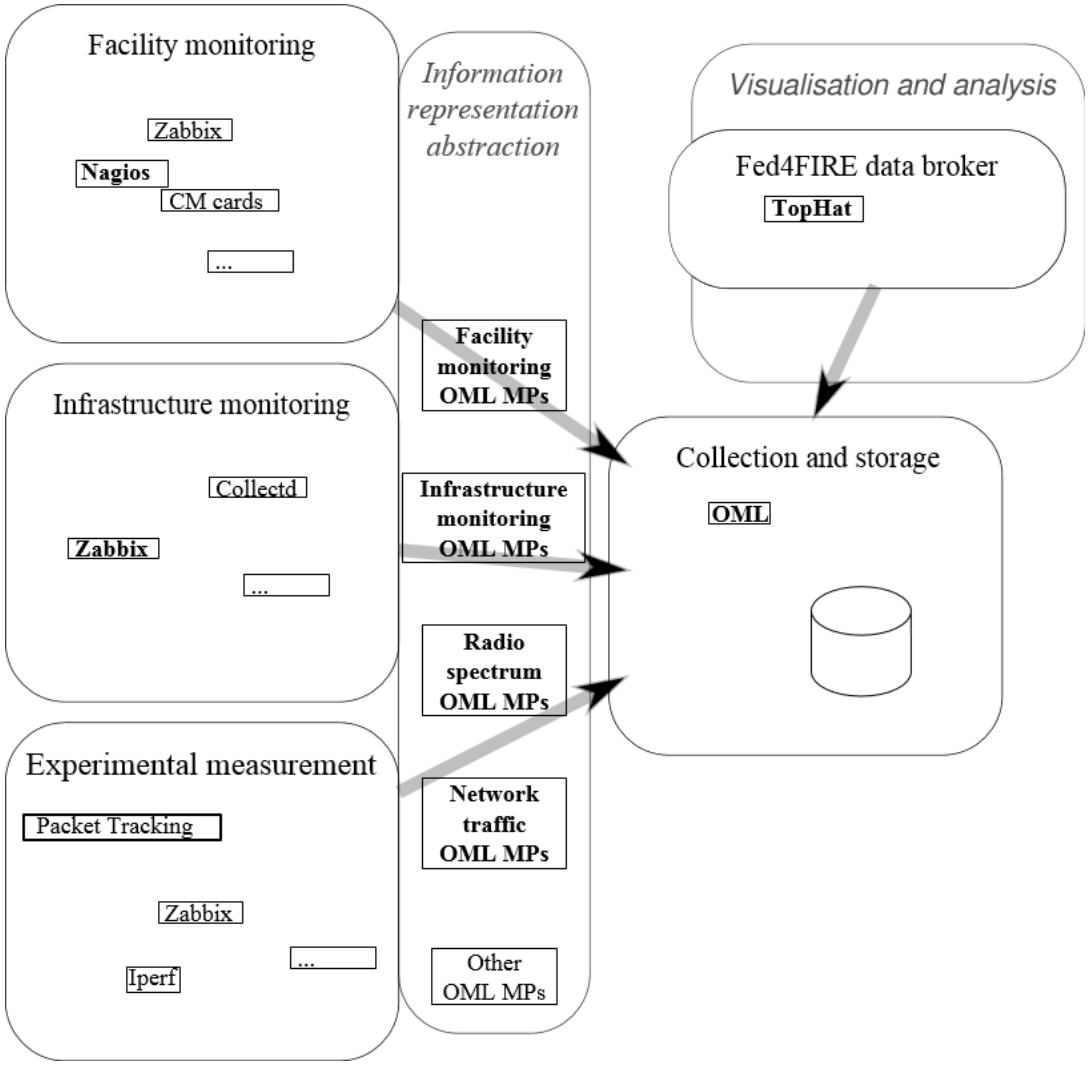

Figure 3: Proposed Fed4FIRE measurement and monitoring implementation for second cycle. Elements in bold are the default proposal for new deployments of on canonical testbed

3.3 Evaluation of possible mechanisms for the new implementations in the second cycle of Fed4FIRE

To fulfil the overall lifecycle of the infrastructure monitoring service starting from offering measurement and monitoring resources up to providing measurement and monitoring data and make them accessible to their stakeholders, new additional approaches next to those already supported in the first cycle are to be identified.

Infrastructure monitoring services are to be first announced for the stakeholders before being consumed. The most relevant stakeholders in Fed4FIRE are the experimenters, SLA service, reputation service, and the reservation service. In order for the SLA and reputation services to use infrastructure monitoring service capabilities, there is no need for announcing such. Predefined metrics are to be agreed on between the testbed providers and the SLA and reputation service supporters. In contrast, infrastructure monitoring capabilities per resource basis should be announced and made available for experimenters to understand which kind of infrastructure monitoring information about each resource can be provided, and of course mad ready to be requested.

There are various solutions for announcing and describing services and resources. Examples can be: the Resource Description Framework (RDF), ontologies, the web services discovery systems that are used for services registration and discovery. The Universal Description, Discovery and Integration (UDDI) [6] is an industry standard that used for describing, publishing and discovering information about web services. Furthermore, Teagle federation framework for Future Internet testbed experimentation [8] uses the information model in the Directory Enabled Networks - next generation (DEN-ng) [9] for describing resources.

However, the GENI Resource Specification (RSpec) [10] is widely used as a common language for describing, requesting and reservation of resources in Future Internet experimentation testbeds.

Fed4FIRE adopted GENI RSpec v3 to be used for describing, requesting and reservation of resources. Monitoring resources and services can be announced and described as normal resources. In order for aligning WP6 to the Fed4FIRE architecture RSpec is used as the most suitable candidate for describing monitoring resources and as well infrastructure monitoring capabilities per resource type.

RSpec are already used for describing experimental resources (e.g. VM, sensor, etc.) from the first cycle of Fed4FIRE, these will be extended in order for providing infrastructure monitoring capabilities of each resource. That means that each RSpec will be extended for monitoring demand.

Additionally, arbitrary monitoring and measurement resources, such as monitoring probes, collectors, converters, data viewers, etc., can be offered to the experimenters as normal resource described and made requestable through RSpec. This can be used for the experiment measuring service.

3.4 Details of the new selected mechanisms in the second cycle of Fed4FIRE

This section presents the new cycle 2 mechanisms that have been selected from the different implementation options described in the previous section. These mechanisms will complement those tools that have been already selected in the first cycle and were briefly discussed in Section 3.2.

3.4.1 Resource monitoring capabilities description and discovery

This section reflects only an implementation effort required to support the fine-grained infrastructure monitoring information for experimenters.

Fed4FIRE adopted SFA for resource management processes: description, allocation, provisioning, and release. Moreover, an RSpec is used in Fed4FIRE for describing, requesting and reservation of resources. For infrastructure monitoring demand, instead of adopting or implementing a new mechanism, the RSpec of a resource should be extended to include information exposing testbed capability of providing infrastructure monitoring information related to that resource.
The experimenters can then see, and on-demand request, infrastructure monitoring capabilities. In addition to this, the experimenter should identify an URI of an endpoint to which infrastructure monitoring data related to the requested resource will be pushed to.
The experimenter request is passed in the SFA technology to the Aggregate Manager (AM) at the testbed level that will then configure and initiate a specific measurement and monitoring resource called Wrapper that is in charge of collecting the respective infrastructure monitoring data from the local monitoring tools and provide them as OML streams to the endpoint identified by the experimenter. Examples on the RSpecs with infrastructure monitoring extensions are to be seen in the Appendix A: SFA / RSpec.

3.4.2 Data broker implementation

3.4.2.1 Per platform

In this section, we detail the development effort required to expose the identified platforms to Manifold though gateways, namely OML and SFA. For each gateway, we indicate the set of parameters that can be configured by the portal administrators (shared among all Fed4FIRE users) and those that are specified manually by the experimenter (since they might depend on the connected user).

Section 3.4.2.1.1 will briefly provide some pointers for people willing to develop additional Manifold gateways implemented as extensions. Manifold already supports a range of platforms, which can serve as a reference for building gateways toward new platforms. In our context, the PostgreSQL gateway will be of specific interest since it is the underlying database technology of OML and TDMI.

Sections 3.4.2.1.2 and 3.4.2.1.2 will thus detail these two gateways which both inherit from the PostgreSQL gateway.

3.4.2.1.1 Developing new Manifold gateways

It is not the purpose of this document to detail implementation and configuration steps since they might be subject to evolution with time. Instead, we here refer the reader to the dedicated documentation website:
http://trac.myslice.info/

It is possible to access source code and installation packages at this address:
http://trac.myslice.info/wiki/Manifold/Install
A gateway template file is provided in manifold/gateways/template.py in the source tree.

UPMC will help Fed4FIRE partners to develop additional gateways they might desire.

Once the gateway is developed and set up, one may configure the kind of data that can be retrieved through this gateway. This is achieved thanks to a metadata file having a format specified in the documentation (see Section 3.4.2.2.1).

3.4.2.1.2 The OML gateway

The development of the OML gateway will not be done from scratch since a proof-of-concept has already been done in the context of the French ANR project F-Lab. This gateway is based on a PostgreSQL gateway provided by Manifold, extending it with OML specificities regarding organization of the database schema, and the metadata stored within proprietary tables.

OML can be configured to export measurements to any database, we here consider the more specific case where it uses PostgreSQL, but further adaptations can be made to accommodate other types of databases.

In the generic case, the user would have to specify the IP configuration of its OML repository(ies), but since Fed4FIRE plans to provide and use a single central OML repository, it can be directly set by the portal administrator. Experiment monitoring data stored in the central repository will be available to Manifold; although measurements directed to a local OML instance are not supported at this stage. OML storage depends on an underlying database, e.g. sqlite or PostgreSQL. Manifold only supports PostgreSQL backend at the moment, which is the reasonable choice to provide a large measurement repository. Depending on the OML server configuration, either the Fed4FIRErepository is queried through a single common account used by Manifold, either each experimenter accesses its data through a dedicated PostgreSQL account (access control to the central OML DB has to be defined by its administrator). Authentication and authorization policies for OML data have not yet been decided by the project, they will be taken into account during the development of the gateway.

Manifold thus requires the following parameters:

- the IP address of the repository and its listening port,
- the PostgreSQL account (user and password) used to fetch the data,

In OML, each experiment is stored into a dedicated database, named after a specific OML identifier. It is not straightforward to map a slice name to such an identifier, and the testbed somehow needs to expose this information to Manifold so that it can do the mapping. One strength of the Manifold framework is that there is no required format to specify this mapping. For example, in the F-Lab case, OML measurements are performed by the IotLAB (formerly SensLAB) testbed, which use a job identifier originating from their scheduler (OAR) to tag measurements. Querying the SFA registry of the testbed (which presents an additional field named oar_job_id) can do the mapping between the slice and the job identifier.

Each database has two special tables, these are:

- ***_senders table***: this table maps a sender identifier to the hostname of the node that performed the measurement;
- ***_experiment_metadata table***: this table stores various information about the experiment: its start time, associated applications and measurements point (in OML terminology), as well as the format of a measurement tuple.

For example, we are considering a run of the sample OML application. Direct access to the PostgreSQL database gives the following:

```
$  psql -h HOSTNAME -U LOGIN -l
                                  Liste des bases de données
    Name    |    Owner     | Encoding |   Collate   |    CType     |   Access privileges
------------+--------------+----------+-------------+--------------+-----------------------
 100        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 101        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 405        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 408        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 411        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 412        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 420        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 451        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 502        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 504        | oml          | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 postgres   | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  |
 template0  | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/postgres          +
            |              |          |             |              | postgres=CTc/postgres
 template1  | postgres     | UTF8     | fr_FR.UTF-8 | fr_FR.UTF-8  | =c/postgres          +
            |              |          |             |              | postgres=CTc/postgres
```

To be noticed that postgres, template0 and template1 are PostgreSQL internal databases.

Supposing the job identifier of our slice is 100, we can look at the content of both special tables as follow:

```
$ psql -h HOSTNAME -U LOGIN -d 100

100=> select * from _senders ;
            name             | id
-----------------------------+----
 node2.devlille.senslab.info |  1
(1 line)

100=> select * from _experiment_metadata ;
            key              |                     value
-----------------------------+-----------------------------------------------
 start_time                  | 1362580349
 table_Application1_counter | -1 Application1_counter count_uint32:uint32
(2 lines)
```

The second table informs us that we have a table for the measurement point counter of Application1, named Application1_counter, and whose structure consists in a single integer (uint32) value (named counter_uint32).

Looking at measurements, we have:

```
100=> select * from "Application1_counter" limit 10;
 oml_sender_id | oml_seq | oml_ts_client | oml_ts_server | count_uint32
---------------+---------+---------------+---------------+--------------
             1 |       1 |      0.792995 |      1.291428 |            0
             1 |       2 |      0.793077 |      1.291934 |            1
             1 |       3 |      0.793096 |      1.292255 |            2
             1 |       4 |      0.793121 |      1.292562 |            3
             1 |       5 |      0.793137 |      1.293089 |            4
             1 |       6 |      0.793152 |        1.2934 |            5
             1 |       7 |      0.793168 |      1.293702 |            6
             1 |       8 |      0.793183 |      1.294025 |            7
             1 |       9 |      0.793197 |      1.294328 |            8
             1 |      10 |      0.793213 |      1.294624 |            9
(10 lines)
```

We see that OML adds four special fields, respectively the identifier of the sender, a sequence number as well as timestamps of the measurement on both the client and the server.

Direct access to the database can be used, but for some applications it might be more appropriate to directly access the different measurements related to a slice, possibly aggregated from different tools, into MySlice. We might also propose a set of default visualization that will help a user make sense of the advancement of his experiment.

The OML gateway will thus extend the PostgreSQL gateway by taking into account the specific structure of the OML databases, and additional metadata information available in the special tables.

3.4.2.2 Additional requirements for queries involving several platforms

3.4.2.2.1 Background: metadata

Manifold deduces query plan according to the metadata it learns, which depend on each platforms managed by Manifold.

Example: SFA

```
class slice {
    const text slice_urn;        /**< Slice Unique name */
    const text slice_hrn;        /**< Slice Human Readable name */
    const text slice_type;
    user        users[];        /**< List of users associated to the slice */
    user        pi_users[];     /**< List of PI users associated to the slice */
    const text slice_description;
    const text created;
    const text slice_expires;
    const text slice_last_updated;
    const text nodes;
    const text slice_url;
    const authority parent_authority;

    KEY(slice_hrn);
    CAPABILITY(retrieve, join, fullquery);
};

class user {
     const string user_hrn;
     const string user_urn;
     const string user_type;
     const string user_email;
     const string user_gid;
     const authority parent_authority;
     const string keys;
     slice slices[];
     authority pi_authorities[];
     const string user_first_name;
     const string user_last_name;
     const string user_phone;
     const string user_enabled;
     KEY(user_hrn);
     CAPABILITY(retrieve, join, fullquery);
};
```

3.4.2.2.2 Requirement: adopting a common ontology

Metadata must have a consistent naming to permit Manifold to perform consistent queries involving several data sources. Different ontologies are under definition and adoption in Fed4FIRE. We will integrate them into Manifold as soon as they are available. In the meantime, we have made a simple choice for naming fields.

3.4.2.3 Querying the data broker API

We illustrate the use of MySlice to access measurements related to a slice through the API. For this, we write a simple python script using the xmlrpc interface of MySlice. This script is based on the developments made in F-Lab.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import xmlrpclib, pprint
from config import auth

# Define the query
q = {
    'fact_table':   'slice',
    'filters':      [['slice_hrn', '=', 'ple.upmc.myslicedemo']],
    'fields':       ['slice_hrn', 'application.application',
'application.measurement_point']
}

# Connect to the XMLRPC API and send the query
srv = xmlrpclib.Server("http://localhost:7080/", allow_none = True)
ret = srv.forward(auth, q)

# Display results (ignoring error handling here)
assert 'result' in ret, "Exception occured!"
for r in ret['result']:
    print r
```

Lines 1-2 correspond to standard shebang headers allowing the script to be executable using the default python interpreter.

Lines 4-5 respectively import the python XMLRPC module, a pretty printer module, and an authentication token defined in an external file.

Lines 7-12 define a query using a python dictionary. This query indicates we are interested in ***slice*** information restricted to the ***ple.upmc.myslicedemo*** slice. Displayed fields will be the HRN of the slice, and the associated applications and measurement points.

Lines 14-16 make a connection to a local XMLRPC server hosting the MySlice API, and forward the query.

Lines 18-21 simply display the resulting records, ignoring potential exceptions for the purpose of this example.

Running this script gives us the following results (indented for clarity)

```
'application': [{'application': 'Application1',
                 'measurement_point': [{'measurement_point': 'counter'}]}],
 'slice_hrn': 'ple.upmc.myslicedemo'}
```

We see that the tool has transparently resolved the ***job_id*** associated to the slice, to use and to request related measurements. We can find the generated query plan in the server logs. It illustrates the different operations performed by MANIFOLD using the SQL terminology.

```
QUERY PLAN:
-----------
<main>
    SELECT slice_hrn, application.application, application.measurement_point
        WHERE slice_hrn == ple.upmc.myslicedemo
            JOIN slice_hrn == slice_hrn
                UNION
                    SELECT slice_hrn FROM oml::slice WHERE <Filter: >
                    SELECT slice_hrn FROM ple::slice WHERE <Filter: >
                    SELECT slice_hrn FROM omf::slice WHERE <Filter: >
                SELECT lease_id, slice_hrn FROM oml::slice WHERE <Filter: >
<subqueries>
    <main>
        SELECT application, lease_id FROM oml::application WHERE <Filter: >
    <subqueries>
        SELECT application, lease_id, measurement_point FROM oml::measurement_point
WHERE <Filter: >
```

More examples can be found in the Manifold sources.

3.5 Implementation Steps

As mentioned previously, this deliverable focuses on the infrastructure monitoring service. This section represents the implementation steps required for providing this service to the Fed4FIRE stakeholders, i.e. the experimenters, the SLA service, and the reputation service.

3.5.1 Infrastructure monitoring for experimenters

Figure 4 shows the workflow for providing infrastructure monitoring information to experimenters. The following, individual implementation steps are accordingly required. While in cycle 2 only the OML endpoint and the Boolean monitoring attribute should be specified, in cycle 3 more detailed monitoring metrics should be defined:

1. A testbed provider describes its resources through RSpec with monitoring support, such as specific metrics that will be measured (e.g. location of sensor, capacity of sensor) or maybe high-level services (spectrum analysis).
2. An experimenter while requesting resources through any experimentation tool, he/she can identify his interest in infrastructure monitoring if required. The experimenter should then identify an URI of an endpoint to which the infrastructure monitoring data is pushed.
3. Monitoring data will be exported to the experimenter destination endpoint as OML streams. This destination endpoint can be an OML server belonging to the experimenter or another resource that understands OML streams and can process them.
4. Upon receipt a request, testbed provider checks if infrastructure monitoring is requested. If yes, an OML Wrapper is then deployed and configured that is responsible for fetching measurement data from locally used monitoring tools and push it to the URI given by the experimenter. It is left to the testbed provider to use one wrapper resource per

resource, one per experiment or one per experimenters. A wrapper should be dynamically able to react to updates such as add more resources, remove some, etc.

5. The wrapper resource pushes measurement data as OML streams to the given URI.

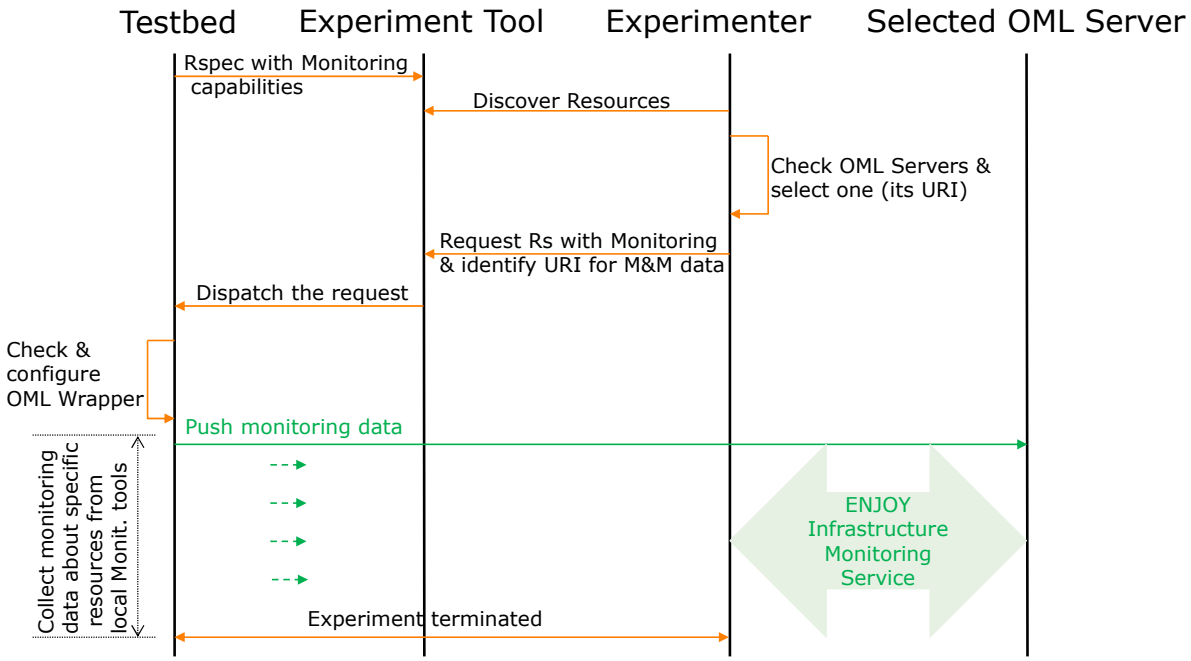

Figure 4: Infrastructure monitoring workflow for experimenters

3.5.1.1 Implementation at the Infrastructure Level

To support the infrastructure monitoring scheme proposed by WP6, each individual infrastructure (aka testbed) should implement or provide the following. In that regard, we assume that the infrastructure already has some means to actually measure infrastructure-related metrics that may be of interest to experimenters.

First the infrastructure should have some means to forward the measured data as OML streams. This could be achieved in two ways. The first option is for the testbed operator/manager to instrument their existing measurement application directly with OML. Indeed, OML provides a client library with binding in different languages (e.g. C, C#, Python, Java, Ruby), which allows one to add to the source code of any application some additional instructions to forward any measurements to the OML library itself, which will in turn encapsulates them in an OML stream towards one or more collection points. The second option is for the testbed operator to develop a separate software, which will pick up the data from any native format that the original measurement application is producing, and pass it on to the OML library, which will then behave as in the previous option. This solution effectively allows one to instrument existing applications which produce measurements, but for which no source code is available (thus preventing the first option).

Many examples of OML-instrumented resources using either this first or second option are available at: https://mytestbed.net/projects/omlapp/wiki/OML-instrumented_Applications.

Second, the infrastructure should provide a solution for various experimenters to share these instrumented resources. This could be realised through their "virtualisation". Indeed, the experiment will not have access to the instrumented resource itself. Rather when user A would like to collect infrastructure data from resource R, the testbed should create a "proxy" resource R' which would provide limited access to the monitored R itself. As far as the user is concerned, the proxy resource (we refer to this with OML Wrapper Resource as well in this deliverable) is indistinguishable from the source of the collected infrastructure data. However, multiple proxies may refer to the same underlying resource.
Finally, the infrastructure should provide standardised description of its monitoring capabilities. We decided to adopt the RSpec format as used by various other federation tools (e.g. SFA) to describe the monitoring capabilities of any infrastructure provided resource. Indeed, currently RSpec of a given resource already provides information about its capabilities such as available memory, storage or computing limits, thus it would be easy to extend such a description to provide monitoring capabilities. We note that although the resource we discussed here are the ones from the previous paragraph, i.e. the "wrapper resource", the practices of including in the RSPec the instrumented features of a resource could be generalised to any resources.

Our proposed implementation requirements for an individual testbed do not mandate which infrastructure measurement they should provide, nor which monitoring application to deploy. Furthermore, we also leave to the testbed the decision on the granularity of the monitoring resource to be made available to the experimenters. For example, one testbed may decide that a single monolithic monitoring resource (i.e. a "proxy resource" as described above) will be made available to access a small set of metrics, while another testbed may testbed to have multiple different monitoring resources for different types of metrics.

As an optional implementation step, a given infrastructure may decide to provide OML servers together with database backends (i.e. PostgreSQL servers) to the experimenters, in the same way as it provides any other resources. The experimenter may use them as collection end-points for his experiment-generated data, and also for his requested infrastructure-specific measurements. This is an optional convenience for the experiment, who alternatively would have to arrange for collection end-points for the data through some other means (e.g. run it himself, or book it as resources from another testbed or third party).

3.5.1.2 Implementation at the Experimenter Level

As detailed in the previous scenario (Figure 3), the experimenter is responsible for making sure that one or many OML own collection points are available to receive the infrastructure-generated measurements that he requested. These collection points are no different from any other resources that the experimenter would like to use in his experiment, and which he would have to first discover, reserve, and provision. As such, the OML collection point could be either hosted/provided by the experimenter himself or by any third party providing resources, such as a testbed provider or some Fed4FIRE entity.

Thus this proposed infrastructure monitoring scheme does not require the experimenter to implement any specific software, but rather allows him to treat infrastructure monitoring in the same manner as collecting measurement from any other instrumented resources.

3.5.1.3 Implementation of Supporting Tools

The tools required in the above implementation steps include the OML server and OML library with its bindings to different languages.

One major version of OML server and library has been released, i.e. OML 2.10. This version includes multiple features such as the support for metadata, dynamic addition of measurement points while an experiment is running, and the support for Boolean and globally unique ID data types. A complete list of features and improvements implemented by OML 2.10 can be found at the following link: http://mytestbed.net/news/46.

In addition a pre-release candidate of the next OML 2.11 version was pushed out recently, which provides features such as the support for IPv6, for vectors of values, and self-instrumentation of the OML entities (allowing reporting of dropped measurements and system resource usage). More information about this pre-release can be found at the following link: http://mytestbed.net/news/50 Finally, three new language bindings where released for the OML library. The original supported languages were C, Python and Ruby, and the new supported ones are C#, Java, and NodeJS. Detailed information on these new bindings can be found at https://github.com/mytestbed/oml4j; https://github.com/mytestbed/oml4node.

3.5.2 Infrastructure monitoring for the SLA service

The SLA management service is interested in both the infrastructure monitoring and facility monitoring data.

For infrastructure monitoring data, each testbed provides monitoring data of a predefined metrics to SLA mgmt. The AM at the testbed level could be in charge of triggering this service, once the experiment resources are started. Thus, monitoring data is provided and only during the experiment lifetime. This is achieved in two steps. First, each testbed could provide an OML server + PostgreSQL database backend to store this data. Possibly the federator could provide an additional central OML server + PostgresSQL for convenience, meaning that testbeds can start with using the central one first to reduce the time and efforts needed to adopt the SLA service. However, similar to how we approach member- and slice authorities in our WP2 architecture described in D2.4, the concept clearly remains that the SLA solution is a distributed architecture. If the central component would be discontinued or disrupted, testbeds can always easily deploy this component themselves. In cycle 2, iMinds plans to operate such an optional central OML server, in order to ease the work of testbeds exploring the usefulness of the SLA service in their specific context. Second, the SLA management module retrieves data per experiment basis from the PostgreSQL database.

Regarding the facility monitoring, the SLA service is not interested in facility monitoring data in the second cycle. For further implementations, SLA management will require facility monitoring, if any provider would be interested in an SLA type which needs it. Similarly to what each testbed currently does for the FLS, facility monitoring data should be as well pushed to the collection OML server that is either deployed locally or at the federation level. The data is then being accessed by the SLA management module.

3.5.2.1 Installation of New Tools

Some requirements tools should be needed from the testbed provider and they are:

- Local monitoring tools for monitoring the testbed infrastructure.
- Each testbed should provide infrastructure monitoring information as OML streams exported into an OML server together with a PostgreSQL database backend at a testbed level or into the central OML server with the PostgreSQL database backend at the federation level
- OML Wrapper: it is in charge of collecting measurement data from the local monitoring tools and pushing the data as OML streams to the OML server.

3.5.3 Infrastructure monitoring for the reputation service

The individual implementation steps for the reputation service are as follows:

1. A testbed provider should define what kind of services (uptime, network performance, etc.) wants to include in the reputation service and provide the corresponding monitoring data as OML streams into an central OML server at the broker level or provide locally (at the testbed level) an OML server for reputation service.
2. Infrastructure monitoring data for the reputation service need to be exported in a per experiment/slice basis. For this reason, an OML Wrapper needs to be deployed and configured for fetching the corresponding measurement data from local monitoring tools and push it to the OML server. To reduce the consumed storage, the data may be stored only for a small time period (a few days or a week).
3. The reputation service will use the Manifold framework to retrieve monitoring data for a particular experiment from the central OML Server that will have a PostgreSQL database backend (or from the testbed-level OML Servers which reside at testbeds used by the target experiment). In the case of data retrieval from testbed-level OML Servers, Manifold will use the slice/experiment ID in order to identify the target OML servers that have the infrastructure monitoring data used in the particular experiment.
4. The reputation service will calculate reputation scores for the various services with the use of the corresponding infrastructure monitoring data and will store the results in its own database

3.5.3.1 Installation of New Tools / Adaptation of Existing Tools

- Requirements from testbed provider
  - Mandatory:
    - Local monitoring tools for monitoring the testbed infrastructure
    - Each testbed should provide infrastructure monitoring information as OML streams exported to either the central OML Server or to a local (testbed-level) OML server
    - OML Wrapper: it is in charge of collecting measurement data from the local monitoring tools and push it as OML streams to the OML server
  - Optional:
    - OML server + PostgreSQL DB backend for storing the infrastructure monitoring data needed for the reputation service
- Requirements from Federation level
  - At the Portal level / Fed4FIRE data broker
    - Manifold should be able to retrieve the infrastructure monitoring data needed by the reputation service in a per experiment (slice) basis, based on experiment (slice) ID, from the central OML server or the testbed-level OML servers.

3.5.4 Infrastructure monitoring for the reservation broker

The individual implementation steps for the reservation broker are as follows:

1. For the reservation broker to make provisioning and reservation decisions, it will need specific monitoring information about the offered resources. This kind of information varies from testbed to another depending on the types and the number of resources offered by a testbed. The metrics that should be measured are predefined between the testbed provider and the reservation broker service. It should be clear here to distinguish between the reservation broker need in terms of infrastructure monitoring information and the real reservations already took place at the testbed level. In this deliverable we describe broker's requirements with respect to infrastructure monitoring information, e.g. CPU and memory related metrics for physical machines. In contrast, the information regarding reservations in a testbed is out of the scope of this deliverable.
2. A testbed provider will provide the monitoring data as OML streams into a central OML server at the broker level.
3. This infrastructure monitoring data needs to be exported periodically. For this reason, an OML Wrapper needs to be deployed and configured for fetching the corresponding measurement data from local monitoring tools and push it to the OML server. To reduce the consumed storage, the data may be stored only for a small time period (a few days or a week).
4. The reservation broker will use the Manifold framework to retrieve monitoring data from the central OML Server that will have a PostgreSQL database backend.

3.5.4.1 Installation of New Tools / Adaptation of Existing Tools

- Requirements from testbed provider
  - Mandatory:
    - Local monitoring tools for monitoring the testbed infrastructure
    - Each testbed should provide infrastructure monitoring information as OML streams exported to the central OML server at the federation level
    - OML Wrapper: it is in charge of collecting measurement data from the local monitoring tools and push it as OML streams to the OML server

- Requirements from Federation level
  - Manifold should be able to retrieve the infrastructure monitoring data needed by the reservation broker from the central OML server

3.6 Coordination

For the federation to be successful, it is important that monitoring and measurement tools provide the data following the same structure. It is therefore needed to provide a unified abstraction for the way data from monitoring and measurement tools are grouped together in meaningful sets, and make these sets standard across tools measuring the similar aspects. OML framework is adopted by Fed4FIRE for this purpose. OML does not enforce any semantics on the schema of its measurement points. NICTA and UTH will curate a list of measurement point schemas to use for specific types of metrics for that purpose, and provide technical support on their implementation into the tools in use by each testbed. This will ensure homogeneity across testbeds using different sets of tools to measure the same characteristics. Note that these measurement point schemas will not only ease the life of the experimenters, but it is also a vital instrument in the implementation of the SLA management, reputation engine and reservation broker that are key services of the federation framework. The exact list of compulsory metrics will be defined based on further discussions and experiences. These will be driven by the coordination task presented in this section.

As aforementioned in this document that testbeds might use different tools for monitoring and measurements but they should provide the data as OML streams. It is therefore needed to develop OML wrappers to convert the data from the local tools formats into OML streams. TUB, NICTA, UTH will provide some implementation examples in the OML-supported languages (Python, Ruby, etc.).

Some testbeds could provide a collection resource (OML server together with database server) as a resource for experimenters. TUB, NICTA, UTH will provide technical support on this matter.

The federation services, portal, reputation engine and reservation broker, each require specific monitoring information to be fetched from a central collection resource (OML server together with database server) at the federation level that will be provided by iMinds. The Manifold as a reference implementation for the data broker will be used for this purpose. It is used by these federation services to fetch their data. In addition to providing measurable characteristics of the offered

resource, the portal could allow the experimenters to access to their data (stored in their collection resources, wherever they are) in a user-friendly manner. This will not be ready in the second cycle but maybe in further cycles. However, the experimenters will use the Manifold to retrieve their data from their OML servers in a user-friendly way since it allows them to just send queries to get their data. The integration of the portal and the Manifold will be implemented by UPMC. On the other hand, the integration between the Manifold and the OML will be done by UPMC, NICTA and UTH as a shared effort. UPMC will help Fed4FIRE partners to develop additional Manifold gateways they might desire.

The types of metrics to be provided to the experimenters as well as their frequencies (update rate) are left to be decided by the testbeds. Metrics required by the SLA management, reputation and reservation services are to be defined between the testbed provider and the developers of these services depending on their functionalities as well as the nature and types of resources/services being offered. This is to be covered by WP7. However, one thing to bear in mind is that the history of the data will not be stored in the central collection point for a long time to avoid having large amount of collected data. Furthermore, the frequencies will be set to significant values for the same purpose.

RSpecs is used in Fed4FIRE for resource description and provisioning. Since it is extendable, it will be used to advertise offering infrastructure monitoring service to the experimenters and to pass their interest on this service on-request basis. TUB will provide examples for this purpose and help on “how to” once needed.

The Aggregate Manager at the testbed should be extended to process the requests. This is out of the scope of WP6, it’s supported by WP5.

4 Summary

4.1 Mapping of Architecture to Implementation Plan

This deliverable presents the specification as well as the design of the second cycle implementation of the Fed4FIRE measurement and monitoring architecture. This architecture is in line with the one defined in D2.4. In order to implement this design, Table 5 presents the implementation steps that have been identified. Most steps in terms of software deployment and instrumentation should be undertaken independently by all participants. Where commonalities exist (e.g. the use of any of the recommended tools "Zabbix, Nagios or collectd" as well as the use of OML) instrumentation should be a common effort. Some implementation efforts are done at the federation level such as the data broker (Manifold) and the central data collection resource.

Functional element	Implementation strategy
Facility Monitoring	• Deploy Nagios and/or Zabbix and/or collectd or any equivalent tool for the same purpose if not yet available (all participants) • Deploy OML if not yet available (all participants with support from NICTA/UTH) • Instrument these relevant measurement systems (all participants, with support from WP6)
Fine-grained infrastructure monitoring for experimenters	• Deploy Nagios and/or Zabbix and/or collectd or any equivalent tool for the same purpose if not yet available (all participants) • Deploy OML if not yet available (all participants with support from NICTA/UTH) • Instrument these relevant measurement systems (all participants, with support from WP6) • Extend the RSpecs for advertising fine-grained infrastructure monitoring capabilities of the offered resources (all participants, with support from TUB) • Extend the Aggregate Manager for fine-grained infrastructure monitoring need (all participants)
Coarse-grained infrastructure monitoring for federation services	• Deploy Nagios and/or Zabbix and/or collectd or any equivalent tool for the same purpose if not yet available (all participants) • Deploy OML if not yet available (all participants with support from NICTA/UTH) • Instrument these relevant measurement systems (all participants, with support from WP6) • Extend the Aggregate Manager for coarse-grained infrastructure monitoring need (all participants)
Experiment measurement	• Deploy OML if not yet available (all participants with support from NICTA/UTH)

	• Instrument relevant measurement systems (all participants, with support from WP6) • Maintain clearinghouse of measurement points (NICTA)
Measurement service	• Provide a measurement service that is able to provide measurements data exported as OML streams without a need for the experimenter to set up the measurement framework (provided by participants as optional service)
Data collection	• Deploy a collection resource (OML server together with database) for infrastructure monitoring and measurements (optional for all participants, iMinds will provide a central one at the federation)
Data access for multiple stakeholders (experimenters and the reputation and reservation)	• Deploy Manifold at the federation level that acts as a data broker between users and their collection resources (supported by UPMC) • Make OML measurement databases accessible to Manifold (UPMC, NICTA, UTH)

Table 5: Implementation strategy of functional elements

4.2 Future Plans

Improvements and extensions for the monitoring and measurement services are planned for future work. In particular, we will address the limitation of the security support in the OML framework.

Furthermore, the new expected requirements and feedback from the experimenters who joined Fed4FIRE through the Open-calls will be addressed for the third development cycle of the project.

As ontologies is being studied as a possible candidate for description of resource instead of RSpecs within the context of WP5. As a result of the ontology efforts, we might adopt their approach to cover measurement and monitoring metrics as well. However we plan to first develop an information model to cover the diversity of metrics across the federation.

References

[1] Zabbix – open source monitoring system, available online at www.zabbix.com, last accessed on December 09, 2013.

[2] Nagios monitoring tool, available at www.nagios.org, visited on December 09, 2013.

[3] Collectd, available online at http://collectd.org/, last accessed on December 09, 2013.

[4] O. Mehani, G. Jourjon, T. Rakotoarivelo, and M. Ott, "An instrumentation framework for the critical task of measurement collection in the future Internet," Under review, 2012.

[5] TopHat. Available online at http://www.top-hat.info/, last accessed on December 09, 2013.

[6] A. E. Walsh, "UDDI, SOAP and WSDL: The Web Services Specification Reference Book", Volume 53, Issue 4, July 2002.

[7] Manifold Framework, Website, http://trac.myslice.info/wiki/Manifold, visited on December 09, 2013.

[8] Teagle, available online at www.fire-teagle.org, last accessed on December 12, 2013.

[9] J. Strassner, "Information model - DEN-ng", 2009. http://www.autonomicmanagement.org/denng/index.php

[10] GENI RSpec v3. Available online at: http://groups.geni.net/geni/wiki/GENIExperimenter/RSpecs, last accessed on December 09, 2013.

Appendix A: SFA / RSpecs

It is not desirable to list all infrastructure monitoring metrics being measured related to a specific resource through the RSpec to avoid having a huge one. Example: given that the resource is a virtual machine (VM), the infrastructure monitoring information will be to measure multiple metrics (CPU, memory, incoming and outgoing traffic on available interfaces, etc.) of the physical machine hosting that VM. The RSpec of the VM resource is to be extended to include the following:

- Mandatory: information about the possibility of providing the infrastructure monitoring service related to the resource.

  For this purpose, a Boolean attribute for any kind of resource whether monitoring services are available in the advertisement or not can be used. The RSpec could look:

```
<rspec type="advertisement">
        <node component_id="xxx" component_manager_id="xxx" exclusive="true"
        monitored="true">
                <available now="true"/>
                <location country="Germany" latitude="52.525961"
                longitude="13.314297"/>
                <hardware_type name="openEPC-UE"/>
        </node>
</rspec>
```

- Optional: a link to a list of metrics that are going to be measured and related to that resource.

  Since we only have the GENI RSpec v3 right now, it might be that it is needed to hard code the information that is interesting in this context within an XML Schema. One example could be:

```
<rspec type="advertisement">
        <node component_id="xxx" component_manager_id="xxx" exclusive="true">
                <monitoring>
                  <metrics>
                    <free_ram supported="true" >
                  </metrics>
                </monitoring>
        <available now="true"/>
        <location country="Germany" latitude="52.525961" longitude="13.314297"/>
        <hardware_type name="openEPC-UE"/>
        </node>
</rspec>
```

The experimenter should then be able to see this kind of information, and identify the following in the resource creation request:

1) whether or not to get benefit of the infrastructure monitoring service
2) a URI of an endpoint where infrastructure monitoring data (OML streams) related to this resource should be sent to

This can be achieved as follows:
For 1), if a <monitoring/> tag does exist.
For 2), the URI of the experimenter OML server + SQL database is identified as
<OML server>URI</OML server>.

The RSpec would then look as follow:

- Mandatory:

```
<rspec type="request">
    <node client_id="VM" component_manager_id="xxx" >
      <sliver_type name="xo.small" />
      <monitoring>
        <oml_server url="http://example.org" />
      </monitoring>
    </node>
</rspec>
```

- Optional:

```
<rspec type="request">
    <node client_id="VM" component_manager_id="xxx" monitor="true">
      <sliver_type name="xo.small" />
      <monitoring>
        <metrics>
          <free_ram requested="true" >
        </metrics>
          <oml_server url="http://example.org" />
      </monitoring>
    </node>
</rspec>
```

Appendix B: Example scripts for instrumenting Zabbix and Nagios monitoring frameworks with OML wrappers.

Instrumenting Zabbix

This example script was provided by Christoph Dwertmann from NICTA. It is kept up to date on github: https://github.com/mytestbed/oml4r/blob/master/examples/oml4r-zabbix.rb. Bellow you can find the code as it was on January 23rd 2014:

```
#!/usr/bin/env ruby

# example script that reads CPU load measurements from a Zabbix server
# and pushes them into an OML database

# make sure you install these two gems
require "zabbixapi"
require "oml4r"

# # Zabbix node names
# nodes = ["10.129.16.11", "10.129.16.12", "10.129.16.13"]

# Define your own Measurement Point
class CPU_MP < OML4R::MPBase
  name :CPU
  param :ts, :type => :string
  param :node, :type => :string
  param :load1, :type => :double
  param :load5, :type => :double
  param :load15, :type => :double
end


# Initialise the OML4R module for your application
oml_opts = {
  :appName => 'zabbix',
  :domain => 'zabbix-cpu-measurement',
  :nodeID => 'cloud',
  :collect => 'file:-'
}
zabbix_opts = {
  :url => 'http://cloud.npc.nicta.com.au/zabbix/api_jsonrpc.php',
  :user => 'Admin',
  :password => 'zabbix'
}

interval = 1

nodes = OML4R::init(ARGV, oml_opts) do |op|
  op.banner = "Usage: #{$0} [options] host1 host2 ...\n"

  op.on( '-i', '--interval SEC', "Query interval in seconds [#{interval}]"
) do |i|
    interval = i.to_i
```

```
  end
  op.on( '-s', '--service-url URL', "Zabbix service url
[#{zabbix_opts[:url]}]" ) do |u|
    zabbix_opts[:url] = p
  end
  op.on( '-p', '--password PW', "Zabbix password
[#{zabbix_opts[:password]}]" ) do |p|
    zabbix_opts[:password] = p
  end
  op.on( '-u', '--user USER', "Zabbix user name [#{zabbix_opts[:user]}]" )
do |u|
    zabbix_opts[:user] = u
  end
end
if nodes.empty?
  OML4R.logger.error "Missing host list"
  OML4R::close()
  exit(-1)
end

# connect to Zabbix JSON API
zbx = ZabbixApi.connect(zabbix_opts)

# catch CTRL-C
exit_requested = false
Kernel.trap( "INT" ) { exit_requested = true }

# poll Zabbix API
while !exit_requested
  nodes.each{|n|
    # https://www.zabbix.com/documentation/2.0/manual/appendix/api/item/get
    results = zbx.query(
      :method => "item.get",
      :params => {
        :output => "extend",
        :host => "#{n}",
        # only interested in CPU load
        :search => {
          :name => "Processor load"
        }
      }
    )
    unless results.empty?
      l15 = results[0]["lastvalue"]
      l1 = results[1]["lastvalue"]
      l5 = results[2]["lastvalue"]
      #puts "Injecting values #{l1}, #{l5}, #{l15} for node #{n}"
      # injecting measurements into OML
      CPU_MP.inject(Time.now.to_s, n, l1, l5, l15)
    else
      OML4R.logger.warn "Empty result usually means misspelled host
address"
    end
  }
  sleep interval
end

OML4R::close()
puts "Exiting"
```

Instrumenting Nagios

This example script was provided by Iakovos Panourgias from the University of Edinburgh EPCC (this organisation owns the copyright of the code, and licenses it under a Creative Commons Attribution 4.0 International License). The code consists of multiple files:

- ReadMe.txt
- _runme.sh
- facilityMonitoring
- facilityMonitoring.rb
- facility_monitoring_controller.rb

These files can be downloaded from the Fed4FIRE SVN, they are bundled in the following file on the SVN: / WP6 Measurements/Deliverables/D6.2/ facilityMonitoring_nagios.zip.

As an illustration of the concept, the most important snippet of code from the facilityMonitoring.rb file is copied bellow:

```
require 'rubygems'
require 'time'
require 'open-uri'
require 'oml4r'
require 'nokogiri'
require 'syslog'

# ICMP (PING) Measurement Point
class ICMP_MP < OML4R::MPBase
  name :icmp
  param :insert_time, :type => :string
  param :node,        :type => :string
  param :up,          :type => :double
  param :last_check,  :type => :string
end

# SSH Measurement Point
class SSH_MP < OML4R::MPBase
  name :ssh
  param :insert_time, :type => :string
  param :node,        :type => :string
  param :up,          :type => :double
  param :last_check,  :type => :string
end


# Initialise the OML4R module for your application
opts = {
```

```
  :appName => 'zabbix',
  :domain => 'BonFIRE',
  :nodeID => 'BonFIRETestbeds',
  :collect => 'file:-'} # Server could also be tcp:host:port

Hosts = []                              # Array that holds the Hosts, from
the NAGIOS parsing.
HostsXML = []                           # Array that holds the Hosts, from
the XML parsing.
ServiceStatuses = []                    # Array that holds the Service
Statuses, from the NAGIOS parsing.

Info = Struct.new(:created, :version, :last_update_check)
# Structure for the NAGIOS Information segment.
ProgramStatus = Struct.new(:nagios_pid, :program_start)
# Structure for the NAGIOS Program Status segment.
HostStatus = Struct.new(:host_name, :check_command, :plugin_output,
:long_plugin_output, :performance_data)             # Structure for the
NAGIOS Host Status segments.
ServiceStatus = Struct.new(:host_name, :service_description,
:plugin_output, :long_plugin_output, :performance_data)    # Structure for
the NAGIOS Service Status segmens.
HostStatusXML = Struct.new(:id, :name, :RVM, :TCPU, :FCPU, :TMEM, :FMEM,
:STATE)        # Structure for the XML Host Status.

info = Info.new()
programstatus = ProgramStatus.new()
host = HostStatus.new()
servicestatus = ServiceStatus.new()
hostXML = HostStatusXML.new()

XML_ARRAY = ["http://bonfire.epcc.ed.ac.uk/one-status.xml",
"http://frontend.bonfire.grid5000.fr/one-status.xml"]  # Init array with
location of BonFIRE XML files

Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning
"facility_monitoring STARTING" }

$ii = 1
while $ii > 0 do

  begin
    OML4R::init(ARGV, opts)
    rescue OML4R::MissingArgumentException => mex
      $stderr.puts mex
        Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning
"facility_monitoring ERROR exiting." }
      exit
  end

  begin
  open("http://nebulosus.rus.uni-stuttgart.de/status.dat") do |f|
```

```
    Hosts.clear                         # Clear the Hosts array
    ServiceStatuses.clear               # Clear the Service Status array

    inInfo = false                      # Initialise the status of the
BOOLeans to false.
    inProgramStatus = false
    inHostStatus = false
    inServiceStatus = false
    inContactStatus = false
    inServiceComment = false
    $lineNumber = 0                     # Initialise the number of lines to
0.

    f.each_line do |line|
      if $lineNumber > 90               # Defensive code to detect changes
in the format of the "status.dat" file or a parser error.
        timeNOW = Time.new.strftime("%Y-%m-%d_%H_%M_%S")        # If we
parse for more than 90 lines, then something is WRONG
        outputfile = File.open("coredump_#{timeNOW}.txt","w")   # So we
dump the input file to coredump_TIME.txt
        f.pos = 0                                               # and we
exit. There is nothing else that we can do.
        outputfile.write(f.readlines)                           # A Human
operator should have a look at the coredump file and
        outputfile.close                                        # fix the
parser.
        OML4R::close()                                          # Close the
OML connection
        abort("Session longer than 90 lines. Either modify the script or
check that the coredump file is correct.")
      elsif inInfo              # We are in the INFO segment.
        if line.strip == "}"    # If we find the close bracket, reset the
lineNumber and set inInfo to FALSE.
          inInfo = false
          $lineNumber = 0
        else                    # We are in the INFO segment. Increment the
lineNumber and store the variables.
          $lineNumber +=1
          if line.include? 'version'
            if !line.include? '_v'
              info[:version] = line.split("=", 2).last.strip
            end
          elsif line.include? 'created'
            info[:created] = line.split("=", 2).last.strip
          elsif line.include? 'last_update_check'
            info[:last_update_check] = line.split("=", 2).last.strip
          end
        end
      elsif inProgramStatus     # We are in the PROGRAM STATUS segment.
        if line.strip == "}"    # If we find the close bracket, reset the
lineNumber and set inProgramStatus to FALSE.
          inProgramStatus =false
```

```
          $lineNumber = 0
        else                    # We are in the PROGRAM STATUS segment.
Increment the lineNumber and store the variables.
          $lineNumber +=1
          if line.include? 'nagios_pid'
            programstatus[:nagios_pid] = line.split("=", 2).last.strip
          elsif line.include? 'program_start'
            programstatus[:program_start] = line.split("=", 2).last.strip
          end
        end
      elsif inHostStatus        # We are in one of the many HOST STATUS
segments.
        if line.strip == "}"    # If we find the close bracket, reset the
lineNumber, set inHostStatus to FALSE and push the object.
          inHostStatus = false
          $lineNumber = 0
          Hosts.push(host.dup)
        else                    # We are in one of the many HOST STATUS
segments. Increment the lineNumber and store the variables.
          $lineNumber +=1
          if line.include? 'host_name'
            if line.include? 'localhost'
              inHostStatus = false
            else
              host[:host_name] = line.split("=", 2).last.strip
            end
          elsif line.include? 'check_command'
            host[:check_command] = line.split("=", 2).last.strip
          elsif line.strip.start_with? 'performance_data'
            host[:performance_data] = line.split("=", 2).last.strip
          elsif line.include? 'plugin_output'
            if !line.include? 'long_plugin_output'
              host[:plugin_output] = line.split("=", 2).last.strip
            else
              host[:long_plugin_output] = line.split("=", 2).last.strip
            end
          end
        end
      elsif inServiceStatus     # We are in one of the many SERVICE STATUS
segments.
        if line.strip == "}"    # If we find the close bracket, reset the
lineNumber, set inServiceStatus to FALSE and push the object.
          inServiceStatus = false
          $lineNumber = 0
          ServiceStatuses.push(servicestatus.dup)
        else                    # We are in one of the many SERVICE STATUS
segments. Increment the lineNumber and store the variables.
          $lineNumber +=1
          if line.include? 'host_name'
            if line.include? 'localhost'
              inServiceStatus = false
            else
```

```
            servicestatus[:host_name] = line.split("=", 2).last.strip
          end
        elsif line.include? 'service_description'
          servicestatus[:service_description] = line.split("=",
2).last.strip
        elsif line.include? 'plugin_output'
          if !line.include? 'long_plugin_output'
            servicestatus[:plugin_output] = line.split("=", 2).last.strip
          else
            servicestatus[:long_plugin_output] = line.split("=",
2).last.strip
          end
        elsif line.strip.start_with? 'performance_data'
          servicestatus[:performance_data] = line.split("=",
2).last.strip
        end
      end
    elsif line.strip == "info {"              # Found the start of the
INFO segment.
      inInfo = true
    elsif line.strip == "hoststatus {"        # Found the start of a HOST
STATUS segment.
      inHostStatus = true
    elsif line.strip == "programstatus {"     # Found the start of the
PROGRAM STATUS segment.
      inProgramStatus = true
    elsif line.strip == "servicestatus {"     # Found the start of a
SERVICE STATUS segment.
      inServiceStatus = true
    end
  end
end
rescue OpenURI::HTTPError => e
  if e.message == '404 Not Found'
    Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning
"OpenURI::HTTPError:" + e.message + "." }
    puts "OpenURI::HTTPError:" + e.message + "."
  elsif e.message == '504 Gateway Time-out'
    Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning
"OpenURI::HTTPError:" + e.message + "." }
    puts "OpenURI::HTTPError:" + e.message + "."
  else
    puts "......" + e.message + "."
    raise e
  end
rescue Errno::ETIMEDOUT => e
    Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning
"Errno::ETIMEDOUT:" + e.message + "." }
    puts "Errno::ETIMEDOUT:" + e.message + "."
rescue Exception => e
    Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning
"UNHANDLED Exception:" + e.message + ",at host:" + xmlSITE + "." }
```

```
      puts "UNHANDLED Exception:" + e.message + ",at host:" + xmlSITE + "."
  end

  puts "Created: " + info[:created] + "[" +
Time.at(info[:created].to_i).to_s + "], Version: " + info[:version] + ",
Last Update Check: " + info[:last_update_check] + "," +
Time.at(info[:last_update_check].to_i).to_s + ",Nagios PID: " +
programstatus[:nagios_pid] + ", Program Start: " +
programstatus[:program_start]
  Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.info
"Created: " + info[:created] + "[" + Time.at(info[:created].to_i).to_s +
"], Version: " + info[:version] + ", Last Update Check: " +
info[:last_update_check] + "," +
Time.at(info[:last_update_check].to_i).to_s + ",Nagios PID: " +
programstatus[:nagios_pid] + ", Program Start: " +
programstatus[:program_start] }

# For each Hosts objects check the status and inject the data
# in the OML data stream
  Hosts.each do |i|
#  puts "Hosts Host name: " + i[0] + ", Command: " + i[1] + ", Output: " +
i[2] + ", Long: " + i[3] + ", Data: " + i[4]         # DEBUG
    if i[1] == "check-host-alive"
      if (i[0].start_with? 'wan') || (i[0].start_with? 'bf')
        if i[2].start_with? 'PING OK'
#          puts "Host :" + i[0] + ", is ALIVE"                  # DEBUG
#          ServerStatusNagiosMP.inject(i[0], info[:created], 0);
          ICMP_MP.inject(Time.now.to_s, i[0], 1,
Time.at(info[:created].to_i).to_s)
        else
#          puts "Host :" + i[0] + ", is DEAD!!!!!!!!!!!!!!!!"   # DEBUG
#          ServerStatusNagiosMP.inject(i[0], info[:created], 3);
          ICMP_MP.inject(Time.now.to_s, i[0], 0,
Time.at(info[:created].to_i).to_s)
        end
      end
    end
    if i[1] == "check_ssh"
      if (i[0].start_with? 'ssh-epc') || (i[0].start_with? 'ssh-inri') ||
(i[0].start_with? 'ssh-ibb')
        if i[2].start_with? 'SSH OK '
          SSH_MP.inject(Time.now.to_s, i[0], 1,
Time.at(info[:created].to_i).to_s)
           else
             SSH_MP.inject(Time.now.to_s, i[0], 0,
Time.at(info[:created].to_i).to_s)
        end
      end
    end
  end

  ServiceStatuses.each do |i|
```

```
#   puts "SVCs Host name: " + i[0] + ", Service: " + i[1] + ", Output: " +
i[2] + ", Long: " + i[3] + ", Data: " + i[4]         # DEBUG
  end

  # Don't forget to close when you are finished
  OML4R::close()

  sleep(60)
end

Syslog.open($0, Syslog::LOG_PID | Syslog::LOG_CONS) { |s| s.warning
"facility_monitoring EXITING" }
```