Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

Tool Suite for Deployment, Monitoring & Controlling of Virtual Service Platforms (Final)

Abstract

In order to bring Virtual Service Platforms to the operational state, besides its pre-configuration, additional deployment and runtime activities need to take place.

To ensure the proper execution of the VSP we advocate the use of several supporting tools for deployment, monitoring and controlling whose design and implementation is challenging for several reasons: (i) monitoring and adaptation tools have to take into account heterogeneousness of underlying service platforms; (ii) different service platforms use different mechanisms and communication schemes for monitoring (i.e. events, JMX, SMNP) from which all needs to be covered by appropriate tools; (iii) it is not trivial to define a standardized monitoring and adaptation interfaces for all existing and future service platforms.

This deliverable shows the final state of the tools proposed to address these challenges.

Document ID:	INDENICA – D4.2.2
Deliverable Number:	D4.2.2
Work Package:	WP4
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2013-09-30
Author(s):	NDL, TUV, SAP, PDM, SUH, SIE, UV

Project Start Date: October 1st2010, Duration: 36months

Version

0.1	20. September 2013	Initial version
0.2	27. September 2013	Updated and revised Section 2.5
0.3	17. October 2013	Updated Section 4 and Apendix A
1.0	21. October 2013	Final version

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

Table of Figures

1 Introduction

The main purpose of the tool suite proposed as a part of this Work Package is to bring the models, mechanisms and tools delivered by Work Packages 1-3 into their operational state producing Virtual Service Platforms that will be deployable and manageable at runtime.

Different tools make it possible to create scalable monitoring mechanisms that allow generating a global overview of the VSP.

Tool suite also allows actors to cope with non-functional metrics of underlying service platforms for different reasons, for example: (i) drilling down for root cause of performance bottlenecks; (ii) estimation and evaluation of current and planned adaptation policies; (iii) monitoring of a global view of the runtime environment.

In the second section of the document we describe how we leverage tools provided by different technical Work Packages to support the WP4 tool suite. In the third section we describe the deployment tool for instantiating different heterogeneous service platforms together with their added-value interfaces. Fourth section describes tools that provide monitoring capabilities. In the same section we also describe how the monitoring engines cooperate with different service platform and what kind of mechanisms they do use. In the fifth section we concentrate on the tools that support adaptation of service platforms both on service level and on platform level. Adaptation described in this section also entails certain changes in the monitoring engine execution. In the last section we provide a summary of the tool suite for Deployment, Monitoring & Controlling of Virtual Service Platforms.

2 Integration of the tool suite with supporting tools

2.1 Goal-based modeling Framework

The elicitation of the requirements of the virtual platforms will be carried out through IRET (INDENICA Requirements Elicitation Tool). IRET is the Eclipse-based framework that supports IRENE (INDENICA Requirements ElicitatioN mEthod). IRENE is a goal-based requirements elicitation approach that blends goals, adaptation capabilities, and variability into a single coherent solution.

Functional and non-functional requirements are rendered through goals. Adaptation capabilities, at requirements level, are specified through special-purpose goals called *adaptation* goals, and variability is added in the form of textual comments entered through particular forms. Different goal models (coming from different applications or from the viewpoint of different stakeholders), or different views on the same model can then be merged by means of simple syntactic tools that help the user merge commonalities, highlight differences, and negotiate among the different alternatives. Interested reader can refer to deliverable [D1.2.1] for a complete presentation of the approach.

As for the tool, IRET is implemented on top of GMF/EMF and provides the user with the usual modeling capabilities required to elicit and specify requirements. Users can adopt an informal approach towards requirements, and thus associate simple textual comments to the different elements, or a more detailed and formal view on requirements, and thus also exploit the formal languages supplied by IRENE.

Produced models, that is, requirements specifications, can be used and handled in two different ways. Since IRET is an Eclipse plug-in, based on EMF, each concept is materialized in a specific object within the system and can be accessed through the standard interfaces provided by Eclipse. In addition, models can also be serialized into XML documents, and then be manipulated through the “well-known” tools. These two options are also the two ways IRET, and its artefacts, can be integrated into the complete INDENICA tool-suite.

2.2 Service Platform Infrastructure Repository and EASy-Producer

To support the creation and maintenance of VSPs, the Service Platform Infrastructure Repository encapsulates design time, deployment time, as well as runtime aspects of service platforms. When the deployment of a concrete service platform instance happens, all variabilities with a binding time of at latest the deployment time have been resolved. The variability resolution is done using the EASy-Producer tools developed in WP2 in an interactive fashion either during development, i.e. before deployment, or, for deployment time variabilities during the deployment process. Thus, at deployment time a partially instantiated variability model which only contains open variabilities to be resolved at runtime is available. This partially instantiated model is used to populate the Service Platform Infrastructure Repository of the service platform instance in order to guide the monitoring and adaptation engines. The repository runtime component is deployed with a service platform instance and contains information about the models used to

generate and create the VSP instance, in addition to variability and deployment configurations. Furthermore, monitoring and adaptation rules, created using the accompanying 'Adaptation rule editor' and 'Monitoring rule editor' tools, are stored in the repository for later use by the monitoring and adaptation engines respectively. Moreover, monitoring data generated by the VSP instance is stored the runtime repository for later analysis. The Service Platform Infrastructure Repository is described in greater detail in Deliverable [D2.3.1].

2.3 Monitoring rule editor

The Monitoring Rule Editor is used to manage monitoring rules stored in the runtime repository component.

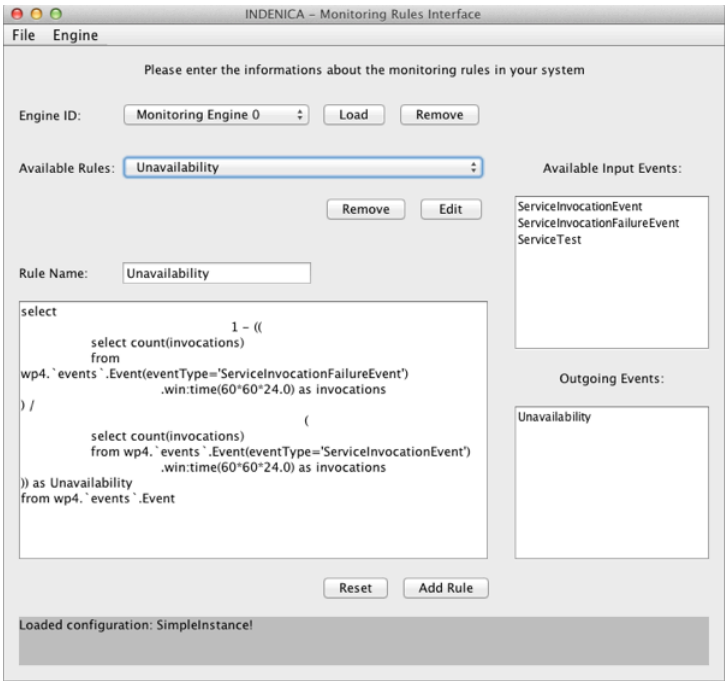


Figure 1: Monitoring Rule Editor showing exemplary monitoring rule

Monitoring rules can either be generated from the VbMF or created manually, and are used by the Monitoring Engines to efficiently and effectively gather runtime data from the integrated service platforms as well as the VSP instance. The Monitoring Rule Editor is described in greater detail in Deliverable [D2.3.1].

2.4 Adaptation rule editor

The Adaptation Rule Editor is used to manage adaptation rules stored in the runtime repository component.

Figure 2: Adaptation Rule Editor showing exemplary adaptation rule

Similar to monitoring rules, adaptation rules can either be generated from the VbMF or created manually, and are used by the Adaptation Engines to effectively adapt the VSP instance and integrated service platforms to changes in the environment in order to always maintain the best possible performance, availability and/or scalability according to business requirements. The Adaptation Rule Editor is described in greater detail in Deliverable [D2.3.1].

2.5 INDENICA Tool Suite for VSP Engineering

The integration of INDENICA View-based Tool Suite (VbMF)[D3.1, D3.3.2] with the tool suite presented in this document is to leverage VbMF's view models in order to (semi-)-automatically generate rules for the monitoring component mentioned in Section 4 and deployment descriptions for the deployment component mentioned in Section 3.

2.5.1 Generation of monitoring rules based on SLA modeling

VSPs and service platforms will be monitored at runtime in order to track the health of the system or to measure certain metrics. The view-based code generation tools described in [D3.1] and the Runtime View are used to generate rules used by the Monitoring Engine. In Section 5.5 of [D3.1], we presented in detail how VbMF views and techniques can support to describe and generate event-based rules that can be fed to some complex event processing framework such as Esper[1] for monitoring systems' and services' SLA properties such as availability, execution time, response time, to name but a few. The main goal of the integration of VbMF and the monitoring component is to implement a richer low-level view that refines the concepts of the aforementioned Runtime View and, based on this low-level view, generates rules and directives in Esper for monitoring the events occurring in VSPs and/or, partially, generates necessary code for monitoring VSP services.

[1] http://esper.codehaus.org

2.5.2 Generation of deployment descriptors

VbMF Deployment View provides high-level concepts such as the artefacts to be deployed and the nodes where the artefacts are hosted. The aforementioned Deployment View will be refined down to support relevant deployment concepts for the corresponding runtime environment where INDENICA VSPs are executed, which is Apache Tuscany for SCA 1.0. In particular, artefacts to be deployed are SCA composites each of which can comprise a number of service components (i.e., SCA components). The SCA 1.0 specification [OSOA] allows an SCA Composite and related artefacts, for instance, its implementation and its required interfaces and components, to be grouped and deployed in a managed unit called *contributions*. A logical group of such SCA contributions that forms an area of business functionality controlled by a single organization, for instance, whole of a business or a department within a business [OSOA] can be deployed and managed in a larger unit called SCA *domain*. Based on the Deployment View, we can generate SCA composite and contribution descriptions that assemble corresponding components of the VSP under consideration and successfully deploy those for executing in an SCA runtime (i.e., Apache Tuscany) [c.f. D3.3.2, D5.3.2]

3 Tool suite for deployment of Virtual Service Platforms

3.1 Deployment Manager

The virtual service platform has to be deployed in a runtime environment. The deployment consists thereby of three major steps as described in Deliverable D4.1. First, all artefacts have to be packaged into a deployable format. Second, these artefacts have to be uploaded to a (remote) location, so they are available for use in the runtime. The third and last step of deployment is the registration of the artefacts in the runtime, so they are known, can be wired together and are available for other services. For the deployment manager to work, several prerequisites have to be fulfilled:

Maven. As build management tool, Maven is used. A typical VIP-Project will consist of a multi-module maven project. Every module in this project describes a contribution in the sense of SCA, consisting of artefacts, composite descriptions and contribution metadata.

Tuscany. For INDENICA, the Tuscany SCA Java runtime will be used, implementing the OSOA SCA specification 1.0. During the deployment process, information about a running Tuscany instance has to be supplied, e.g. server address, user credentials etc.

SCA Deployment Configuration. Information about every SCA composite has to be supplied. Tuscany needs every composite to be run on a single Node (a Tuscany runtime itself). These composites have to be registered at some Endpoint, so its services are made available. This information is supplied as parameters in the maven descriptor (pom.xml).

The deployment process is initiated with the command `mvnindenica:deploy`, run at the parent project after the packaging phase. Every child project then will be deployed as follows: The artefacts are uploaded to the Tuscany server and are started in the same runtime. After uploading, the artefacts are scanned for contribution metadata and all deployable composites are registered in the Tuscany domain. For every composite, a node will also be registered and started. The address of the node will thereby be determined by the SCA deployment configuration.

3.2 Example of the Deployment Process

The following example describes a sample warehouse application that offers several services, implemented in java. The whole setup consists of two projects, the maven parent project and a maven project for compiling and packaging the services. Details about the Tuscany platform to which to deploy this application are provided in the pom.xml of the parent project:

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>sca.indenica</groupId>
<artifactId>parent</artifactId>
<packaging>pom</packaging>
<version>1.0</version>
```

```
<name>parent</name>

<modules>
<module>../warehouse</module>
</modules>

<build>
<plugins>
<plugin>
<groupId>com.sap.research</groupId>
<artifactId>indenica-maven-plugin</artifactId>
<version>1.0</version>
<configuration>
<tuscany>
<server>127.0.0.1</server>
<mgmtPort>9990</mgmtPort>
</tuscany>
</configuration>
</plugin>
</plugins>
</build>

</project>
```

The domain server as well as the port of the management interface has to be described in the configuration part of the indenica-maven-plugin (in the parent project). This implicitly configures the Tuscany target for every child module of this maven build.

As described earlier, every composite will run in a separate node. The configuration of this node will be expressed in the pom.xml of every child module. The following section of the pom.xml of the *warehouse* module configures the endpoints for the services:

```
<plugin>
<groupId>com.sap.research</groupId>
<artifactId>indenica-maven-plugin</artifactId>
<version>1.0</version>
<configuration>
<scaNode>
<ports>
<catalogs>8101</catalogs>
<currency>8102</currency>
</ports>
</scaNode>
</configuration>
</plugin>
```

The described services must correspond to the defined names for each composite. This configuration is only necessary, if composites exist. The SCA composite descriptions have to reside in the root of the archive to be deployed. A typical packaged Indenica SCA project structure looks like the following example:

```
warehouse.jar
├catalogs.composite
├currency.composite
├   warehouse.html
│
```

```
├──META-INF
│    ├   MANIFEST.MF
│    └sca-contribution.xml or sca-contribution-generated.xml
│
└──services
├Cart.class
├Catalog.class
├Item.class
├Order.class
├ShoppingCartImpl.class
├Total.class
├Warehouse.class
    │
    └──currency
├CurrencyConverter.class
└CurrencyConverterImpl.class
```

Important for the deployment process are the bold files. The deployment manager will process each of these files to gain enough information for the deployment activity.

Running the `mvnindenica:deploy`command results in the deployment of the two composites *catalogs* and *currency* to the nodes with the endpoints 127.0.0.1:8101 rsp. 127.0.0.1:8102.

To undeploy the composites, run the `mvnindenica:undeploy` command on the parent project.

4 Tool suite for monitoring of Virtual Service Platforms

4.1 SPASS-meter

SPASS-meter[2] is a flexible resource monitoring framework which enables observing the resource consumption of individual parts of a (Java) software system at runtime. Regarding resources, SPASS-meter supports the collection of runtime information about execution time (CPU and response time), memory consumption (allocation and use), file transfer and network transfer. Typical resource monitoring tools provide either large amounts of detailed information which is not related to individual services (e.g. InsECT[CO04], J-RAF2 [BH04] or OpenCore[JI12]) or which is aggregated only on system level (e.g. JMX [O08]) representing summarized effects of all running services. Avoiding such superfluous collection, and, thus, reducing the so called monitoring overhead, is one of the main aims of SPASS-meter. SPASS-meter and its architecture have already been discussed in D4.2.1. In this section, we will summarize SPASS-meter here only briefly and discuss the improvements over D4.2.1.

Therefore, SPASS-meter must be configured for a specific system under monitoring (SUM, e.g., a service platform instance) to gather information on the services of interest. The configuration may be given in terms of source code annotations or as an external XML file. Depending on that configuration, SPASS-meter focuses particularly on the resource consumption of the specified parts of the SUM and, if configured appropriately, it collects also information about the SUM as a process or on the whole execution environment in order to support deriving relative statements about the SUM. SPASS-meter collects its information based on raw data obtained from an instrumented SUM, more precisely from a SUM modified for monitoring according to the SUM-specific configuration. Based on that configuration, SPASS-meter may preparethe SUMprior to, at runtime or in a mixed way. Each of these three instrumentation modesmay be beneficial with respect to overhead and applicability to certain (limited) environments. Instrumentation prior to runtime avoids runtime instrumentation overhead but cannot exploit information known only at runtime, e.g., the actual target of polymorphic method calls. Instrumentation at runtime can take such information into account and even optimize the actual instrumentation dynamically. Such runtime optimization is done by SPASS-meter to significantly improve the performance of monitoring the memory consumption of the SUM. Therefore, it dynamically adjusts the instrumentation so that only relevant parts of the SUM are under observation. However, dynamic instrumentation consumes also some (additional) resources, thus, causing runtime overhead. In service-oriented systems, the mixed mode may be applied, e.g., static instrumentation for the platform and dynamic instrumentation for dynamically loaded services which are unknown at static instrumentation time.

[2] SPASS is the acronym for Simplifying the Development of Adaptive Software Systems and SPASS-meter is one of the foundational building blocks of our approach in this field. In German, the term “Spass” means “fun” and points to the tons of fun the developers had and will have while realizing the approach and its tooling.

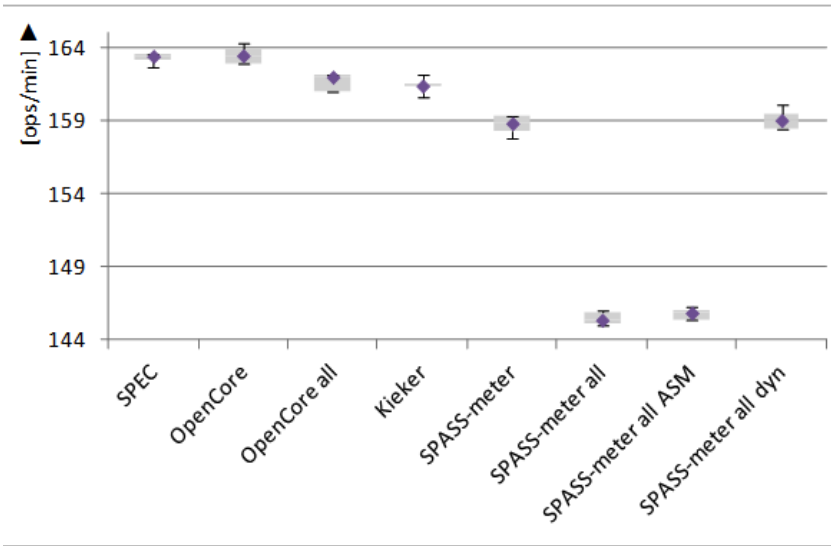


Figure 3: Execution time overhead comparison in terms of SPECjvm2008 operations per minute

In the third year of INDENICA, SPASS-meter has been optimized in order to reduce the monitoring overhead. Basically, strategies such as explicit trade off making in the implementation or pre-aggregation at very early stages of the data collection helped reaching that goal. Further, we analysed the performance of SPASS-meter under load using the well-known (and freely available) SPECjvm2008 benchmark suite[3] as experimental subject. Thereby, we compared SPASS-meter with recent monitoring tools such as OpenCore or Kieker running under the same conditions in the same environment. However, there are various differences between OpenCore[JI12], Kieker (e.g., [HWH12])and SPASS-meter, in particular regarding the approach, the configuration possibilities, the online analysis capabilities and the offered resources to be monitored (details can be found in [ES12]). Therefore, we compared the tools with respect to their common capabilities, i.e., execution time monitoring. For the experiments, we used a recent Dell Optiplex 790MT running Ubuntu Linux 12.04.1 64 bit in server configuration and JDK 1.6.0_34 64 bit as Java environment. Our analysis [ES12][4]shows that the commercial tool OpenCore runs below 1% (two similar configurations in Figure 3), Kieker around 1.2% and SPASS-meter at 2.8% ("SPASS-meter" in Figure 3) execution time overhead. The increased overhead of SPASS-meter is mainly due to the complex runtime aggregation capabilities required to determine the resource consumption of individual parts of the SUM. Compared from the output perspective, SPASS-meter produces a handy and focused summary of some Kbytes while the other two tools produce large log files for offline analysis (in the order of Mbytes for OpenCore and in the order of GBytes by Kieker). Further, monitoring network and file transfer with SPASS-meter leads to similar overheads, i.e., 2.8% execution time overhead (not shown in Figure 3), while memory allocation and memory use can be observed at total 10.8% ("SPASS-meter all", and "SPASS-meter all ASM" in Figure 3) just instrumenting and monitoring all memory

[3]http://www.spec.org/jvm2008/

[4]A journal artical about SPASS-meter, its approach, its architecture and an even more detailed comparative analysis is under submission.

consumptions or 3.9% runtime overhead using dynamic optimization ("SPASS-meter all dyn" in Figure 3). Regarding memory consumption overhead, all three tools operate approx. 0.5% overhead.

The documentation about usage, installation and setup of the SPASS-meter is available as Appendix 2: SPASS-meter Quick Guide to this document.

4.2 ECoWare

ECoWare, which stands for Event Correlation Middleware, is a distributed data aggregation and persistency tool. In INDENICA ECoWare is used to provide means to reason on a complex service-based system, by aggregating raw data collected from its multiple layers. It allows us to correlate behaviours being seen in the Service Platform Environment with behaviours being seen at the underlying virtual resources layer.

A typical ECoWare deployment (see Figure 4) consists of four different types of components: (i) the execution environments for which we want to collect run-time data (together with appropriate probes), (ii) a series of processors for providing the actual data aggregations, (iii) a persistency database, and (iv) theECoWare Dashboard for visualizing the aggregated data. The components collaborate through a RabbitMQ Publish and Subscribe (P/S) event bus for which ECoWare defines a normalized event format. In order to manage the normalized format and to collaborate properly, each component is required to implement appropriate RabbitInputAdapters and RabbitOutputAdapters.

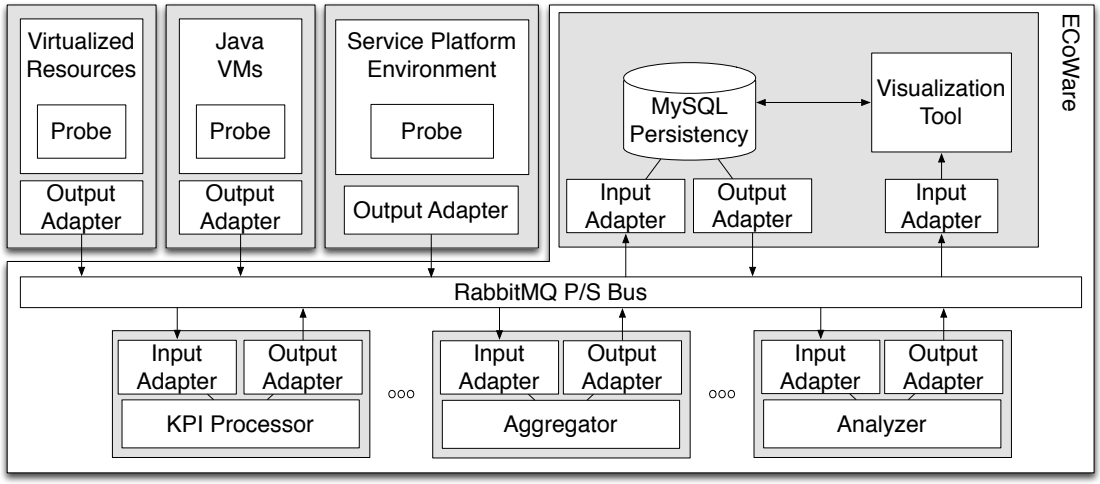


Figure 4: Architecture of the ECoWare Framework.

The execution environments can be of any kind. For example, we could be probing the Service Platform Environment, the Java VMs being used, and the underlying virtualized resources. This allows us to aggregate information coming from all three of these layers, with the goal of incrementally constructing comprehensive knowledge of the running system along its multiple layers.

Data processing is provided by three different kinds of processors QoS Processors, Aggregators, and Analyzers. These processors subscribe to events on the P/S bus, execute their internal logic, and publish their results back to the bus. This allows them to be strung together to obtain the aggregated information we desire. Thanks

to this loose coupling ECoWare can be considered very extensible. In the future we will investigate a stronger integration with the other monitoring tools produced in the context of the Indenica project.

QoS Processors and Aggregators are built using Esper, a component for complex event processing activities. Esper components execute queries defined using an Event Processing Language (EPL). EPL is an SQL-like language for developing event conditions, correlations, and aggregations. The difference between SQL and EPL is that, instead of running queries against stored data, Esper stores queries and runs data through them. The execution model is thus continuous rather than limited to the exact moment at which the query is submitted. Analyzers are built using a simple assertion analyzer inspired by our previous work on WSCoL [WSCoL]. WSCoL was developed for the definition of functional monitoring of BPEL processes, and has been modified to receive data that are not in XML form.

The ECoWare dashboard is a Java Desktop application that system managers can use to visualize data collected through ECoWare. It supports both live charting, and online and offline violation drill-down analysis. Online and offline drill-down analysis allows managers to choose a violation, and visualize the multiple data collections that were triggered by that event. Violations, as well as the correlated data are collected on-demand from ECoWare's persistency database, where they are automatically collected every time an aggregator is triggered. Like for certain security cameras, we currently store data for 24 hours, and periodically cleanse the data that are no longer needed.

New features in ECoWare

The main novel contribution to ECoWare itself is that it now has a simple declarative language called mlCCL that can be used to identify the data we want to collect from the running system, how we want to aggregate them to build higher-level knowledge, and how we want to analyse the data to discover undesired behaviours. mlCCL stands for „Multi-layer Collection and Constraint Language". (mlCCL currently only supports SCA-based systems. In the future, it will be extended to support other kind of software systems.)

mlCCL is based on the assumption that data are described in terms of Service Data Objects (SDOs). A Service Data Object is a language-agnostic data structure typically used to facilitate the communication between diverse service-based entities. A data object consists of a set of named properties. Each property can be either single- or multi-valued (i.e., an array), and can have either a primitive (i.e., a number, a string, or a Boolean) or complex data-type (i.e., a service data object). mlCCL allows for 2 kinds of data collections: messages and indicators.

Message collection is all about capturing the messages that are sent or received by a SCA component. In mlCCL we need to identify a location using the syntax

locName = [before | after] endpoint;

where *locName* is a location name chosen by the designer, *before* and *after* are mlCCL keywords, and *endpoint* is a (methodName, [serviceName | referenceName], componentName) triple that uniquely identifies a method in a SCA-based system. At

this point a message collection is expressed as

aliasName = collect(locName);

where aliasName is an alias that can be used to refer to the collected SDOs, and collect is a function that takes a location as its sole parameter. The SDO contains the collected message, as well as the aliasName and location values used in the collection specification. On top of that, it also contains a timestamp indicating "when" the message was sent or received by the service runtime2, and an instanceID, that is a unique ID that identifies the specific service call. Corresponding request and response messages share this unique ID.

Indicator Collection is all about collecting periodic information, i.e., information that is not triggered by any specific service call. An indicator can be a KPI (Key Performance Indicator) or a RI (Resource Indicator). In this case we use the following syntax:

aliasName = collect(indicatorName, serviceID, outputRate, pastWindow, property);

where *aliasName* is the alias that will be used to refer to the collected SDOs. *indicatorName* is the name of the indicator we want to collect, while *serviceID* identifies the service for which we are requesting the indicator. In case we are collecting a KPI, the *serviceID* is an endpoint triple, like the ones defined for message collection but without the *before* or *after* keywords. In case we are collecting a RI, the *serviceID* is the unique ID of a virtual machine surrounded by brackets. The *outputRate* identifies the frequency with which a new indicator value should be produced, while the *pastWindow* parameter specifies the amount of past time to take into consideration when calculating the new value. A value, and a time unit that can be "seconds", "minutes", or "hours", defines the *pastWindow* parameter. Finally, *property* is a mlCCL data analysis expression that is used by indicators that, in order to be calculated, need to know how many SDOs, collected in the last *pastWindow*, satisfy a given property

Currently mlCCL supports four KPIs (i.e., avgRT, arrivalRate, throughput, and reliability), and fifteen RIs (i.e., cpuIdle, cpuUser, and cpuSystem for CPU activity; diskOctets and diskOps for disk activity; memUsed, memBuffered, memCached, and memFree for memory activity; and netIncoming, netOutgoing, netPacketsTX, netPacketsRX, netErrorsTX, netErrorsRX for network activity).

Once the SDOs have been collected we can aggregate them. Since different SDOs are collected at different times, the aggregation needs to specify "when" it should be achieved. We solve this issue by requiring that the designer specify a "primary/triggering" SDO. The syntax we use is:

aliasName = aggregate(primary, list);

where *aliasName* is the alias of the new aggregated SDO, *primary* is a previously defined alias, and *list* is a comma-separated list of previously defined aliases. Every time the framework sees a new *primary* SDO, it automatically collects the last known

SDO of each secondary type. If the designer wants a past window of SDOs for a given secondary type, this can be achieved by appending *window(interval)* to the secondary SDO type.

mlCCL also provides means to analyse certain properties of the collected SDOs. In mlCCL we refer to a SDO's internal properties appending method *get(propertyName)* to the SDO's alias. Depending on the type of property we extract, mlCCL also provides a number of methods for further manipulation, such as *absolute value* and *square root* for numbers; and *substring*, *length* and *replace* for strings. If the SDO is an array, we can obtain the *length* of the array, the *i-th value* in the array, or a *subset* of all the values that satisfy a given property. In this case the property can refer to the elements in the array using a pre-defined *elem* alias. Finally, if the SDO is an array of numbers we can obtain the *sum*, the *average*, or the *minimum* or *maximum* values of the array. mlCCL allows us to define an expression that predicates over the properties of SDOs. The expression is expressed using a slight variation of the WSCoL language. We use the following syntax:

⟨prop⟩ ::= ¬⟨prop⟩ | ⟨prop⟩&&⟨prop⟩ | ⟨prop⟩||⟨prop⟩ | ⟨array⟩ . ⟨quant⟩ (⟨prop⟩) |
⟨term⟩⟨rop⟩⟨term⟩
⟨term⟩ ::= ⟨prim⟩ | ⟨term⟩⟨aop⟩⟨term⟩ | ⟨const⟩
⟨rop⟩ ::=<|≤|==|≥|>
⟨quant⟩::= forall | exists
⟨aop⟩ ::=+|−|×|÷|%

where *prim* is a primitive value (i.e., number, string, or boolean) extracted from a SDO, and *array* is an array of primitive values, all of the same type. Boolean, relational (*rop*), and arithmetic operators (*aop*) follow their usual definitions. *forall* and *exists* state that a property should hold for all, or for at least one, of the values in an *array*. In this case the *property* can refer to the elements in the array using an implicit *$elem* alias.

Like aggregation, analysis also requires a "primary/trigger" SDO, so that it can know when all the data needed for the analysis is available, and when it should activate. The syntax is therefore

aliasName = evaluate(primary, expression);

where *aliasName* is the alias that is used to identify the SDO that is created every time expression holds, while *primary* is a previously defined alias that is used within *expression*. When the primary alias is collected, the evaluation of the expression is triggered. Examples of how we use mlCCL can be found in the new version of Appendix 1.

Integration with other project contributions

During the third year of the Indenica project we have made two contributions that go in the direction of strengthening the integration with the other project contributions. The first is that we have removed the Siena P/S bus from ECoWare's architecture, and replaced it with a RabbitMQ bus. The reason for this is that this

event distribution bus is being used by the other partners as well. This meant changing all our Siena Input and Output Adaptors to RabbitMQ adapters.

In the previous year, Resource Indicators were collected from a VM using a third-party tool called collected. Now, we have introduced new Resource Indicators that focus on the Java VM that is running the software system. This allows us to gather information from a layer that is intermediate between the actual SCA application, and the virtualized resources that are used for the system's provisioning. The data coming from the JVM are collected using SPASS-meter. This means we can now collect data about execution times (CPU and response time), memory consumptions (allocation and use), file and network transfers. Currently, the SPASS-meter resource indicators are not supported inside mICCL. They are, however, supported through XML-based configuration.

Furthermore, we have used ECoWare in the context of the Remote Maintenance System Platform. To do this we developed specific probes that can be used to capture the interactions that occur with the platform's core, which consists of Java servlets deployed to a JBoss Application Server. We also were able to capture information regarding the interactions that servlets had with the backend database. More information regarding the use of ECoWare in the Remote Maintenance System Platform can be found in Deliverable D5.4.

Online presence

During the third year we also developed an online presence for ECoWare. We have developed a complete tutorial on how to install ECoWare, and how to use it. The tutorial consists of a five step tour of ECoWare's main features, and allws developers to be up and running within minutes. The publicly available version of ECoWare also provides full JavaDoc documentation of the source code.

To increase ECoWare's visibility we published a paper in the IEEE 20th International Conference on Web Services (ICWS 2013) entitled „Event-based Multi-level Service Monitoring" [ECoWare].

4.3 MONINA

MONINA (MONitoring, INtegration, Adaptation) is a DSL for concise, easy and reusable specification of platforms integrated into a VSP, along with monitoring and adaptation rules governing their behaviour. The language is developed using the Xtext[5] framework, allowing for tight integration in the Eclipse platform. The plugin offers syntax highlighting, as well as several automated sanity checks to ease system specification. The language plugin is furthermore integrated into the overall Indenica tool suite, allowing for the usage of existing system models stored in the infrastructure repository. Future versions of the plugin will offer a graphical abstraction in addition to the textual DSL for increased simplicity and ease of use.

Figure 5shows a simple definition for a service platform to be integrated into a VSP. The 'ApplicationServer' component emits 'RequestFinished' events after processing requests and supports a 'DecreaseQuality' action, which can be triggered by

[5] http://www.eclipse.org/Xtext/

adaptation rules. Emitted events are processed by the 'AggregateResponseTime' query, which aggregates them over five minutes, creating an 'AverageProcessingTime' event. This event is converted to a fact, which might trigger 'DecreaseQualityWhenSlow' adaptation rule. The physical infrastructure consists of hosts 'vm1' and 'vm2'. Runtime elements without defined costs are assigned default values, which are refined at runtime. In the following we will discuss the most important language constructs of MONINA in more detail.

```
event RequestFinished {
  request_id : Integer
  processing_time_ms : Integer
}

event AverageProcessingTime {
  processing_time_ms : Integer
}

action DecreaseQuality {
  amount : Double
}

component ApplicationServer {
  endpoint {
    at "/app_server"
    emit RequestFinished
    action AdjustQuality
  }
  host vm1
  cost 32
}

host vm1 { capacity 128 }
host vm2 { capacity 256 }

query AggregateResponseTimes {
  from ApplicationServer
  event RequestFinished as e
  emit AverageProcessingTime(
    avg(e.processing_time_ms))
  window 5 minutes
}

fact {
  from AverageProcessingTime
}

rule DecreaseQualityWhenSlow {
  from AverageProcessingTime as f
  when f.processing_time_ms > 2000
  execute ApplicationServer
          .DecreaseQuality(5)
}
```

Figure 5: Sample MONINA System Definition

event Monitoring events are described listing attributes contained in emitted messages. Events are then used in component definitions, monitoring query declarations, as well as facts.

fact Facts constitute the knowledge base for adaptation actions. Fact definitions reference an event type and a partition key.

action Similar to events, adaptation actions list all their valid parameters. Actions are used in component definitions as well as adaptation rules.

component A component definition references all monitoring events the platform can emit (including their frequency), all adaptation actions that can be performed, as well as its processing requirements. Furthermore, it is correlated with a concrete instance of the component in question at deployment.

query Monitoring queries are used to define the aggregation, filtering and enrichment of emitted monitoring data in a CEP fashion. Monitoring rules will either emit complex aggregated events to be consumed by other monitoring rules, directly issue adaptation actions, or emit facts to be used in adaptation rules.

rule Adaptation rules allow for the usage of complex business management rules to govern system behaviour. Monitoring rules emit facts to be used for

reasoning over the current system state. Adaptation rules can either publish new facts or issue adaptation actions.

host Hosts represent possible deployment locations of components, monitoring queries and adaptation rules. A host description contains its processing capacity.

4.3.1 Event

In our work, we follow the event-based interaction paradigm. Events are emitted by components to signal important information. Furthermore, events can be emitted by monitoring queries as a result of the aggregation or enrichment of one or more source events.

Figure 6 shows a simplified grammar of the event construct in Extended Backus-Naur Form (EBNF). Event declarations start with the event keyword and an event type identifier. As shown in the figure, an event can contain multiple attributes, defined by specifying name and type separated by a colon. Currently, supported event types are a variety of Java types such as `String`, `Integer`, and `Decimal`, and `Map<?,?>`.

⟨*event*⟩ ::= 'event' ⟨*ID*⟩ '{' ⟨*attr*⟩* '}'

⟨*attr*⟩ ::= ⟨*attr-name*⟩ ':' ⟨*type*⟩

⟨*attr-name*⟩ ::= ⟨*ID*⟩

Figure 6: Simplified Event Grammar in EBNF

Since listing all available event types for every application would be a tedious and error-prone task, we automatically gather emitted event types from known components to improve reusability and ease of use. This procedure is described in more detail in Section 4.3.4.

4.3.2 Action

Complementary to monitoring events described above, adaptation actions are another basic language element of MONINA. Adaptation actions are invoked by adaptation rules and executed by corresponding components to modify their behaviour. Figure 7 shows a simplified grammar of the action construct in EBNF. Action declarations start with the action keyword followed by the action type identifier. Furthermore, actions can take parameters, modelled analogously to event attributes shown in Figure 6.

⟨*action*⟩ ::= 'action' ⟨*ID*⟩ '{' ⟨*attr*⟩* '}'

Figure 7: Simplified Action Grammar in EBNF

Similar to events, adaptation actions offered by known components do not need to be specified manually, but are automatically gathered from component specifications, as mentioned in Section 4.3.4.

4.3.3 Fact

Facts constitute the knowledge base for adaptation rules and are derived from monitoring events. A fact incorporates all attributes of the specified source event for

use by adaptation rules. Figure 8 shows a simplified grammar of the fact construct in EBNF. Fact declarations start with the fact keyword and an optional fact name. A fact must specify a source event type that is used to derive the fact from. Furthermore, an optional partition key can be supplied. If the fact name is omitted, the fact will be named after its source event.

⟨*fact*⟩ ::= 'fact' ⟨*ID*⟩? '{' ⟨*ID*⟩ ⟨*partition-key*⟩? '}'
⟨*partition-key*⟩ ::= 'by' ⟨*ID*⟩

Figure 8: Simplified Fact Grammar in EBNF

The partition key construct is used to enable the creation of facts depending on certain event attributes, allowing for the concise declaration of multiple similar facts for different system aspects. For instance, a fact declaration for the event type `ProcessingTimeEvent` that is partitioned by the component id attribute will create appropriate facts for all encountered components, such as `ProcessingTime(Component1), . . . , ProcessingTime(ComponentN).` In contrast,a fact declaration for the `MeanProcessingTimeEvent` without partition key will result in the creation of a single fact representing the system state according to the attribute values of incoming events.

4.3.4 Component

A component declaration incorporates all information necessary to integrate third-party system into the Indenica infrastructure. Fig. 5 shows a simplified grammar of the component construct in EBNF. Component declarations start with the **component** keyword and a component identifier. A component specifies all monitoring events it will emit with an optional occurrence frequency, supported adaptation actions, as well as a reference to the host the component is deployed to.

⟨*component*⟩ ::= 'component' ⟨*ID*⟩ '{' ⟨*metadata*⟩? ⟨*c-elements*⟩* ⟨*host-ref*⟩ '}'
⟨*metadata*⟩ ::= ('vendor' ⟨*STRING*⟩)? ('version' ⟨*STRING*⟩)? . . .
⟨*c-elements*⟩ ::= ⟨*endpoint*⟩ | ⟨*refs*⟩
⟨*refs*⟩ ::= ⟨*event-ref*⟩ | ⟨*action-ref*⟩
⟨*action-ref*⟩ ::= 'action' ⟨*ID*⟩
⟨*event-ref*⟩ ::= 'event' ⟨*ID*⟩ ⟨*frequency*⟩?
⟨*endpoint*⟩ ::= 'endpoint' ⟨*ID*⟩? '{' ⟨*e-addr*⟩ ⟨*refs*⟩* '}'
⟨*frequency*⟩ ::= 'every' ⟨*Decimal*⟩ 'seconds' | ⟨*Decimal*⟩ 'Hz'
⟨*host-ref*⟩ ::= 'host' ⟨*ID*⟩

Figure 9: Simplified Component Grammar in EBNF

For brevity, further elements such as endpoint addresses, are omitted in the presented grammar snippet but are included in the implementation.

As mentioned before, it is usually not necessary to manually specify component, action, and event declarations. The Indenica infrastructure provides for means to automatically gather relevant information from known components through the control interface that is part of the runtime infrastructure.

4.3.5 Monitoring Query

Monitoring queries allow for the analysis, processing, aggregation and enrichment of monitoring events using CEP techniques. In the context of the Indenica project we provide a simple query language tailored to the needs of the specific solution.

A simplified EBNF grammar of the monitoring query construct is shown in Figure 10. A query declaration starts with the **query** keyword and a query identifier. Afterwards, an arbitrary number of event sources for the query is specified using the **from** and **event** keywords to specify source components and event types. A query then specifies any number of event emission declarations, denoted by the **emit** keyword followed by the event type and a list of expressions evaluating the attribute assignments of the event to be emitted. For brevity we omit the specification of the ⟨*cond-expression*⟩ clause that represents a SQL-style conditional expression. Queries can be furthermore designed to operate on event stream windows using the **window** keyword, specifying either a number of events to create a batch window or a time span to create a time window. Conditions expressed using the **where** keyword are used to limit query processing to events satisfying certain conditions, using the conditional expression construct mentioned above. Finally, queries can optionally indicate the rate of incoming vs. emitted events, as well as an indication of required processing power. These values are user-defined estimations in the initial setup, and are adjusted continuously during runtime to accommodate changes in the environment.

⟨*query*⟩ ::= 'query' ⟨*ID*⟩ '{' (⟨*sources*⟩ | ⟨*emits*⟩)*
⟨*window*⟩? ⟨*condition*⟩? ⟨*io-ratio*⟩? ⟨*cost*⟩? '}'

⟨*sources*⟩ ::= 'source' ⟨*ID*⟩ (',' ⟨*ID*⟩)*
'event' ⟨*ID*⟩ (',' ⟨*ID*⟩)*

⟨*emits*⟩ ::= 'emit' ⟨*ID*⟩ (⟨*attr-emit*⟩*)*

⟨*attr-emit*⟩ ::= ⟨*cond-expression*⟩ ('as' ⟨*ID*⟩)?

⟨*window*⟩ ::= 'window' (⟨*batch-window*⟩ | ⟨*time-window*⟩)

⟨*batch-window*⟩ ::= ⟨*Integer*⟩ 'events'

⟨*time-window*⟩ ::= ⟨*Integer*⟩ ('seconds' | 'minutes' | 'days' | ...)

⟨*condition*⟩ ::= 'where' ⟨*cond-expression*⟩

⟨*io-ratio*⟩ ::= 'ratio' ⟨*Decimal*⟩

⟨*cost*⟩ ::= 'cost' ⟨*Decimal*⟩

Figure 10: Simplified Monitoring Query Grammar in EBNF

In addition to the query construct presented above, the language infrastructure allows for the integration of other CEP query languages, such as the Esper Event Processing Language[6] (EPL) if necessary.

[6] http://esper.codehaus.org/esper-4.10.0/doc/reference/en-US/html/epl_clauses.html

4.3.6 Adaptation Rule

Adaptation rules employ a knowledge base consisting of facts to reason on the current state of the system and modify its behaviour when necessary using a production rule system. Figure 11 shows a simplified grammar of the adaptation rule construct in EBNF. A rule declaration starts with the **rule** keyword and a rule identifier. After importing all necessary facts using the **from** keyword, a rule contains a number of **when**-statements where the condition evaluates a ⟨*cond-expression*⟩ as described above, referencing imported facts, and the **then** block specifies a number of adaptation action invocations including any necessary parameter assignments. Optionally, a rule can indicate processing requirements (cf. Figure 10) that will be adjusted at runtime.

⟨*rule*⟩ ::= 'rule' ⟨*ID*⟩ '{' (⟨*r-sources*⟩)+ ⟨*stmt*⟩+ ⟨*cost*⟩?'}'

⟨*r-sources*⟩ ::= 'from' ⟨*ID*⟩ ('as' ⟨*ID*⟩)?

⟨*stmt*⟩ ::= 'when' ⟨*cond-expression*⟩ 'then' ⟨*action-expr*⟩+

⟨*action-expr*⟩ ::= ⟨*ID*⟩ ' (' ⟨*action-attr*⟩ (',' ⟨*action-attr*⟩)* ')'

⟨*action-attr*⟩ ::= ⟨*cond-expression*⟩ ('as' ⟨*ID*⟩)?

Figure 11: Simplified Adaptation Rule Grammar in EBNF

As with monitoring queries, the adaptation rule module is tailored to the requirements of the Indenica infrastructure but also allows for the usage of different production rule languages, such as the Drools [7] rule language, if more complex language constructs are required.

4.3.7 Host

Hosts represent the physical infrastructure available for deployment of infrastructure components. Figure 12 shows a simplified grammar of the host construct in EBNF. A host declaration starts with the host keyword and a host name. An address in the form of a fully qualified domain name (FQDN) or an IP address can be supplied. If no address is given, the host name will be used instead. Furthermore, a capacity indicator is provided that will be used for deployment decisions.

⟨*host*⟩ ::= 'host' ⟨*ID*⟩ '{' ⟨*address*⟩? ⟨*capacity*⟩ '}'

⟨*address*⟩ ::= ⟨*fqdn*⟩ | ⟨*ip-address*⟩

⟨*capacity*⟩ ::= 'capacity' ⟨*Decimal*⟩

Figure 12: Simplified Host Grammar in EBNF

4.4 Monitoring engines

The INDENICA infrastructure provides an extensible platform monitoring framework employing novel concepts for organizing and layering monitoring concerns to allow for efficient distribution of software components to reduce communication overhead. Furthermore, sophisticated processing methods, such as data ageing,

allow for the effective usage of historical system health data while keeping transmission and storage overhead minimal.

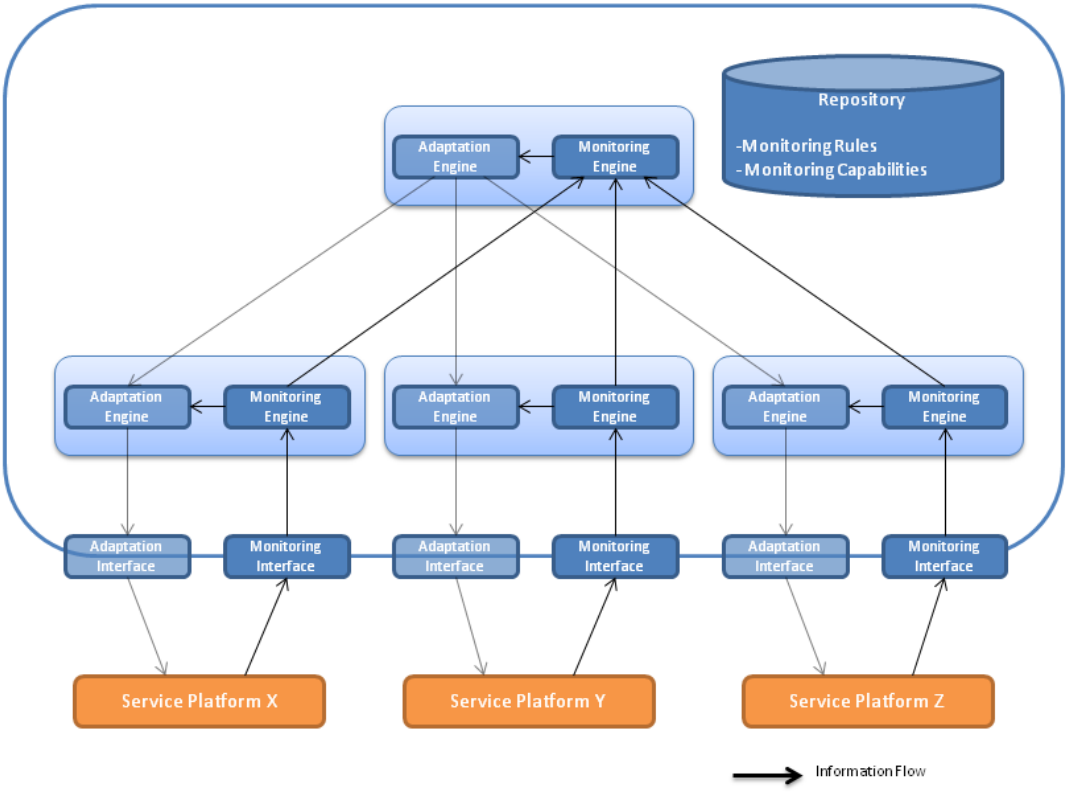


Figure 13: Runtime Infrastructure Architecture: Monitoring Overview

The monitoring infrastructure is instantiated according to configuration directives deployed to the repository (cf. Section 2.2) by creating all necessary monitoring engines and establishing connections to the integrated service platforms. Additionally, connections between monitoring engines and according adaptation engines are set up.

4.5 Domain-specific events

Events monitored in the VSP come from different service platforms. In order to allow efficient monitoring of events the necessary unification of these events has to be done. INDENICA proposes an extensible event model to which all underlying platforms need to be compliant with to catch all messages. The team identified three major points of unification.

First is the format that is used to define messages, the second is the internal structure of these messages and third is the type of messaging technique that is used to gather events. Events that are sent by the underlying service platforms may be written in different formats such as JSON [JSON], XML [XML], key-value list, etc. and several mechanisms can be used to get them, e.g. publish-subscribe, polling, streaming, etc.

4.5.1 Events in Remote Maintenance System

The first Use Case example that we use is the Remote Maintenance System which uses XML-based events which are sent to the monitoring engine using Rabbit MQ message broker.

XML Schema

Generated events are compliant with the following XML Schema:

```
<?xmlversion="1.0"encoding="UTF-8"?>
<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:elementname="event"type="EventType"/>

<xs:complexTypename="EventType">
<xs:sequence>
<xs:elementname="eventId"type="xs:long"minOccurs="1"/>
<xs:elementname="eventName"type="xs:string"minOccurs="1"/>
<xs:elementname="timestamp"type="xs:dateTime"
      minOccurs="1"/>
<xs:choice>
<xs:elementname="request"type="RequestType"/>
<xs:elementname="response"type="ResponseType"/>
<xs:elementname="systemParameters"
      type="SystemParametersType"/>
</xs:choice>
<xs:elementname="errorMessage"type="ErrorMessageType"
      minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexTypename="RequestType">
<xs:sequence>
<xs:elementname="requestId"type="xs:long"minOccurs="1"/>
<xs:elementname="adminId"type="xs:string"minOccurs="0"/>
<xs:elementname="userSipId"type="xs:string"minOccurs="0"/>
<xs:elementname="cameraSipId"type="xs:string"minOccurs="0"/>
<xs:elementname="cameraIpAddress"type="xs:string"minOccurs="0"/>
<xs:elementname="receiverSipId"type="xs:string"minOccurs="0"
      maxOccurs="unbounded"/>
<xs:elementname="callParameters"type="xs:string"minOccurs="0"/>
<xs:elementname="sessionId"type="xs:string"minOccurs="0"/>
<xs:elementname="userStatusId"type="xs:integer"minOccurs="0"/>
<xs:elementname="xPosition"type="xs:double"minOccurs="0"/>
<xs:elementname="yPosition"type="xs:double"minOccurs="0"/>
<xs:elementname="horizontalAngle"type="xs:double"minOccurs="0"/>
<xs:elementname="verticalAngle"type="xs:double"minOccurs="0"/>
<xs:elementname="zoom"type="xs:double"minOccurs="0"/>
<xs:elementname="targetUserSipId"type="xs:string"minOccurs="0"
      maxOccurs="unbounded"/>
<xs:elementname="videoSourceSipId"type="xs:string"minOccurs="0"/>
<xs:elementname="subscriptionId"type="xs:string"minOccurs="0"/>
<xs:elementname="startTime"type="xs:dateTime"minOccurs="0"/>
<xs:elementname="stopTime"type="xs:dateTime"minOccurs="0"/>
<xs:elementname="textMessage"type="xs:String"minOccurs="0"/>
<xs:elementname="userGroup"type="xs:string"minOccurs="0"/>
<xs:elementname="user"type="UserData"minOccurs="0"
      maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexTypename="ResponseType">
<xs:sequence>
<xs:elementname="requestId"type="xs:long"minOccurs="1"/>
<xs:elementname="responseId"type="xs:long"minOccurs="1"/>
```

```
<xs:elementname="sessionId"type="xs:string"minOccurs="0"
      maxOccurs="unbounded"/>
<xs:elementname="userSipId"type="xs:string"minOccurs="0"/>
<xs:elementname="userStatusId"type="xs:intiger"minOccurs="0"/>
<xs:elementname="user"type="UserData"minOccurs="0"
      maxOccurs="unbounded"/>
<xs:elementname="camera"type="CameraData"minOccurs="0"
      maxOccurs="unbounded"/>
<xs:elementname="subscriptionId"type="xs:string"minOccurs="0"/>
<xs:elementname="fileUrl"type="xs:string"minOccurs="0"/>
<xs:elementname="fileParameters"type="xs:string"minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexTypename="SystemParametersType">
<xs:sequencemaxOccurs="unbounded">
<xs:elementname="parameter"type="xs:string"minOccurs="1"/>
<xs:choice>
<xs:elementname="valueInt"type="xs:integer"/>
<xs:elementname="valueLong"type="xs:long"/>
<xs:elementname="valueDouble"type="xs:double"/>
<xs:elementname="valueDate"type="xs:dateTime"/>
<xs:elementname="valueString"type="xs:string"/>
<xs:elementname="valueBoolean"type="xs:boolean"/>
</xs:choice>
</xs:sequence>
</xs:complexType>

<xs:complexTypename="UserData">
<xs:sequence>
<xs:elementname="userId"type="xs:string"minOccurs="0"/>
<xs:elementname="userSipId"type="xs:string"minOccurs="0"/>
<xs:elementname="userName"type="xs:string"minOccurs="0"/>
<xs:elementname="userPasswd"type="xs:string"minOccurs="0"/>
<xs:elementname="userGroup"type="xs:string"minOccurs="0"/>
<xs:elementname="statusId"type="xs:integer"minOccurs="0"/>
<xs:elementname="userParameters"type="xs:string"minOccurs="0"/>
<xs:elementname="ip"type="xs:string"minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexTypename="CameraData">
<xs:sequence>
<xs:elementname="cameraSipId"type="xs:string"minOccurs="0"/>
<xs:elementname="cameraIpAddress"type="xs:string"minOccurs="0"/>
<xs:elementname="xPosition"type="xs:double"minOccurs="0"/>
<xs:elementname="yPosition"type="xs:double"minOccurs="0"/>
<xs:elementname="verticalAngle"type="xs:double"minOccurs="0"/>
<xs:elementname="horizontalAngle"type="xs:double"minOccurs="0"/>
<xs:elementname="zoom"type="xs:double"minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexTypename="ErrorMessageType">
<xs:sequence>
<xs:elementname="errorMsgId"type="xs:integer"minOccurs="0"/>
<xs:elementname="message"type="xs:string"minOccurs="0"/>
</xs:sequence>
</xs:complexType>

</xs:schema>
```

This XML schema corresponds to events generated by Remote Maintenance subsystem during its functional operations. Main element of the presented XML schema is *event*, which can be composed of only one of its subcomponents; the most

important are three complex elements: *request*, *response*, *systemParameters*. *Request components* used to send data related with web service query. *Response* carries the data related to the web service's server answers. *SystemParameters* element is responsible for data gathered by the machine monitoring (CPU, virtual memory and storage related data). Besides, there are also other complex types, e.g. *user* and *camera which are located* inside *request* and *response elements*. Error handling is performed by an *errorMessage* element, which is added in that case to every other event messages.

Installation

4.5.2 Events in Yard Management Subsystem

The second subsystem that was used to derive a common event model was Yard Management Subsystem, which is sending events formatted based on JSON. Gathering events from this subsystem requires polling to the REST-based endpoint, which in result sends events that were not previously fetched.

The Format of events is a simple 4 attribute JSON structure:

```
{
"id" :"<id>",
"triggered":<timestamp>,
"type":"<string>",
"data":{<object>}
}
```

The description of types and inner structure of the data element can be found in the Table 1 and Table 2.

The target unified event model used by INDENICA will be derived from base service platforms. After that each of base platforms will need to be tailored to comply with the model and the communication mechanisms used in the monitoring engine.

These unified models will base a monitoring interface, which will be developed in the second half of the project.

	type	description	data
Trailer	TruckCheckIn	checkin of truck at checkpoint	appData
	TruckCheckOut	checkout of truck at checkpoint	appData
	TruckDelayed	A truck reports a delay	appData
	TruckRescheduling	tresheduling due to delays of trucks or delays at dock	appData
Jockey	TaskScheduled	a task was created for a jockey	taskData
	TaskReserved	a jockey reserved a task	taskData
	TaskFinished	a task was finished by a jockey	taskData
	TrailerRelocation	additional event on finished trailer reloc.	taskData
Errors	MisdirectedTrailer	if a trailer finds itself at a wrong dock	trailerId, appData

	scheduling collision	collision in schedule due to delays	appData1, appData2
	NoAppointmentFound	if no appointment could be found for a req. Appointmentdate	appData
	NoFreeWaitingbay	if no waiting bay could be found for a check-in-truck	appData

Table 1: Internal structure of JSON-based event of Yard Management Subsystem

appData	appointmentId, start, end, dock, truckId
taskData	timestamp, taskType, trailerId, jockeyId, origin, destination

Table 2: Inner structure of data elements of Yard Management Subsystem event model

4.5.3 Events in Warehouse Management System

The third system was the warehouse system which contains two subsystems, the warehouse management system and the conveyor control subsystem. Both subsystems add events to message queues.

The **warehouse management system** supports events related to transportation units and events related to orders. Events are sent as text based key value pairs.

Order related events

```
MessageType : OrderCreated | PickingStarted | OrderFinished
OrderId : <alphanumerical identifier>
```

Transport unit related events

```
MessageType : TransportUnitCreated | TransportUnitStored
TransportUnitId : <alphanumerical identifier>
```

The Conveyor control system supports events related to conveyors and events related to stacker cranes:

Conveyor related events:

```
MessageType : ConveyorTransportStarted
TransportUnitId : <alphanumerical identifier>
```

```
MessageType : ConveyorTransportFinished
TransportUnitId : <alphanumerical identifier>
TransportUnitLocation : <alphanumerical identifier>
```

Stacker crane related events

```
MessageType : StackerCraneStartedPickingUp
CraneId : <alphanumerical identifier>
```

```
TransportUnitId : <alphanumerical identifier>
```

```
MessageType : StackerCranePickedUp
CraneId : <alphanumerical identifier>
TransportUnitId : <alphanumerical identifier>
```

```
MessageType : StackerCranePickedUp
CraneId : <alphanumerical identifier>
TransportUnitId : <alphanumerical identifier>
TransportUnitLocation : <alphanumerical identifier>
```

4.6 Runtime Governance Dashboard

Runtime Governance Dashboard is a tool to visualize the usage of domain-specific platforms at runtime. Its purpose is to show which web services are used the most, what the parameters are, how many errors and exceptions are being caught and allowing drill-down analysis for corresponding service call requests and responses.

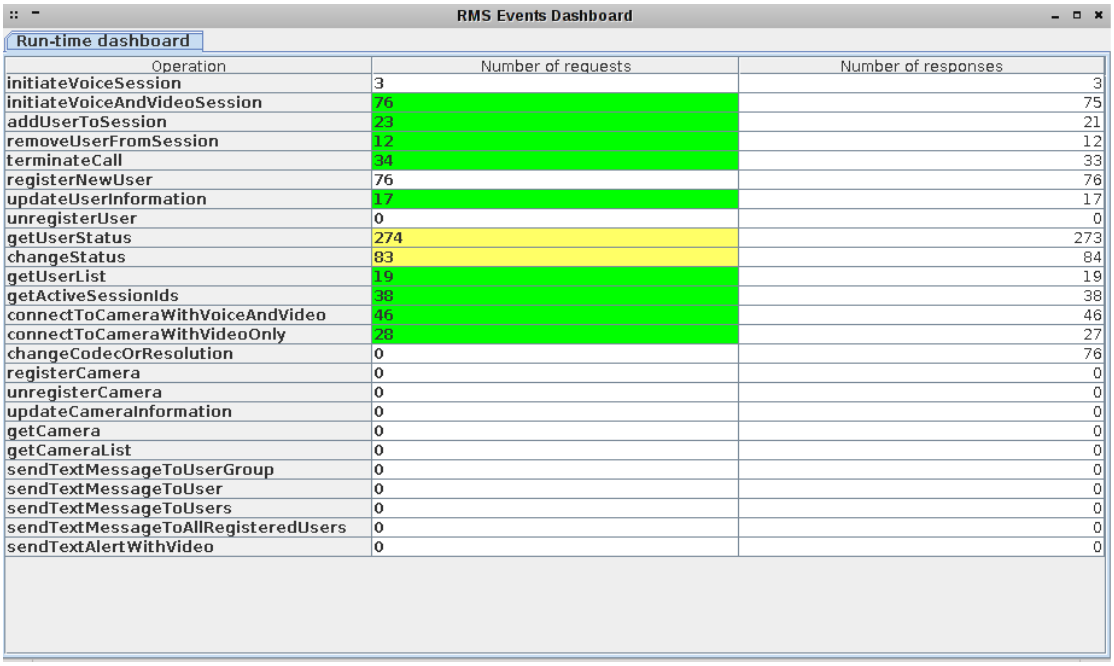
Runtime Governance Dashboard allows using dynamic data models (meta-models) for monitored events, i.e. in case of different observed service platforms it will adjust to the fields of the message received. Also if certain fields that are in the model are being empty, they will not be shown in the dashboard to reduce the number of columns.

The dashboard is based on observing web-service calls (request and responses) thus require events in a certain XML-based structure:

```
<xs:element name="event" type="EventType" />
<xs:complexType name="EventType">
<xs:sequence>
      <xs:element name="eventId" type="xs:long" minOccurs="1"/>
      <xs:element name="eventName" type="xs:string" minOccurs="1"/>
      <xs:choice>
            <xs:element name="request" type="RequestType"/>
            <xs:element name="response" type="ResponseType" />
      </xs:choice>
      <xs:element name="errorMessage" type="ErrorMessageType"
minOccurs="0"/>
</xs:sequence>
</xs:complexType>
```

It is required that domain-specific service platform's services' names are included in eventName parameter and eventId parameter corresponding to the name is also present. Request and Response types are assumed to include additional parameters, which will be displayed in the dashboard.

The main dashboard automatically extracts all possible web service function names from the corresponding XML schema and provides the number of requests and responses caught during the runtime execution of the environment. The main dashboard is depicted in the Figure 14.



Operation	Number of requests	Number of responses
initiateVoiceSession	3	3
initiateVoiceAndVideoSession	76	75
addUserToSession	23	21
removeUserFromSession	12	12
terminateCall	34	33
registerNewUser	76	76
updateUserInformation	17	17
unregisterUser	0	0
getUserStatus	274	273
changeStatus	83	84
getUserList	19	19
getActiveSessionIds	38	38
connectToCameraWithVoiceAndVideo	46	46
connectToCameraWithVideoOnly	28	27
changeCodecOrResolution	0	76
registerCamera	0	0
unregisterCamera	0	0
updateCameraInformation	0	0
getCamera	0	0
getCameraList	0	0
sendTextMessageToUserGroup	0	0
sendTextMessageToUser	0	0
sendTextMessageToUsers	0	0
sendTextMessageToAllRegisteredUsers	0	0
sendTextAlertWithVideo	0	0

Figure 14: Main view of Runtime governance dashboard

If there are any events already present, the dashboard allows opening the detailed view of a single type of web service function after double-clicking the cell with the number. After that the detailed view is available, which shows requests and corresponding responses for a single type of service. The detailed view is not showing all possible parameters that were provided in the XML schema for the operation, but shows only the ones, that were used in the events caught.

The detailed view of the dashboard is depicted in the

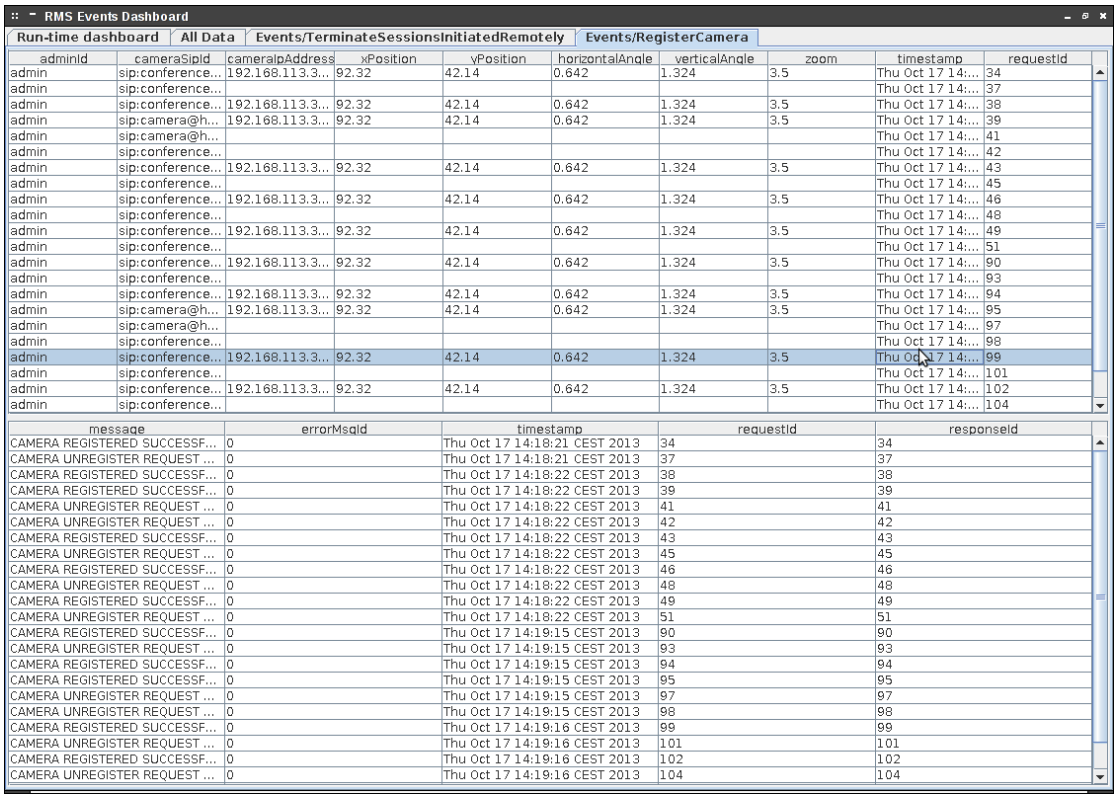


Figure 15: Detailed view of Runtime governance dashboard

The Runtime governance dashboard has been developed in Java using Swing.

4.6.1 Installation and configuration

Prerequisites

- Maven (version 2 or 3)
- Java (version 6)
- Circa 100 MB free disk space
- Installed, configured and running RMS Platform

Installation of monitoring dashboard

Complete installation and configuration process can be performed by invoking in the main adaptive monitoring directory:

```
mvn clean install
```

Starting the dashboard

```
java -jar dashboard-indenica/adaptive-monitoring-dashboard/target/adaptive-monitoring-dashboard.jar
```

4.7 System Supervision Dashboard

The system supervision dashboard has been developed complementary to the Runtime governance dashboard to provide the general health state of the environment on which the base platforms are running on.

The data is gathered from the monitoring engine and stored in the database (hsqldb), which is then used to push merged data to the JSP and allows view manipulation in JavaScript.

Rabbit MQ Message broker

XML messages are sent to the RabbitMQ, which is queuing system based message broker. Queuing is very comfortable for monitoring engine because messages do not overload its server and can be popped in every moment of time. Applied publish/subscribe RabbitMQ system is shown in Figure 16. It is composed of producer (base platform), exchange, queue (both placed on a Rabbit MQ server) and consumer (monitoring engine). Producer and consumer can publish/subscribe to specified topic, default one is *event*.

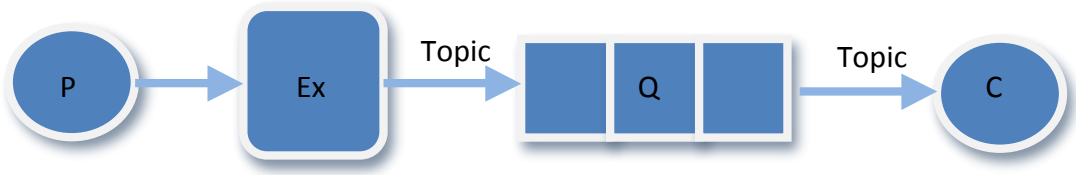


Figure 16: Rabbit MQ publish/subscribe system - producer, exchange, queue and consumer

Displaying monitoring data using web user interface

Monitoring messages are sent to the Rabbit MQ can be also retrieved and presented in a web browser, for which Google Chart is used. Examples of collected and processed machine monitoring data are presented in Figure 17.

Parameter	Value
processorUsage [%]	4.1
processorIdle [%]	95.7
freeVirtualMemory [bytes]	109759527
totalVirtualMemory [bytes]	126550016
maximumVirtualMemory [bytes]	1877213184
Disc 0: freeSpace [bytes]	0
Disc 0: totalSpace [bytes]	0
Disc 0: usableSpace [bytes]	0
Disc 1: freeSpace [bytes]	103591343915
Disc 1: totalSpace [bytes]	127991935795
Disc 1: usableSpace [bytes]	103591343915
Disc 2: freeSpace [bytes]	0
Disc 2: totalSpace [bytes]	0
Disc 2: usableSpace [bytes]	0

Figure 17: Mean values of machine monitoring data

Data that is presented in Figure 17 can be also shown in a more readable form, which is depicted in Figure 18. The decision of how the data should be presented is left to the system user.

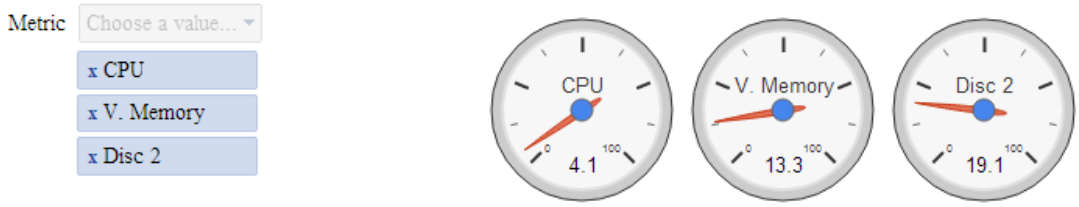


Figure 18: Dashboards with the most important data

Tracking of the current system changes is possible as well. The example of processor usage is shown in Figure 19. Various lengths of time filters are supported available to manual manipulation by the user.

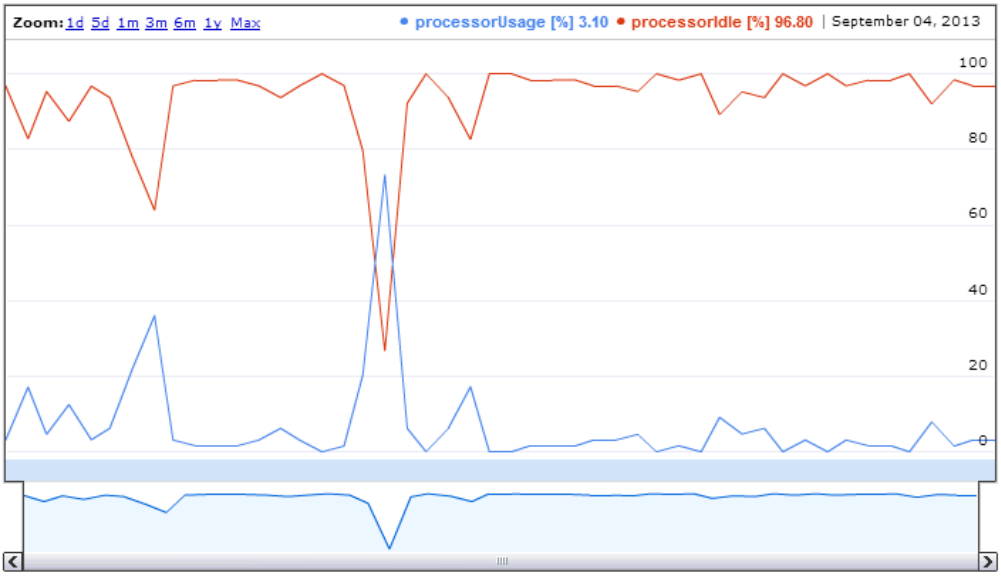


Figure 19: Timeline with usage of processor

System supervision dashboard has been developed in Java (JSP) and JavaScript with JBOSS as an application container.

4.7.1 Installation and configuration

To install the System Supervision Dashboard it is required to have Java (version at least 6), Apache Maven (version at least 2.2.1), RabbitMQ and about 100 MB of available disc space. In order to test the tool it is essential to have one of the domain-specific platforms installed and running – the RMS platform is the reference platform which was used during the development process.

Complete installation and configuration process (without installation of RabbitMQ - since it is not considered as a part of our System Supervision Dashboard pack and should be installed separately) can be performed just by invoking in the main System Supervision Dashboard directory:

```
mvn clean jboss-as:deploy -Djboss-as.username=admin
-Djboss-as.password=password
-Djboss.server.ip=jboss.ip.address
```

5 Tool suite for adaptation of Virtual Service Platforms

5.1 Adaptation Engines

In concert with the monitoring infrastructure, INDENICA provides an extensible, layered platform adaptation framework, geared towards efficient and effective control of service platforms, minimizing communication overhead while maintaining high flexibility and allowing for complex management structures.

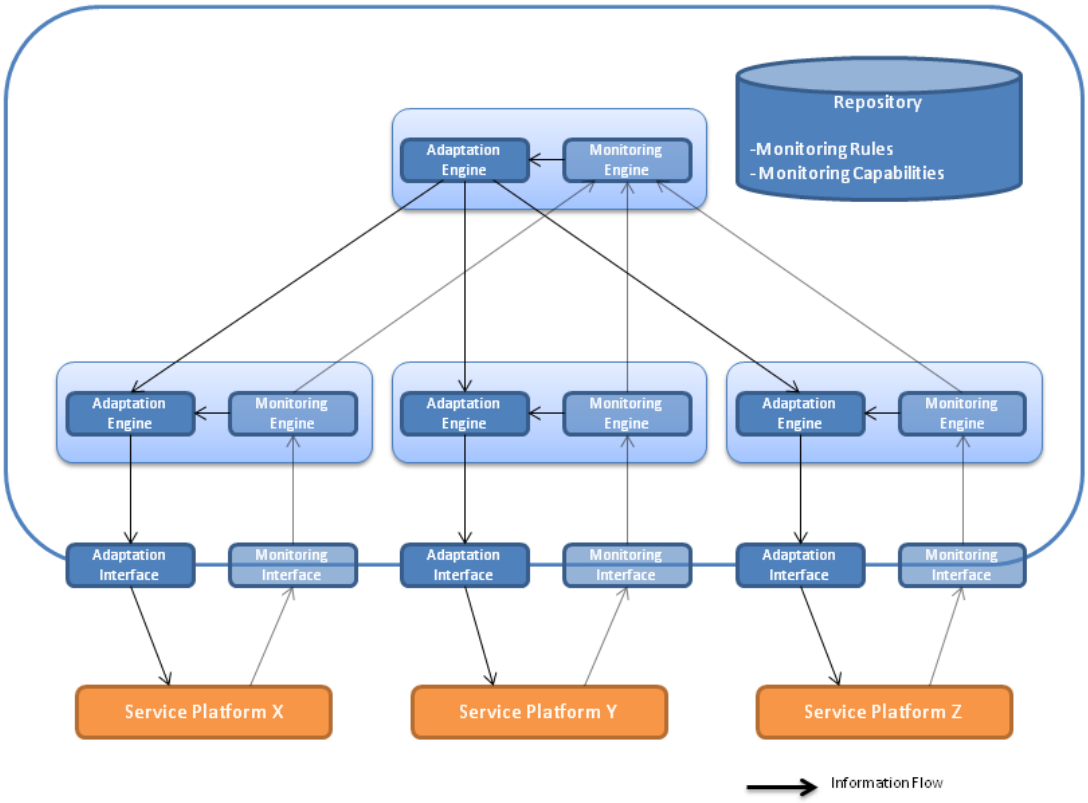


Figure 20: Runtime Infrastructure Architecture: Adaptation Overview

The adaptation infrastructure is instantiated according to configuration directives deployed to the repository (cf. Section 2.2) by creating all necessary adaptation engines and establishing connections to the integrated service platforms. As mentioned in Section 4.4, connections between adaptation engines and according monitoring engines are set up.

5.2 Adaptive Monitoring Interface

The Adaptive Monitoring Interface was designed and implemented during the third year of the project and is used to gather common type of information from underlying service platforms with the ability to adapt how monitored information is processed and passed for further analysis. Adaptive Monitoring Interface was developed and demonstrated together with Runtime Governance Dashboard described in section 4.5 of this document. The main concept is to provide adaptive services to steer how certain types of events are forwarded to appropriate user's

queues or dashboards. The architecture of the Adaptive Monitoring Interface is depicted in the Figure 21.

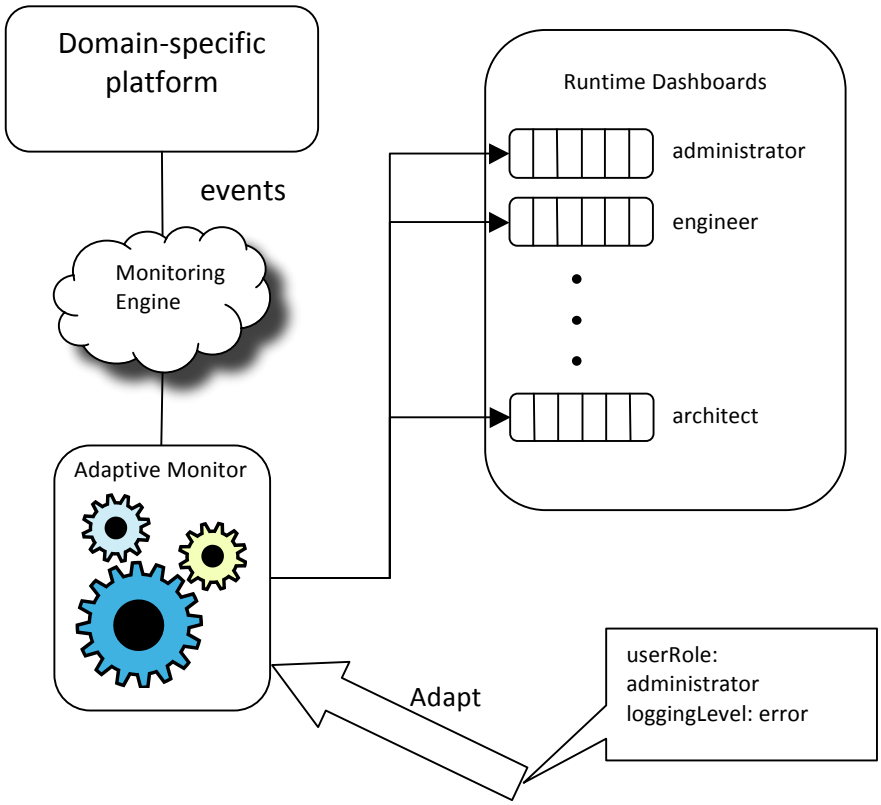


Figure 21: Adaptive Monitoring Interface Architecture

The approach requires domain-specific platforms to provide three types of information about the events occurred as XML-formatted messages that are sent to the monitoring engine. The types of events forwarded to the monitoring engine included:

- Service requests
- Service responses
- Updates on system parameters

Subset of messages generated by the domain-specific platforms need to be forwarded to the governance dashboard so they can be spotted by system administrators, accountants, business logic specialists or any other personnel inside the organization. Adaptive Monitoring Interface is realized as a mapping between characteristics of an event (event logging level) and characteristics of the users (event role name). This map can be pre-defined, and at the runtime it can be adapted using web services listed in the table 3.

Web service name	Parameters	Description
createRoleWithLoggingLevel	roleName loggingLevel	Maps user role with given event logging level
removeRole	roleName	Destroys map entries for given

		user role
getRoleNames	----	Gets names of all used user roles
changeRoleLoggingLevel	roleName loggingLevel	Changes logging level for a given user role

Table 3: List of adaptation services for Adaptive Monitoring Interface

The Adaptive Monitoring Interface has been implemented in Java as a component for JBOSS version 7.

5.2.1 Installation and configuration

To install the Adaptive Monitoring Interface it is required to have Java (version at least 6), Apache Maven (version at least 2.2.1), Apache ActiveMQ and about 100 MB of available disc space. In order to test the tool it is essential to have one of the domain-specific platforms installed and running – the RMS platform is the one that was used for evaluation of the tool.

Complete installation and configuration process (without installation of activeMQ - since it is not considered as a part of our adaptive monitoring pack and should be installed separately) can be performed just by invoking in the main adaptive monitoring directory:

```
mvn clean jboss-as:deploy -Djboss-as.username=admin
-Djboss-as.password=password
-Djboss.server.ip=jboss.ip.address
```

5.3 Adaptation of base platforms

The adaptation is managed by decentralized adaptation engine. With regards to the adaptation interface we have decided to follow similar path as we do with the monitoring interfaces, which is the unification of the models and implementation of several communication mechanisms, i.e. HTTP-Service Calls, direct messaging via queues, SOAP, etc.

In the Warehouse Management System the adaptation service is provided as SOAP-based Web Service. The service is able to launch two different types of storage strategies – FastGoodsIn and OptimizedStorage depending on the situation in the warehouse.

In the Yard Management System the adaptations are more complex and include providing an ‘expected result’ as a feedback, which in the future will help us to create self-adapted sub-system, which does not only fire some adaptation actions, but also reacts, as the adaptation does not result with expected outcome.

The adaptation strategy in Yard Management System has been depicted in Table 3.

	Monitored Subject	Rules	adaptationAction	Expected Result
Occupancy rate of	Docks			
	Parking areas			
	Whole yard	if dock.load == 100% and waitingBay.load>= 80%, notification to operator		decrease of occupancy
Trucks	Waiting time	if a truck waitingtime is over 4h, reschedule it with higher priority over newer trucks	post on */adpt/prioritizeTruck* with **truckId=x**	this and other trucks will be rescheduled
	Time on yard	if timeonyard==high, waitingtime==high and dock.load== medium, change algorithm to dynReschedulingAlgo	post on */adpt/algo* with **scheduling=dyn|static**	major and dynamic reschedulings reducing waiting time of trucks
Jockeys	Idle time	if avg idle time null/high, decrease/increase jockey count	post on /adpt/jockey with **action=dec|inc**	idle time is improved (other jockeys are intended to work in the warehouse instead)
Error rate	scheduling collision	if a collision takes place, resolve it with dynamic algorithm	post on */adpt/reschedule* with **appId=x&scheduling=dyn**	
Other		if a dock is disabled, reschedule all appoints to different docks	post on */adpt/dock/{id}* with **state=disabled|enabled**	rescheduling in/excluding this dock

Table 3: Adaptation strategy for Yard Management System

The adaptation of the Remote Maintenance System is based SOAP-based web service interface. There are multiple functions available to change the strategy how the RMS is working at runtime as well as in deployment time. The complete list of

these services is attached in the table 4. For each of this services there are get and set functions which can be easily integrated in more complex adaptation engine, which will be able to “control” the behaviour of multiple domain-specific service platforms at the same time. These services can be triggered with Boolean values where true means allowing and false is disabling certain functionalities.

Adaptation service names	Description
TextMessageHistoryPermission	Store text messages in internal database for further use
DirectVoiceAndVideoSessionInitiationPermission	Allowing users to make direct calls (video and audio) to other users (phone to phone)
RemoteVoiceAndVideoSessionInitiationPermission	Allowing system-initiated calls (video and audio)
DirectVoiceSessionInitiationPermission	Allowing users to make direct calls (audio only) to other users (phone to phone)
RemoteVoiceSessionInitiationPermission	Allowing system-initiated calls (audio only)
DirectVideoSessionInitiationPermission	Allowing users to make direct calls (video only) to other users (phone to phone)
RemoteVideoSessionInitiationPermission	Allowing system-initiated calls (video only)
DirectTextMessageSessionPermission	Allowing text messaging between users
RemoteTextMessageSessionPermission	Allowing system-initiated text messaging
MachineMonitoringPermission	Allowing monitoring of the RMS

Table 4: Adaptation services for Remote Maintenance System

6 Summary

There are multiple tools developed during INDENICA specifically for deployment, monitoring, adaptation and controlling of Virtual Service Platforms. Tool suites have been evaluated and used in the context of WP5 creating an important value of the use cases and demonstration scenarios.

Tools have been used during dissemination and exploitation activities and were identified by industrial partners as potential components for their usage in the products and services in the future.

A Appendix 1: EcoWare Usage Guide

A.1 Installation and Setup

Requirements

In order to use the ECoWare framework you need this system configuration:

- as ECoWare is implemented in Java, you must have the Java Virtual Machine (1.7 version) installed on your system. In particular, you want to have the Java-Development-Kit (JDK
- as ECoWare uses RabbitMQ as a messaging broker to manage publish/subscribe processes, you must have RabbitMQ on your system.
- ECoWare uses “Esper” and “EPL - Event Processing Language” to manage complex event processing, so the knowledge of these concepts is necessary for a correct use of ECoWare).

ECoWare is cross-platform, so you can run and use it on “Microsoft Windows”, “Linux” and “Mac OS X”.

Installation

This is a brief guide on how to install and use the ECoWare framework. ECoWare is an open-source software, so you can import it in your Eclipse workspace and work with ECoWare’s source-code. You can use ECoWare like it is or contribute new changes.

Technological aspects

For a correct working flow, we point out some important things:

- we recommend that using Eclipse IDE as development environment. Be sure to download the correct 32-bit or 64-bit version, depending on your machine.
- we suggest using JDK 7.

Before getting started with development, you'll need to have an Eclipse git plug-in installed. We suggest using EGit, as it's a very popular git plug-in.

N.B. To install new plug-in in Eclipse, use the *Help → Install New Software...* menu.

Import ECoWare source-code into Eclipse

To import ECoWare source-code into Eclipse you have to follow this steps:

- Select *File → Import...*
- Browse *Git → Projects from Git*, and then click *Next*
- Select *URI* an then click *Next*
- Now in the *URI* field type this URI: "https://github.com/samguinea/ecoware.git". If everything goes right, the

Host and *Repository path* fields should populate automatically. Check that the selected *Protocol* is *https*, and then click *Next*

- In the *Branch Selection* dialogue, tick *master* branch and then click *Next*
- Now, in the *Local Destination* dialogue, select the directory to store your local repository or keep the suggested one if it is right for you. Don't change the other fields and then click *Next*
- Wait for the repository to be cloned (don't worry if this may take a couple of minutes, it's normal)
- Then check that in the showed dialogue the *Import existing projects* option is selected and then click *Next*
- Finally, in the last dialogue (*Import Projects*) check that the local repository that you have created in the step 6 is selected (also Search for nested projects should be selected) and then click *Finish*.

Wait for the end of the final importing operations and the project will be imported in Eclipse. So now you are ready to work with it.

EcoWare Access Manager

The “ecowareaccessmanager” is a module that provides a helper interface between high level aspects (like ECoWare process) and low level aspects (like usage of the RabbitMQ bus for send/receive messages). In other words, it makes the usage of the bus for sending and receiving data transparent. This is achieved by means of two classes, “ECoWareMessageReceiver”, that can be used for receive messages from the bus, and “ECoWareMessageSender” that can be used to send messages to the bus.

As said, the “ECoWareMessageSender” class let you easily send messages to the bus, without facing of low level aspects related to the bus management such as creation, configuration and direct use of the bus. In fact, after the creation of an “ECoWareMessageSender” object you can send a message to the bus using very few but intuitive methods.

First of all, you have to create a new “ECoWareMessageSender” object:

ECoWareMessageSender sender = new ECoWareMessageSender(hostname, publicationID);

where *hostname* is the host on which can be found the messaging bus (that is RabbitMQ server) and *publicationID* is the publication ID of the messages that will be send.

Now, to send a message, all you have to do is:

- start (open) the connection to the bus:
  - s*ender.startConnection();*
- send a message:
  - *sender.send(message, event_type, event_id);*
- when you have sent all messages, stop (close) the connection to the bus:

- *sender.stopConnection();*

Now, let's have a closer look at the send method. As can be seen, that method requires three parameters: the “message” that needs to be sent, the “event type” to which the message is related to, and the “event id” (for future use, “-1” if not used).

The “message” parameter is the body of an ECoWare event, and this is modeled as a “HashMap”, so its content is a set of pairs. Each event has its specific map that is required (and you must respect) to make possible their correct usage during analysis processes.

For example, a “StartTime” event (event type = START_TIME) or a “EndTime” event (event type = END_TIME) is defined by this map:

<"key", String.class>
<"value", long.class>

that is, the name of the first element of the map is “key” and its type is “String”, while the name of the second element of the map is “value” and its type is “long”.

So, for example, if you want send a “StartTime” event and its related data (that is, the message body), you can do it by using the following Java statements:

```
HashMap<String, Object> mapMessage = new HashMap<String, Object>();
mapMsg.put("key", "105");
mapMsg.put("value", 1.0);
sender.send(mapMessage, ECoWareEventType.START_TIME, -1);
```

A.2 Tutorial

In this section we provide a short tutorial to help you to learn how use “ECoWare" to create your analysis applications using "ECoWare KPIs" as processing units. First of all, however, we introduce general aspect related to the creation and the usage of "ECoWare" objects to create your applications.

In generale, an ECoWare Process is a set of one or more "KPI objects", such as "Calculators", "Filters" and "Aggregators", that can be used to realize your analysis process. To create a process, we have to create a "launcher" that create and launch the execution of the "KPI objects" (one or more). So, we create a main class with an "ECoWareProcessor" object that is an object that, using a XML file as processor configuration, can create and launch an ECoWare process:

```
publicclassProcessorLauncher{
        private ECoWareProcessor processor;
        .....
        .....
}
```

Now we can define a static "main" method for our "ProcessorLauncher" class which instantiates a new ECoWareProcessor object, configures it passing to its constructor

a XML configuration file (that, as we will see, contains all the needed information to create our calculator/s), and then starts the so created ECoWareProcessor object:

```
staticvoidmain(args[]){
   processor =new ECoWareProcessor("configuration_file.xml");
   processor.start();
}
```

where configuration_file.xml is a XML file with the specification of the ECoWare process, which as said above, in general, can either be a single KPI or a composition of more KPIs.

The general XML structure for defining a KPI calculator is the following:

```
<Calculator>
<name>CALCULATOR_NAME</name>
<subscriptID>SUBSCRIPTION_ID_1</subscriptID>
<subscriptID>SUBSCRIPTION_ID_2</subscriptID>
      .....
<subscriptID>SUBSCRIPTION_ID_N</subscriptID>
<publicationID>PUBLICATION_ID</publicationID>
<computation>
<intervalUnit>INTERVAL_UNIT</intervalUnit>
<intervalValue>INTERVAL_VALUE</intervalValue>
<outputUnit>OUTPUT_UNIT</outputUnit>
<outputValue>OUTPUT_VALUE</outputValue>
</computation>
</Calculator>
```

where:

- CALCULATOR_NAME is the name of the KPI calculator we want to use.
- from SUBSCRIPTION_ID_1 to SUBSCRIPTION_ID_N are the subscriptions to which the calculator wants register for. An important thing to point out is that there must be at least one subscription.
- PUBLICATION_ID is the calculator publication ID (to “sign” messages that it send on the bus).
- INTERVAL_UNIT and INTERVAL_VALUE refer to the parameters for the inspector window used in the EPL (Esper) query. Valid values for “INTERVAL_UNIT” are those that are admitted in the EPL language syntax (eg. “seconds”, “minutes”, etc.), while valid values for “INTERVAL_VALUE” are numbers.
- OUTPUT_UNIT and OUTPUT_VALUE refer to the parameters for the “OUTPUT” clause of the EPL query. Admitted values are the same as those admitted for “INTERVAL_UNIT” and “INTERVAL_VALUE”.

The general XML structure for defining a KPI filter is the following:

```
<Filter>
<name>FILTER_NAME</name>
<eventName>KPI_TO_FILTER</eventName>
```

<attributeName>KPI_ATTRIBUTE_TO_FILTER</attributeName>
<subscriptID>SUBSCRIPTION_ID(=KPI_PUBLICATION_ID)</subscriptID>
<publicationID>FILTER_PUBLICATION_ID</publicationID>
<cutoff>THRESHOLD</cutoff>
</Filter>
where:

- FILTER_NAME is the name of the KPI filter. Possible names are “HPFilter” (for a “High Pass” filter) and “LPFilter” (for a “Low Pass” filter).
- KPI_TO_FILTER is the name of the KPI to which the filter must be applied.
- KPI_ATTRIBUTE_TO_FILTER is the KPI attribute to filter.
- SUBSCRIPTION_ID is the reference to the publication ID of the KPI to filter.
- FILTER_PUBLICATION_ID is the filter publication ID (to “sign” messages that it send on the bus).
- THRESHOLD is the threshold limit used to determine the accepted values (and, by the way, the rejected values).

Generally speaking, in a XML configuration file can be defined one or more KPIs, each of them encapsulated in a specific tag (eg. Calculator for a KPI calculator, Filter for a KPI filter, etc.). In addition, as first tag, must be present the host name where can be found the messaging bus (that is the host on which RabbitMQ is running). This can be specified using the busHostName tag. Finally all of that must be enclosed in the ecoware tag.

Example of an Average Response Time

In this tutorial we will see how to create an Average Response Time calculator in ECoWare using the ECoWare Processor class. An "Average Response Time" calculator is a processor that compute the average response time related to one or more specific services (or tasks) to which it subscribes.

First of all, we have to create the "launcher" that creates and launches the execution of the calculator. So, we create a main class (eg. “AvgRTLauncher.java”) with an "ECoWareProcessor" object (that, as we know, is an object that, using a XML file as processor configuration, can create and launch an ECoWare process):

```
publicclassAvgRTLauncher{
        private ECoWareProcess process;
        .....
        .....
}
```

Now we can define a static "main" method for our "AvgRTLauncher" class which instantiates a new ECoWareProcessor object, configures it passing to its constructor a XML configuration file (that, as we will see, contains all the needed information to create an average response time calculator), and then starts the so created ECoWareProcessor object:

```
staticvoidmain(args[]){
```

```
   processor =new ECoWareProcessor("AvgResponseTime.xml");
   processor.start();
}
```

where AvgResponseTime.xml is a XML file with the specifications of the ECoWare process.

For this tutorial we want a specific KPI calculator, that is an "Average Response Time" calculator, which has “AvgRT_Browser” as publication ID, “BrowserInfo” as its unique subscription ID, and which process data within last 20 seconds and outputs result data every 5 seconds.

So the complete XML file configuration (“AvgResponseTime.XML”) will be like this:

```
<ecoware>
<busHostName>localhost</busHostName>
<Calculator>
<name>AvgRT</name>
<subscriptID>ComponentInfo</subscriptID>
<publicationID>AvgRT_Component</publicationID>
<computation>
<intervalUnit>seconds</intervalUnit>
<intervalValue>20</intervalValue>
<outputUnit>seconds</outputUnit>
<outputValue>5</outputValue>
</computation>
</Calculator>
</ecoware>
```

In this way we have created a processor that calculate/evaluate the average response time relative to a process that sends data with the “ComponentInfo” Id.

To test this processor, we have also implement a simple sender, that is the process that send data (on the bus) with the “ComponentInfo” Id.

To simplify things we can say that an average response time calculator evaluates the average of a set of “EndTime - StartTime” differences, where the set refers to events that fall in the window defined in the “AvgRT” calculator configuration file (as we seen previously). Knowing this, the simplest sender we can produce to test our “AvgRT” calculator is one that sends a sequence of “StartTime” and “EndTime” events. So, our sender can be implemented like this (eg. “SenderTest.java”):

```
HashMap<String, Object> mapMessage = new HashMap<String, Object>();
   ECoWareMessageSender sender;
   sender = new ECoWareMessageSender("localhost", "ComponentInfo");
   sender.startConnection();
   for(int i=0; i<msgs; i++){
      mapMsg.put("key", "105");
      mapMsg.put("value", 1.0);
      sender.send(mapMsg, ECoWareEventType.START_TIME, -1);
      mapMsg.put("key", "105");
      mapMsg.put("value", 3.0);
      sender.send(mapMsg, ECoWareEventType.END_TIME, -1);
      Thread.sleep(10);
```

```
    mapMsg.put("key", "105");
    mapMsg.put("value", 2.0);
    sender.send(mapMsg, ECoWareEventType.START_TIME, -1);
    mapMsg.put("key", "105");
    mapMsg.put("value", 3.5);
    sender.send(mapMsg, ECoWareEventType.END_TIME, -1);
    Thread.sleep(100);
  }
  sender.stopConnection();
```

For the meaning of mapMessage see the official ECoWare documentation. In short, the mapMessage is the message format that ECoWare actors use to "comunicate" with one another.

Adding a Filter

A filter is an object that is applied to an kpi processor to filter its data. Generally speaking, in ECoWare exist two type of filters: a "High Pass" filter and a "Low Pass" filter. A "High Pass" filter is a filter that accepts all values that are above a certain threshold limit and rejects all other values, while a "Low Pass" filter is the opposite because it accepts all values that are under a certain threshold limit and rejects all other values.

In this tutorial we will see how it is possible in ECoWare to create a *High Pass* filter and how to apply it to an *Average Response Time* calculator.

For this tutorial we want two KPIs:

- an "Average Response Time" calculator, which has “AvgRT_Component” as publication ID, “ComponentInfo” as its unique subscription ID, and which process data within last 20 seconds and outputs result data every 5 seconds.
- a "High Pass" filter which filters the "value" attribute of an "AvgRT" calculator; the "AvgRT" calculator sends messages with “AvgRT_Component” as ID. The publication ID of the filter is "AvgRT_Component_HPFilter" and the threshold limit is "2.0".

So the complete XML file configuration (“FilteredAvgRT.XML”) will be like this:

```
<ecoware>
<busHostName>localhost</busHostName>
<Calculator>
<name>AvgRT</name>
<subscriptID>ComponentInfo</subscriptID>
<publicationID>AvgRT_Component</publicationID>
<computation>
<intervalUnit>seconds</intervalUnit>
<intervalValue>20</intervalValue>
<outputUnit>seconds</outputUnit>
<outputValue>5</outputValue>
</computation>
```

```
</Calculator>

<Filter>
<name>HPFilter</name>
<eventName>AvgRT</eventName>
<attributeName>avg</attributeName>
<subscriptID>AvgRT_Component</subscriptID>
<publicationID>AvgRT_Component_HPFilter</publicationID>
<cutoff>2.0</cutoff>
</Filter>
</ecoware>
```

When you run this tutorial, you will see that the filter activates every time the value produced by "AvgRT" is greater than "2.0" (look at the output produced by "FilteredAvgRTLauncher"). After a first run, try to set the "value" key of the second "END_TIME" event to "3.0" and see what change.

B Appendix 2: SPASS-meter Quick Guide

This is a brief introduction into the usage of the SPASS-meter instrumentation framework. As the support for monitoring non-Java SUMs is currently under development, we will discuss here exclusively the installation, the setup and an example for Java SUMs.

B.1 Installation

SPASS-meter is packaged into a set of Java archives (JAR) for Windows and Linux operating systems as they differ in the included native data gatherer library. Depending on the instrumentation mode, the JARs are linked to the SUM in different ways:

- **Static instrumentation:** The `SPASS-meter-ant.jar` contains the instrumentation layer, the static instrumentation tool, and the integration into the build process. Currently, a simple ANT task for build-process integration is provided. For runtime, the `SPASS-meter-static.jar` includes the probe collection layer, the data aggregation layer, the data presentation layer as well as pre-packaged extensions to such as the implementation of the INDENICA monitoring interface for integration with the INDENICA runtime environment. The `SPASS-meter-static.jar` needs to be included into the class path of the SUM.
- **Dynamic or mixed-mode instrumentation:** The `SPASS-meter-ia.jar` contains the instrumentation layer and the Java instrumentation agent. Due to technical reasons, the Java instrumentation agent loads dynamically two further JARs, one for boot time and one for runtime. The `SPASS-meter-boot.jar` contains annotations and interfaces which need to be present at boot time of the SUM in order to resolve dependencies inserted into the SUM or (dynamically) into the Java library. The `SPASS-meter-rt.jar` contains the upper layers for processing notification calls as well as the analysis extensions similar to `SPASS-meter-static.jar` described above.

If the SPASS-meter monitoring scope specification is given in terms of source code annotations, the `SPASS-meter-annotations.jar` needs to be included into the class path at development time.

In order to install SPASS-meter the JARs mentioned above just need to be copied into one directory (lib directory). The specific JARs are specified as JVM parameters, either the java agent parameter or the class path parameter. In the dynamic case, the location of JARs which are loaded at runtime is inferred based on the already specified JARs.

B.2 Setup

For applying SPASS-meter to a SUM, the monitoring framework needs to be configured. The configuration is twofold: a global configuration which determines the basic operation mode and defaults as well as the monitoring scope specification. The global configuration is given as part of the JVM or tool parameters, respectively.

The monitoring scope can either be specified using source code annotations, e.g. in handcrafted or generated code, or as an XML file. The monitoring scope defines the monitoring groups, i.e. the relevant classes (and methods if required) as well as the individual resources to be monitored. Depending on the specified analysis extensions, the results of monitoring may be a summary file, live events etc.

In INDENICA, the XML monitoring scope specification will be generated from the variability model (runtime variabilities) and further input taken from the monitoring requirements or specification (WP2/WP3). As output, monitoring events will be sent over the monitoring interface event mechanism to the INDENICA runtime environment. Required parameters for the event mechanism will be taken from the deployment specification.

B.3 Example

The example below illustrates the generic application of SPASS-meter to Mobicents JAIN SLEE [MJSLEE]. As the Mobicents source code shall not be modified for monitoring its services, the following XML monitoring scope specification defines that

- Monitoring starts with the startup of the Mobicents SLEE container
- All resources supported by SPASS-meter except for memory usage (`memAccounting` mode `CREATION`) are accounted for all dynamically started Mobicent services (implementing `javax.slee.Sbb`)
- Monitoring stops at the end of the Mobicents SLEE container.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://sse.uni-hildesheim.de/instrumentation"
memAccounting="CREATION">

<namespace name="" typeOf="javax.slee.Sbb"/>

<namespace name="org">
<namespace name="mobicents">
<namespace name="slee">
<namespace name="container">
<module name="SleeContainer">
<behavior signature="initSlee()">
<startSystem/>
</behavior>
<behavior signature="shutdownSlee()">
<endSystem/>
</behavior>
</module>
```

```
</namespace>
</namespace>
</namespace>
</namespace>
</configuration>
```

For dynamic instrumentation the Java agent is given as a JVM parameter. Below, the Java agent is packaged in `spass-meter-ia.jar`, the monitoring scope is given in the `mobicents.xml` file and regular update events are issued each 2000 ms.

```
   -javaagent:instrumenter\spass-meter-
ia.jar=xmlconfig=mobicents.xml,
   outInterval=2000
```

The static instrumentation of Mobicents is illustrated below in terms of a simple ANT task (fragment). The static instrumentation tool takes two directories with JAR files as input, produces instrumented jar files as output and considers the specified options currently specified in the same format as for dynamic instrumentation shown above.

```
   <spassInstrumenter
      classpathref="classpath"
      in=”lib/*.jar, server/default/deploy/mobicents-slee/lib/*.jar”
      out="instrumented"
      params="xmlconfig=mobicents.xml,outInterval=2000” />
```

C Appendix 3: Indenica Runtime Platform Demonstrator

This section presents a guide on the steps necessary to set up the INDENICA runtime platform in Eclipse, as well as a standalone application.

C.1 Initial Eclipse Project Setup

Getting the project up and running in Eclipse for the first time involves the following steps:

- Don't worry about errors due to missing build.properties or due to an unbound classpath variable (e.g. M2_REPO), as they should be fixed by maven automatically.
- Set your default installed JRE to your JDK (required for maven)
- Install a maven integration, e.g. m2eclipse
- Run maven on the project in order to populate the maven repository
- Refresh your project. Additionally, it might be needed to
  - create an empty `build.properties` file
  - modify your `eclipse.ini` to point your -vm to the JDK-VM to get rid of a tools:jar:1.5.0 error

C.2 Getting Project Dependencies

The runtime platform uses MongoDB and RabbitMQ for data storage and messaging. These components must be running in order for the platform to work.

You can either install these requirements on your development machines or use a prepared virtual instance.

C.2.1 Manual Installation of Dependencies

For installation of MongoDB and RabbitMQ, please refer to the respective project web sites. After installing and starting the components, you will need to adjust the platform environment configuration to point to your local instances.

NOTE: At the moment, this information is scattered throughout several files. Future refactoring and cleanup will improve this situation. Using the preconfigured virtual machine described below should be an easier way to get the necessary dependencies running.

Currently, environment configuration is stored in the following files in src/main/resources/

- src/main/resources/META-INF/sca-deployables/*.composite: The *.composite files contain information about how to instantiate runtime instances. The RepositoryComponent section contains a reference to the MongoDB instance. Adjust the dbAddress property accordingly.

- src/main/resources/files2DB/properties/*.properties: The *.properties files contain configuration information for the initial data population module. The eventRepositoryAddr property should point to the RabbitMQ instance. Adjust this property accordingly.

C.2.2 Use a preconfigured Virtual Machine with all Dependencies

The current default configuration assumes that there are active RabbitMQ and MongoDB instances running on host `192.168.56.101`.

These requirements can be easily fulfilled by perusing `vagrant` and a virtual machine provided by TUV. The steps necessary to start the virtual machine are:

- Install vagrant (see http://vagrantup.com/)
  ```
  gem install vagrant
  ```
- Add the `indenica_support_dependencies` base box:
  ```
  vagrant box add indenica_support_dependencies \
      http://db.tt/qQcjgPzp
  ```
- Initialize the machine:
  ```
  mkdirsupport_components&& cd support_components
  vagrant initindenica_support_dependencies
  ```

You only have to perform these steps once. After the initial setup, the configured virtual machine can be started using:

```
vagrant up
```

After a few minutes, the virtual machine should be running, the network settings applied and the environment ready to go.

The virtual machine can be stopped using:

```
vagrant halt
```

The machine can be started again using vagrant up. For further information, please refer to the Vagrant Documentation.

C.3 Starting the Platform

C.3.1 Command Line

When you have the dependencies running, you need to deploy the platform configuration data before running the simulation:

mvnexec:java -Dexec.mainClass="indenica.deployment.utils.Populator"

With all prerequisites deployed, the Warehouse simulation can be started using:

```
mvnexec:java -Dexec.mainClass="indenica.deployment.Launcher" \
             -Dexec.arguments="UseCaseWP4Runtime"
```

This will start the demo you saw at the Munich meeting.

C.3.2 Eclipse

To start the simulation from within Eclipse, run the indenica.deployment.usecase.UseCaseLauncher class as a Java Application.

To stop the runtime instance, hit return in the console.

References

[BH04] W. Binder and J. Hulaas. A Portable CPU-Management Framework for Java.*IEEE Internet Computing*, 8:74–83, September 2004.

[CO04] A. Chawla and A. Orso. A generic instrumentation framework for collecting dynamic information.*SIGSOFT Softw. Eng. Notes*, 29:1–4, September 2004.

[ES12] H. Eichelberger and K. Schmid, Erhebung von Produkt-Laufzeit-Metriken: Ein Vergleich mit dem SPASS-Meter-Werkzeug, in: Büren, G., Dumke, R. R., Ebert. C., Münch, H. (Eds.), Proceedings of the DASMA Metrik Kongress (MetriKon '12), pp. 171–180, (in German).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring, 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '12), pp. 247–248

[SGF 2009] The Open Group: The Open Group SOA Governance Framework; Draft Technical Standard, 2009. www.opengroup.org/projects/soa-governance.

[D1.2.1] INDENICA Deliverable D1.2.1 - Requirements Engineering Framework, Language and Tools for Service Platforms (Interim), 2011-10-31

[D2.3.1] INDENICA Deliverable D2.3.1 - Service Platform Infrastructure Repository Concept & Realization (Interim), 2012-01-31

[D3.1] INDENICA Deliverable D3.1 – View-based Design Time and Runtime Architecture for Tailoring VSPs, 2011-10-18

[D3.2] INDENICA Deliverable D3.2 - Architecture for Role-Based Governance of Virtual Service Platforms, 2012-01-31

[D3.3.2] INDENICA Deliverable D3.3.2 – Tool Suite for Virtual Service Platform Engineering (Final), 2013-09-30

[D5.2] INDENICA Deliverable D5.2 - Report on Concepts for Tailoring and Extending Service Platforms, not yet published

[JI12] JXInsight/OpenCore, 2012, jinspired.com/.

[JSON] Internet Engineering Task Force, RFC 4627, 2006, http://www.ietf.org/rfc/rfc4627.txt

[MJSLEE]Mobicents JAIN SLEE, Red Hat Middleware LLC, 2008, http://www.mobicents.org/slee/intro.html

[OSOA] Open SOA (2007). Service Component Architecture (SCA) Specifications V1.00.http://www.osoa.org

[O08] Oracle, Java Management Extensions (JMX) Technology, 2008, www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

[SPEC08] SPEC Corp., SPECjvm2008, http://www.spec.org/jvm2008/

[WSCoL]L. Baresi, S. Guinea, “Self-supervising BPEL Processes,” IEEE Transactions on Software Engineering

[XML] W3C, Extensible Markup Language, 2008, http://www.w3.org/TR/REC-xml/