Deliverable D4.1.b
Non-Functional Aspects

UPM and Thales

Due date of deliverable: 31/10/2009
Actual submission date: 19/01/2009

This work is licensed under the Creative Commons Attribution 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA. This work is partially funded by EU under the grant of IST-FP7-216446.
## Change History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Status</th>
<th>Author (Partner)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>9 Dec 2009</td>
<td>Draft</td>
<td>UPM and Thales</td>
<td>First Version</td>
</tr>
<tr>
<td>0.2</td>
<td>15 Dec 2009</td>
<td>Draft</td>
<td>UPM</td>
<td>Second version</td>
</tr>
<tr>
<td>0.3</td>
<td>19 Jan 2009</td>
<td>Draft</td>
<td>Thales</td>
<td>Third version</td>
</tr>
<tr>
<td>1.0</td>
<td>19 Jan 2009</td>
<td>Final</td>
<td>UPM</td>
<td>Final version</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

This deliverable compiles the work performed architectural support for the non-functional aspects during the second reporting period. It reports on the patterns contributed to the Enterprise SOA top level pattern. It provides the descriptions of topics that were called in the second open contribution cycle managed by WP4 and a summary of the participation in this call as well its current status. Finally, it provides the results of the evaluations performed by the proofs of concept during stage 1.
# Document Information

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>Acronym</th>
<th>NEXOF-RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP7 – 216446</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full title</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NESSI Open Framework – Reference Architecture</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project URL</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.nexof-ra.eu">http://www.nexof-ra.eu</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EU Project officer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arian Zwegers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>Title</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D4.1.b</td>
<td>Non-Functional Aspects</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Work package Number</th>
<th>Title</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Non-Functional Aspects</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date of delivery</th>
<th>Contractual</th>
<th>Actual</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31/10/2009</td>
<td>31/10/2009</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1.0, dated 19/01/2010</td>
<td></td>
<td>final</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nature</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Report</td>
<td>Demonstrator</td>
<td>Other</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract (for dissemination)</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Keywords</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Internal reviewers</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer1 (Organization)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reviewer2 (Organization)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors (Partner)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricardo Jiménez-Peris (UPM), Francisco Pérez-Sorrosal (UPM), Marta Patiño-Martínez (UPM), Damián Serrano García (UPM), Pascal Bisson (Thales), Phong Cao (Thales), Daniel Gidoin (Thales)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Responsible Author</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricardo Jiménez-Peris</td>
<td>Email</td>
<td><a href="mailto:rjimenez@fi.upm.es">rjimenez@fi.upm.es</a></td>
</tr>
<tr>
<td>UPM</td>
<td>Phone</td>
<td>+34 656 68 29 48</td>
</tr>
</tbody>
</table>
# Table of Contents

**Executive Summary** ........................................................................................................... 3

**Table of Contents** ................................................................................................................ 5

1 **Introduction** ....................................................................................................................... 10
   1.1 Scope of the Deliverable .............................................................................................. 10
   1.2 Related Documents .................................................................................................... 10
   1.3 Organization of this Document .................................................................................. 10

2 **Patterns for Non-Functional Attributes in Enterprise SOA** .................................. 11
   2.1 Introduction ................................................................................................................ 11
   2.2 Monitoring in Enterprise SOA Pattern (UPM) .......................................................... 11
      2.2.1 Problem Description ......................................................................................... 11
      2.2.2 Functional Requirements ............................................................................... 12
      2.2.3 Non-Functional Qualities (Quality Attributes) .............................................. 13
      2.2.4 Assumptions ..................................................................................................... 14
      2.2.5 Solution ............................................................................................................ 15
      2.2.6 Relationships to other patterns ....................................................................... 31
      2.2.7 Relationships to Components Catalogue ....................................................... 32
      2.2.8 Relationships to Standards Catalogue ............................................................ 33
      2.2.9 Application Examples ..................................................................................... 35
      2.2.10 References ...................................................................................................... 36
   2.3 Service Runtime: Multi-Tier Transactional Service Runtime Pattern (UPM) .......... 36
      2.3.1 Problem Description ......................................................................................... 36
      2.3.2 Functional Requirements ............................................................................... 37
      2.3.3 Non-Functional Qualities (Quality Attributes) .............................................. 38
      2.3.4 Assumptions ..................................................................................................... 39
      2.3.5 Solution ............................................................................................................ 40
      2.3.6 Relationships to other patterns ....................................................................... 57
      2.3.7 Relationships to Components Catalogue ....................................................... 57
      2.3.8 Relationships to the Standards Catalogue ....................................................... 58
      2.3.9 Application Examples ..................................................................................... 58
      2.3.10 References ...................................................................................................... 59
   2.4 Security in Enterprise SOA Pattern (Thales) ............................................................... 60
      2.4.1 Problem Description ......................................................................................... 60
      2.4.2 Functional Requirements ............................................................................... 61
      2.4.3 Non-Functional Qualities (Quality Attributes) .............................................. 64
      2.4.4 Assumptions ..................................................................................................... 66
2.5 Trigger Writeset Extraction Pattern (UPM) ......................................................... 96
  2.5.1 Problem Description ........................................................................... 96
  2.5.2 Functional Requirements ................................................................. 97
  2.5.3 Non-Functional Qualities (Quality Attributes) ................................. 97
  2.5.4 Assumptions .................................................................................. 98
  2.5.5 Solution .......................................................................................... 99
  2.5.6 Relationships to other patterns ......................................................... 101
  2.5.7 Relationships to Components Catalogue ......................................... 101
  2.5.8 Relationships to Standards Catalogue ............................................ 102
  2.5.9 Application Examples .................................................................... 102
  2.5.10 References .................................................................................. 102

2.6 Log Mining Writeset Extraction Pattern (UPM) ............................................ 102
  2.6.1 Problem Description ........................................................................ 102
  2.6.2 Functional Requirements ................................................................. 103
  2.6.3 Non-Functional Qualities (Quality Attributes) ................................. 103
  2.6.4 Assumptions .................................................................................. 104
  2.6.5 Solution .......................................................................................... 105
  2.6.6 Relationships to other patterns ......................................................... 107
  2.6.7 Relationships to Components Catalogue ......................................... 107
  2.6.8 Relationships to Standards Catalogue ............................................ 108
  2.6.9 Application Examples .................................................................... 108
  2.6.10 References .................................................................................. 108

2.7 Writeset Extraction Based on Extended DB Interfaces Pattern (UPM) ... 108
  2.7.1 Problem Description ........................................................................ 109
  2.7.2 Functional Requirements ................................................................. 109
  2.7.3 Non-Functional Qualities (Quality Attributes) ................................. 109
  2.7.4 Assumptions .................................................................................. 110
  2.7.5 Solution .......................................................................................... 111
  2.7.6 Relationships to other patterns ......................................................... 113
  2.7.7 Relationships to Components Catalogue ......................................... 114
  2.7.8 Relationships to Standards Catalogue ............................................ 114
  2.7.9 Application Examples .................................................................... 115
  2.7.10 References .................................................................................. 115

2.8 Gray-Box DB Replication Pattern (UPM) .................................................... 115
3.1.3 Gray-Box Database Replication Architectural Pattern in WANs...... 210
3.1.4 Vertical Replication Architectural Pattern For Scalable and Highly Available Stateful Services.................................................. 215
3.2 PoC Results Related to High Availability and Scalability .................. 220

3.2.1 Database Replication Interfaces for Highly Available Stateful Services ......................................................................................... 220
3.2.2 Gray-Box Database Replication Architectural Pattern For Highly Available Stateful Services.................................................. 230
3.2.3 Gray-Box Database Replication Architectural Pattern in WANs...... 235
3.2.4 Vertical Replication Architectural Pattern For Scalable and Highly Available Stateful Services.................................................. 240

3.3 PoC Descriptions Related to Security .................................................. 248
3.3.1 Security PoC .......................................................................................... 248
3.4 PoC Results Related to Security ................................................................. 256
3.4.1 Pattern: Trusted Timestamping Pattern .............................................. 256
3.4.2 Pattern: Non-Repudiation Pattern ...................................................... 267

4 Open Process......................................................................................... 273
4.1 Proposed Topics ................................................................................... 273
4.1.1 Topic 4.5 Multilevel security for SOA. .................................................. 273
4.1.2 Topic 4.6 Dynamic security in SOA .................................................... 275
4.1.3 Topic 4.7 Service Level Agreements (SLAs) and Quality of Service (QoS). .............................................................. 278
4.1.4 Topic 4.8 Federated and Autonomic Management in SOA ............... 279
4.2 Kick off of Call2 ..................................................................................... 280

5 Appendix ............................................................................................... 281
5.1 Contribution on Terms to the Glossary ................................................ 281
5.1.1 General Terms (UPM) ........................................................................... 281
5.1.2 Terms on Availability and Scalability (UPM) .................................. 282
5.1.3 Terms on Security (Thales) ................................................................. 282

Publications ............................................................................................. 295
1 INTRODUCTION

1.1 Scope of the Deliverable

This deliverable compiles all the contributions made around non-functional aspects to vertical WPs. The vertical WPs present the contributions already in a filtered and integrated form. This deliverable helps to understand all the work performed in the context of the non-functional aspects WP despite the status of the contributions (integrated, filtered, not yet integrated) to the vertical WPs. It also provides traceability of the work performed by the horizontal WP and how it has been later integrated into vertical WPs.

1.2 Related Documents

All project deliverables are interrelated. The reader is referred to deliverable D7.5 for the integrated patterns in Enterprise SOA. D8.0 describes the process for the proofs of concepts. D8.1 describes the selected proofs of concepts for stage 1. D8.2 compiles the proofs of concept results from stage 1. The open construction process is described and reported in the deliverable series D5.x.

1.3 Organization of this Document

Section 2 presents all the architectural patterns contributed to the Enterprise SOA top level pattern by WP4. These patterns also include the abstract and concrete components contributed to the corresponding component catalogues. Section 3 compiles the results of the Proofs of Concepts (PoCs) performed by WP4 in stage 1. Section 4 reports on the second open construction cycle. The appendix includes some other contributions such as the terms contributed to the glossary during the last reporting period.
2 PATTERNS FOR NON-FUNCTIONAL ATTRIBUTES IN ENTERPRISE SOA

2.1 Introduction

This section compiles the architectural patterns contributed to WP7 for the Enterprise SOA top level pattern. The process followed to construct this top level pattern is reported in WP7 and not repeated here. In the following figure is depicted the top-level view of the ESOA pattern highlighting in blue the functionalities of the architecture impacted by WP4:

The management, monitoring and security tools have been exclusively provided by WP4, whilst for the runtime component WP4 has been one of the horizontal WPs contributing to it. More concretely, WP4 has contributed to this component the multi-tier service runtime and a set of patterns for increasing the availability and scalability of multi-tier systems. Some patterns aims at increasing the scalability and/or availability of both individual tiers and the multi-tier runtime as a whole. Some other patterns are patterns that help to solve smaller architectural problems and can be used for the bigger scope patterns. For instance, replica discovery patterns are used by different scalability/availability patterns of individual tiers and the whole multi-tier service runtime.

2.2 Monitoring in Enterprise SOA Pattern (UPM)

2.2.1 Problem Description

Enterprise SOA systems are inherently complex. This complexity is due to the great amount of software and hardware components -called computational resources- that conform a service oriented infrastructure. The service-oriented
business applications and services running on top of the infrastructures may also be complex or participate in a more complex structure by consuming other services. In order to deal with this inherent complexity, some form of control is needed. Moreover, these applications and services must guarantee certain quality of service (QoS) to their consumers (clients). Monitoring is mainly used to make management decisions. Thus, in order to control properly all these components and guarantee the required QoS, it is necessary to observe/monitor their state and behaviour.

Therefore, as it has been stated before, in a SOA monitoring is needed at three different levels:

- **Application/Service level** – Monitoring at this level allows to manage the functional software components -called *Business Agents* - that conform the business logic and also make decisions on the underlying infrastructure (e.g. adding new services to the infrastructure). So, this task is necessary for example to guarantee the SLAs between service producers and consumers. It may be also necessary to monitor the static *Software Components* (programs) from which the agents are created (for example, to monitor the creation or deletion of agents) and the QoS of the Service provided by an agent.

- **NEXOF Compliant Infrastructure (NCI) level** – In this level is considered the monitoring of the software components that constitute the NEXOF infrastructure such as designers, ESBs, runtimes, registries etc. These elements are called *Infrastructure Agents* and provide basic services to the service-oriented applications and services running on top of a NCI. Monitoring these components may help in making decisions about what are the required computational resources to guarantee a certain QoS for the services running on top of the NCI.

- **Computational Resource Infrastructure level** – Monitoring at this level allows to manage the hardware and software resources (e.g. hosts, network components, virtual machines etc.) that constitute the substrate on which the NCIs run. Managing Computational Resources is necessary to ensure that the underlying infrastructure offers the required QoS to the services of the NCI.

So, to summarize, it is necessary for an Enterprise SOA to monitor all its components –from the business services to the underlying infrastructure— in order to achieve a correct operation of the whole enterprise system and applications.

**2.2.2 Functional Requirements**

The following use-case diagram (Figure 1) shows the functional requirements related to monitoring that this pattern meets.
Figure 1 Functionalities provided by the Monitoring Tool for ESOA

- **configureMonitoringPolicy**: This functionality allows to configure the monitoring policies and features of the Enterprise SOA infrastructure and business components.

- **getMonitoringData**: This functionality returns the data of the system that needs to be monitored. It allows recovering both, the functionality of the applications and services running on top of a SOA infrastructure and the functionality of the computational resources provided by a SOA infrastructure.

- **subscribeEvent**: It allows an agent to subscribe its interests to different types of events produced in components.

- **notifyEvent**: Eventually, the agents that registered interest in events are notified every time an event occurs.

### 2.2.3 Non-Functional Qualities (Quality Attributes)

The following table summarizes the quality attribute the pattern affects.

<table>
<thead>
<tr>
<th>Availability</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
<tr>
<td>Performance (efficiency)</td>
<td>-</td>
</tr>
</tbody>
</table>
2.2.3.1 Rational

In the following text is described how the design choices of the Monitoring Tool affect the quality attributes identified.

**Maintainability** → The maintenance of the components of the Monitoring Tool of the NEXOF Compliant Infrastructure is eased by decomposing the Monitoring Tool into loosely coupled sub-components. Each sub-component fulfils one functionality (playing a concrete role in the monitoring tasks), so it is possible modify the implementation of a particular functionality without affecting the rest of the components.

**Availability** → By means of the decomposition of the Monitoring Tool the availability of the monitoring service can be also improved. The Monitoring Tool itself can monitor its own components. In this way, a failure on an agent implementing a particular role can be detected. This failure can be notified to the management component of the NEXOF Compliant Infrastructure, so the system can react to the failure and launch another agent providing the same service on a different computational resource.

**Performance** → The monitoring tasks may affect negatively (degrade) the performance of the services provided by an E-SOA. However, the functionality provided by the Monitoring Tool for performing the monitoring tasks has been decomposed in two different approaches in order to select the most appropriate one depending on the requirements of the service consumer. The Pull approach is used when it is necessary to monitor explicitly an element of the system without affecting too much the performance. The Push (publish/subscribe) approach affects negatively the performance because it requires more resources and interactions with the infrastructure. However, it allows reacting to events produced on the elements of the system, what can be used to improve other quality attributes as it has been stated above.

**Adaptability (Modifiability and Portability)** → By separating the different concerns of the monitoring task, the system can be adapted to fulfil new requirements. Also by abstracting the two different approaches of monitoring by means of a well-defined interface, the adaptability of the system is also increased. This interface also allows connecting in cascade several monitoring components to ease the control of the Enterprise SOA infrastructure.

2.2.4 Assumptions

This section describes the context where the proposed pattern is applicable.
The diagram in Figure 2 shows the functionalities required by the Monitoring Tool from other parts of the system before it makes sense to apply the pattern.

The proposed pattern is applicable in a context where the following functionalities are provided:

- **configureMonitoringPolicy**: this functionality is required by the Monitoring Tool in order to configure the monitoring policies of the components that need to be observed.

- **getLoggedData**: this functionality must be provided by the components that are able to be monitored explicitly in order to extract monitoring data. The functionality must return the data specified by the consumer from applications and services running on top of a NCI, the NCI components and the computational resources of the E-SOA infrastructure.

- **subscribeEvent**: this functionality is required by the Monitoring Tool in order to be notified when some events are produced by particular components.

- **notifyEvent**: this functionality is required by the Monitoring Tool in order to notify events to the clients that have subscribed their interest through the subscribeEvent method.

- **isAuthorized**: this functionality is required by Monitoring Tool in order to check if the clients have the rights to access the monitored data from the components.

### 2.2.5 Solution

The Monitoring Tool is the component that allows observing the state (e.g. attributes/features) and behaviour (e.g. events) of the Software Components, Business Agents and Services existing on an Enterprise SOA system. This component also allows monitoring the resources of the infrastructure; in particular it allows getting useful information about the different components of the system, both software (Infrastructure Agents) and hardware (Computational Resources), i.e. the registry, the ESB, the runtime, hosts, communication links...
etc. In order to accomplish its task, the Monitoring Tool gets logged data from the runtime components and agents and from the computational resources that compose the enterprise SOA.

2.2.5.1 Monitoring Alternatives

There are two style alternatives for this pattern in order to perform the monitoring task on a component depending on the request originator (either the originator is a part of the system acting as a client or as a server): Pull and Push. Both alternatives can be used independently or co-exist in the same system architecture.

Pull Monitoring (Reactive Monitoring)

In the Pull monitoring, the request for monitoring a component is initiated by the actor (client). That is, an actor (or any other requester agent) sends a specific request for getting information about a particular component to the Monitoring Tool. Upon receiving the request, the Monitoring Tool, checks the security policies of the actor with regard to the particular component (e.g. if he/she is allowed to get information from the component), and finally delegates the request to the concrete Enterprise SOA Infrastructure Agent (e.g. Service Runtime, GUI Runtime etc.) in order to retrieve the logged data for the desired component.

For example, with this alternative, when the system administrator considers that a service component must be monitored, he/she will perform a request directly to the Monitoring Tool asking for the information of that particular service. The next figure (Figure 3) shows in more detail how the components interact to monitor a service using the Pull approach.

![Figure 3 Monitoring a service (Pull)](image)

Figure 3 Monitoring a service (Pull)

---

1 From that point on, we will refer to computational resources also as components.
In the previous figure, an actor (in this case the system administrator) sends a message to the Monitoring Tool in order to get information from a particular service. Upon receiving the message, the Monitoring Tool checks if the service information can be obtained by the particular actor, in this case the system administrator. Finally, if the authorization is obtained, the Monitoring Tool asks the Service Runtime where the particular service is running for the monitoring information required.

**Push Monitoring (Proactive Monitoring)**

In this alternative, the monitoring information about a component is originated in the component itself. Previously, an actor (e.g. a system administrator) or any other requester agent (e.g. an autonomic management tool) has manifested interest in monitoring a particular component/agent. In this case, the Monitoring Tool provides an operation to register the interest from clients in particular components/agents and manages a data structure that allows to correlate each component/agent with the clients that want to stay informed about the possible events that occur in the component/agent.

The monitoring system is notified when events occur in each component in order to deliver information to the client (agent) that has requested monitoring information in a particular component. Then the Monitoring Tool checks the clients registered to that particular event and sends them a notification message (e.g. implementing the publish/subscribe or observer pattern [GHJV95, BMRS+96]).

As an example of this pattern, when a system administrator considers that a service component must be monitored, he/she will register first that component in the Monitoring Tool using the provided operation by the interface. Eventually, the system administrator will be informed by the Monitoring Tool. The following figure (Figure 4) shows more details of the previous scenario using the Push approach.
First of all, the administrator user (actor) manifests its interest in monitoring a particular event on a particular service to the Monitoring Tool. Then, the Monitoring Tool checks if the actor is authorized to monitor the particular service. If the administrator is authorized to monitor the service, the Monitoring Tool subscribes its interest for the event in the service runtime and also registers in itself the interest of the actor (administrator) in that event, in order to notify him/her when receiving events from the Service Runtime.

Upon an event occurs in a component, the component notifies the service runtime and the Service Runtime notifies the Monitoring Tool. Then, the Monitoring Tool retrieves the actors registered in that particular event. Finally, the Monitoring Tool sends a notification to each registered actor.

2.2.5.2 Components of the Monitoring Tool
The Monitoring Tool can be decomposed in the components shown in Figure 5.
As it is shown in the figure, the operations provided by the interface of the Monitoring Tool are the following:

- `configureMonitoringPolicy(actor: Actor, agent: Agent, p[]): Parameters`

These operations allow an actor of the system to configure a component in order to be monitored. Both Business and Infrastructure Agents can be configured in order to be monitored. Computational Resources are supposed to be wrapped by an agent, so they can also be monitored. So, the agent parameter represents Software Components, Business and Infrastructure Agents, Services or Computational Resources. The parameters allow to set up different policies, metrics and parameters that will be monitored in the running agent providing a service or in the still non-instantiated component (program). For example, for agents, it can be specified the style in gathering data (push or pull), a time frame to gather statistics from the agent or the concrete QoS attributes or events to monitor and provided by the service. From a software component, it is possible to monitor the creation/deletion of agents from software components.
- **getMonitoringData***(actor: Actor, agent: Agent, p[]): Parameters)*  
An actor invokes this operation in order to gather information for a particular component (pull style). Depending on the component (represented by the agent parameter) that is going to be monitored (e.g. either is an agent wrapping a Computational Resource, an Infrastructure or a Business Agent or a Service), the information can be obtained from the business agent itself, from the infrastructure agents (e.g. the registry, the runtime, the ESB etc.) or from the agents wrapping computational resources. The parameters allow to specify the attributes and events to monitor (e.g. response time, throughput, specific properties of a resource such as CPU usage) and other options (e.g. a time frame for gathering the monitoring data).

- **subscribeEvent***(actor: Actor, agent: Agent, e: Event)*  
This functionality allows an actor in the system (e.g. something acting as a client) to subscribe to a type of event on a particular Enterprise SOA component (the agent parameter represents either Software Components, Business and Infrastructure Agents, Services or Computational Resources) in order to be notified every time an event of the type specified occurs.

- **notifyEvent***(agent: Agent, e: Event)*  
This functionality allows the different components of the Enterprise SOA (the agent parameter represents either Software Components, Business and Infrastructure Agents, Services or Computational Resources) to notify the Monitoring Tool about the events that have triggered.

The interface provided by the Monitoring Tool allows to connect several Monitoring Tools in cascade in a similar way as in the EGA Reference Model [EGA]. Each Monitoring Tool can be configured to monitor a specific part of the system and connected to other Monitoring Tools creating a hierarchy.

The required operations by the Monitoring Tool are:

- **configureMonitoringPolicy***(actor: Actor, agent: Agent, p[]): Parameters)*  
This operation must be provided by all the Enterprise SOA components (Agents, Software Components, Services or Computational Resources) that need to be monitored in order to configure their monitoring policies. The agent parameter allows to specify the concrete component to monitor if the operation is called in a container (e.g. in the Service Runtime Infrastructure Agent). The parameters allow to set up different policies—in form of metrics, quality of service parameters etc.— and events to be monitored in the component. For example, it can be specified the style in gathering data (push or pull), a time frame to gather statistics from the component or the concrete QoS attributes to retrieve.
− getLoggedData(agent: Agent, p[]: Parameters)

  This operation must be provided by all the Enterprise SOA components (Agents, Software Components, Services or Computational Resources) that need to be monitored. This operation allows to collecting monitoring data from those components. The agent parameter allows to specify the concrete component to monitor if the operation is called in a container (e.g. in the Service Runtime Infrastructure Agent). The parameters allow gathering data only from concrete attributes of the component or set up specific events and options for gathering data.

− isAuthorized(actor: Actor, agent: Agent)

  The Monitoring Tool requires this operation from the component that manages the security of the Enterprise SOA system in order to check if the actor has the rights to access the information provided by the component.

− subscribeEvent(actor: Actor, agent: Agent, e: Event)

  The Monitoring Tool registers its interest in monitoring a particular component (Agents, Software Components, Services or Computational Resources) by means of this operation. This operation may be required by any other actor in order to be notified about the events that may occur on the particular component that implements it. Upon receiving events from the component, the Monitoring Tool will notify the actors subscribed to that event using the notifyEvent operation.

− notifyEvent(agent: Agent, e: Event)

  The Monitoring Tool requires this operation in order to notify to the actors that have subscribed their interest though the subscribeEvent method (that is, Agents, Software Components, Services or Computational Resources) about the events triggered by the different agents of the system.

The components in which the Monitoring Tool is decomposed are the following (See Figure 5):

**Collector**

Its main functionality is to collect the required monitoring data. It also acts as the component that isolates the other components of the Monitoring Tool from the particular interface style (push or pull) used to collect monitoring data from the monitored components of the Enterprise SOA (either Software Components, Business and Infrastructure Agents, Services or Computational Resources).

The Collector component provides two interfaces. The first one is called *IP Collector* (IP: Interface-Provided). This is a public interface and provides the operations that allow the components of the Enterprise SOA to notify events to the Monitoring Tool. The second one is protected and is called *IPP Collector*
(IPP: Interface-Provided-Protected). The functionalities provided by this interface are used internally by the other components of the Monitoring Tool.

**IP Collector Provided Operations**

- **notifyEvent(agent: Agent, e: Event)**

  This component receives the events that have occurred in other components in the Enterprise SOA system (the agent parameter represents either Software Components, Business and Infrastructure Agents, Services or Computational Resources). It collects and packs the information received in the event and passes that information to the statistics manager component.

**IPP Collector Provided Operations**

- **subscribeEvent(actor: Actor, agent: Agent, e: Event)**

  This is a protected operation called by the Publish/Subscribe Manager. This component acts as a mediator for the subscribeEvent operation received by the Publish/Subscribe Manager. It delegates the subscription to the specific component of the Enterprise SOA specified in the operation. This operation is required in order to be notified about the events that will occur on the component. The components will notify the collector about the events produced using the notifyEvent operation.

- **getMonitoringData(agent: Agent, p[]: Parameters)**

  This is a protected operation called by the Statistics Manager. This component acts as a mediator for the getMonitoringData operation received by the Statistics Manager. It delegates the gathering of monitoring information to the specific component of the Enterprise SOA specified in the operation using the getLoggedData operation.

**Required Operations**

The Collector requires the following operations from other components of the Enterprise SOA:

- **getLoggedData(agent: Agent, p[]: Parameters)**

  This operation is described in the required operations of the Monitoring Tool.

- **subscribeEvent(actor: Actor, agent: Agent, e: Event)**

  This operation is described in the required operations of the Monitoring Tool.

**Statistics Manager**

The functionality of this component is twofold. On the one hand, this component allows configuring the components monitored by the Monitoring Tool. On the other hand, it delivers the monitoring information received from the Collector (by
means of the `putMonitoringData` operation) to the other components of the Monitoring Tool (Log Manager and Publish/Subscribe Manager) or to the requesters (Actors - e.g. agents or persons - can send to this component explicit requests in order to get monitoring data from software components.)

The Statistics Manager component provides two interfaces. The first one is called _IP Statistics Manager_ (IP: Interface-Provided). This is a public interface and provides the operations that allow configuring the components in order to be monitored. The second one is protected and is called _IPP Statistics Manager_ (IPP: Interface-Provided-Protected). The functionalities provided by this interface are used internally by the other components of the Monitoring Tool.

### IP Statistics Manager Provided Operations

- `configureMonitoringPolicy(actor: Actor, agent: Agent, p[]: Parameters)`

  This operation allows an actor to configure a component in order to be monitored (the agent parameter represents either Software Components, Business and Infrastructure Agents, Services or Computational Resources). If the actor is allowed to configure the component (e.g. if the security tool returns true for the `isAuthorized` operation), the configuration functionality is delegated to the particular component.

- `getMonitoringData(a: Actor, agent: Agent, p[]: Parameters)`

  An actor invokes this operation in order to gather information for a particular component using the pull style (the agent parameter represents either Software Components, Business and Infrastructure Agents, Services or Computational Resources). The Statistics Manager component first checks if the actor is allowed to access the component information. Then it tries to retrieve the required information from memory or from the Log Manager. If the information is found, it returns it to the caller. Otherwise, it delegates the operation to the Collector. Upon the Collector returns the information, the Statistics Manager delivers it to the client and saves the statistics in the Log Manager if necessary.

### IPP Statistics Manager Provided Operations

- `putMonitoringData(d: Data)`

  This operation provides the functionality that allows the Collector to transfer monitoring information to the Statistics Manager with a standard format independent of the style (push or pull). Depending on the information received, the Statistics Manager stores the information in the Log Manager, delivers it to the requestor or notifies the Publish/Subscribe Manager.

### Required Operations

The Statistics Manager requires the following operations from other components of the Entreprise SOA:
configureMonitoringPolicy(actor: Actor, agent: Agent, p[]: Parameters)
This operation is described in the required operations of the Monitoring Tool.

isAuthorized(a: Actor, agent: Agent)
This operation is described in the required operations of the Monitoring Tool.

Log Manager
Stores the monitoring data received from the Statistics Manager component. It provides only one interface called IPP Log Manager (IPP: Interface-Provided-Protected) that includes protected operations called by other components of the Monitoring Tool. The Log Manager does not require operations from other Enterprise SOA components.

IPP Log Manager Provided Operations

setStats(s: Stats)
This is a protected operation called by the Statistics Manager. By using this operation, the Statistics Manager is able to store monitoring statistics about a particular component.

getStats(agent: Agent)
This is a protected operation called by the Statistics Manager. By using this operation, the Statistics Manager is able to retrieve monitoring statistics about a particular component.

Publish/Subscribe Manager
By means of this component, actors can subscribe to events produced by monitored components. It is also in charge of notifying the actors about the events produced. Internally, it correlates each monitored component with the agents (clients) that want to stay informed about the events that occur in the component.

The Publish/Subscribe Manager component provides two interfaces. The first one is called IP Publish/Subscribe Manager (IP: Interface-Provided). This is a public interface and provides the operations that allow the Enterprise SOA components to subscribe to events that may be triggered in other components. The second one is protected and is called IPP Publish/Subscribe Manager (IPP: Interface-Provided-Protected). The functionalities provided by this interface are used internally by the other components of the Monitoring Tool.

IP Publish/Subscribe Manager Provided Operations
- subscribeEvent(a: Actor a, agent: Agent, e: Event)

This functionality allows an actor in the system (e.g. something acting as a client) to subscribe to a type of event on a particular component (the agent parameter represents either Software Components, Business and Infrastructure Agents, Services or Computational Resources). The Publish/Subscribe manager stores the correlation between the components monitored in the system and the actors that are interested in the events on those components. Eventually, the actor will be notified every time an event of that type occurs.

IPP Publish/Subscribe Manager Provided Operations

- setStats(s: Stats)

This is a protected operation called by the Statistics Manager. By using this operation, the Statistics Manager is able to communicate events occurred in a particular component. Upon receiving a message, the Publish/Subscribe Manager gets the actors registered for that particular event on the component and sends them a notification (using the notifyEvent operation on the component).

Required Operations

The Publish/Subscribe Manager requires the following operations from other components of the Entreprise SOA:

- notifyEvent(agent: Agent, e: Event)

This operation is described in the required operations of the Monitoring Tool.

- isAuthorized(actor: Actor, agent: Agent)

This operation is described in the required operations of the Monitoring Tool.

2.2.5.3 Scenarios

The following figures describe several scenarios that are fulfilled with this pattern.

Scenario 1 (Configure Component for Monitoring)

This scenario shows how a component can be configured in order to be monitored. In this particular case, this functionality is called by a system administrator. In Figure 6 is shown a sequence diagram describing the process.
Figure 6 Configure component for monitoring
First of all, the system administrator sends a message to the Monitoring Tool interface in order to configure a particular component, specifying the style and features to monitor in the associated parameters. This operation is implemented by the Statistics Manager. Upon receiving the message, the Statistics Manager checks first if the administrator is allowed to monitor the component. Assuming that he/she is allowed, the Statistics Manager stores in an internal structure the parameters related to the configuration for to take further actions. From this point on, the Statistics Manager delegates the configuration message received to the particular Enterprise SOA component where the component is deployed. The figure shows different alternatives (sub-scenarios) that consider business components placed in the Runtime environment (e.g. services, processes, GUI components etc.), and Infrastructure Agents (e.g. the Registry, the Computational Resources Management Tool or the ESB).

Scenario 2 (Get Monitoring Data from Component)

This scenario shows how an actor of the system can monitor a component explicitly (following the pull style). In this case, this functionality is called by a system administrator. In Figure 7 it is shown a sequence diagram describing the process.
Figure 7 Get monitoring data from component (Pull)
The scenario shows a common part at the beginning and the different sub-scenarios.

In the common part, the administrator sends a message to the Monitoring Tool interface in order to get specific monitoring data (described in the event parameter) from a particular component. This operation is implemented by the Statistics Manager. The Statistics Manager checks first if the administrator is allowed to monitor the component. If true, the Statistics Manager can fulfill the request using different alternatives in the following order:

First, the Statistics Manager tries to find the required information in the information that has in its cache. If the information is found in the cache, it is returned to the client.

If the Statistics Manager can not complete the required information for the component with in-memory data, then it tries to find the information in the Log Manager. Once the information for the component is retrieved from the log, a filter is applied in order to select the appropriate information before delivering it to the requestor.

Finally, if the request can not be fulfilled with the information stored up to now in the Log Manager, the Statistics Manager delegates the call to the final component. This delegation is mediated by the Collector component, which invokes the concrete component for retrieving the data and packs the received information in a format that is independent from the invocation style (in this case pull). Once the monitoring information is packed, the Collector returns it to the Statistics Manager (putMonitoringData). Upon receiving the information, the Statistics Manager unpacks the information, creates the statistics (createStats) and checks the configuration for the requested component. As the configuration shows that it is a pull invocation, the Statistics Manager stores the stats in the Log Manager (setStats) and generates the data required by the requestor (filterStats). Finally, the filtered data is returned to the requestor.

**Scenario 3 (Subscribe to an Event and Notification)**

This scenario shows how a component can be monitored using a publish/subscribe approach (push style) by an actor of the system. In this case, the subscription is performed by a system administrator and the notifications are triggered by the components themselves. In Figure 8 is shown a sequence diagram describing the process.
Figure 8 Monitoring a component using publish/subscribe (Push)
The figure splits the scenario for the subscription to events and the notifications.

In the subscription part (shown on top), the administrator sends a message to the subscribeEvent operation on the Monitoring Tool interface in order to subscribe to particular events triggered by a specific component, getting certain monitoring data (described in the event parameter). This operation is implemented by Publish/Subscribe Manager. First of all, the Publish/Subscribe Manager checks first if the administrator is allowed to monitor the component. If true, the Publish/Subscribe Manager delegates the subscribeEvent operation to the Collector which, in the end, performs the subscription of the Monitoring Tool on the final component in order to be notified about the events on the component. Finally, the Publish/Subscribe Manager registers in an internal structure the correlation between the actor requesting the subscription, the component to monitor and the events to monitor. This will be used when event notifications are delivered to the Publish/Subscribe Manager.

On the notification part (shown below), the components trigger events that are redirected to the subscribers, in this case the Monitoring Tool. Upon receiving an event, the Collector retrieves the data and packs the received information in a format that is independent from the invocation style (in this case push). Once the monitoring information is packed, the Collector returns it to the Statistics Manager (putMonitoringData). Upon receiving the information, the Statistics Manager unpacks the information, creates the statistics (createStats) and checks the configuration for the requested component. As the configuration shows that it is a push invocation, the Statistics Manager stores the stats in the Log Manager (setStats) and also delivers the information to the Publish/Subscribe Manager. Finally, the Publish/Subscribe Manager generates the data required by the requestor (filterStats), gets the actors subscribed to the event and sends them a notification message including the required data.

2.2.6 Relationships to other patterns

This section provides the relationships that the Monitoring pattern has with the top-level Enterprise SOA pattern. The relationship is shown in Figure 9.
The Monitoring pattern is part of the Enterprise SOA pattern because provides an architectural solution for a subset of components belonging to the Enterprise SOA pattern. In this case, the Monitoring pattern specializes the Monitoring Tool component.

Also the Publisher-Subscriber pattern [BRMS+] is also part of the Monitoring pattern. The Publisher-Subscriber pattern is based on the Observer pattern [GHJV] and is used to inform several components about certain events that have occurred in the system. In this pattern, a publisher notifies about certain state changes to a set of registered components -called subscribers- that have been registered in the publisher. In the Monitoring pattern, the Publisher-Subscriber pattern is used to allow the clients of the Monitoring Tool to register to certain events that may occur in the Enterprise SOA infrastructure in order to be notified when they are triggered.

2.2.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.2.7.1 Abstract Components

The abstract components related to this pattern are those described in the solution section. All of them form a basic architecture for monitoring:

**Collector:**

This component is in charge of collecting the required monitoring data. It is also the component that isolates other components of a monitoring solution from the particular interface style (push or pull) used to collect monitoring data.

**Statistics Manager:**

It is in charge of delivering the monitoring information received from the Collector to the information requesters (either administrators, users or other components, including the Log Manager and Publish/Subscribe Manager).

**Log Manager:**

Figure 9 Pattern relationships
It is in charge of storing in a persistent storage the monitoring data received from the Statistics Manager component.

**Publish/Subscribe Manager:**

It is in charge of registering components to receive the events produced by monitored components and notifying those components when the events are produced.

### 2.2.7.2 Concrete Components

The components described in the solution are generic. This allows to implement them in monitoring solutions based on different monitoring specifications, such as the JMX specification [SUN-JMX] and the EGA Reference model [EGA1, EGA2] (See next section).

### 2.2.8 Relationships to Standards Catalogue

A well-known example of specification for building management/Monitoring Tools is JMX [SUN-JMX]. JMX is a Java specification included in the Java Community Process (JCP) that allows to create management architectures, providing APIs and services for building Web-based, distributed, dynamic and modular solutions to manage/monitor Java resources. JMX has been integrated in most J(2)EE application servers (JBoss, JOnAS, WebSphere, WebLogic, etc.) as the solution to manage/monitor such complex environments.

The JMX architecture is presented in the following figure:

![Figure 10 JMX Architecture](image)
The managed resources are abstracted (wrapped) by components called *MBeans* (Managed Beans). The architecture of JMX is organized into layers using the layers pattern [BMRS+96]. The first level is the so-called the *Probe level* or *Instrumentation level*. It contains the MBeans instrumenting the resources. As occurs in this pattern, the resources monitored by JMX must provide an interface to access the attributes to monitor by means of the corresponding MBean. It also defines a generic notification model based on the Java event model that lets developers build proactive management solutions in a similar way as the pattern described here allows with the push approach. The next level is the *Agent level*. This level acts as an intermediary between the MBean and the applications. It provides a specification for implementing agents, which control the resources and make them available to remote management applications. Agents are usually located on the same machine as the resources they manage, but this is not a requirement. A JMX agent consists of an MBeanServer and a set of services for handling MBeans. The MBeanServer is the core concept of JMX and acts as a Mediator [GHJV95] between the managed elements and the management applications. Finally, the *Remote Management level* enables remote applications – e.g. management consoles – to access the MBeanServer by means of connectors and adaptors. A connector provides full remote access to the MBeanServer API using various communication frameworks (RMI, IIOP, JMS, WS-*,...), while an adaptor adapts the API to another protocol (e.g. SNMP) or to Web-based GUI (HTML/HTTP, WML/HTTP,...).

The main benefits provided by the JMX architecture are the following:

- Applications can be managed/monitored with little impact on their design. The MBeanServer is the key component that enables this feature.
- The JMX architecture is scalable. The component-based approach allows the implemented solutions to scale from small devices to large telecommunications switches.
- The architecture is adaptable in order to integrate future management/monitoring concepts. The architecture allows to implement flexible and dynamic management/monitoring solutions that can adapt new technologies; for example, a JMX solution can use other lookup and discovery services from technologies such as Jini, UPnP or Service Location Protocol (SLP).

The EGA Reference model [EGA1, EGA2] also includes monitoring features in the specification. Monitoring in EGA allows observing the performance metrics of a grid component, either by passively by monitoring the request processing, or actively by creating and submitting requests aiming at measuring the performance of particular grid components. The *grid management entity (GME)* is the element that represents the holistic management of a full data centre, including monitoring. The GME encompasses both automated and human management and the goal is to delegate all management/monitoring tasks to it (e.g. the GME is in charge of managing the life cycle of a grid component). As the approach of the pattern described here does, the GME abstracts the
management/monitoring tasks from the managed components (using a Façade pattern [GHJV95]) although in practice managed components contain a fraction of the management entity. This also allows to build hierarchical and distributed GMEs. The main operations related to monitoring provided by the GME are similar to the ones provided by this pattern: get the status of a grid component, update the configuration of grid components, etc.

Other specifications such as Simple Network Management Protocol [SNMP] or Web Services Distributed Management [WSDM] also include monitoring among their features.

2.2.9 Application Examples

Monitoring Tools are present in any enterprise platform because monitoring is a cornerstone in real world environments. The following figure shows a monitored environment with three components: a BPEL manager, a web crawler and a transaction manager.

![Monitoring Example](image)

Figure 11 Monitoring Example

This environment is monitored through SNMP and JMX. Every monitored component provides an agent. As the components are also implemented in different languages (Java and C++), the required adapters for each possible configuration have been developed. The JMX console gets the monitoring information from the components through the MBeans and the SMNP console through the corresponding SMNP adaptor of each component. The figure also
show different deployment configurations for the agents, that can be either collocated with the monitored component itself or be external to it.

2.2.10 References


[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


2.3 Service Runtime: Multi-Tier Transactional Service Runtime Pattern (UPM)

2.3.1 Problem Description

Many applications have strict consistency over the data they handle. The requirements are typically the well-known transactional ACID properties that almost all database management systems (DBMSs) guarantee today:

- **Atomicity.** It guarantees that either all of the operations included in a transaction are performed or none of them are.
- **Consistency.** It ensures that a database is in a consistent state before the start of a transaction and reaches another consistent state once the transaction has finished (whether successful or not).
- **Isolation.** It is constraint that requires that during a transaction execution, the operations performed by other transactions cannot access or see the data in an intermediate state of the transaction.
- **Durability.** It guarantees that, once a transaction has finished, the changes will persist, and not be undone.

Additionally, for the sake of maintainability, separation of concerns is another important requirement for applications.
As we have mentioned before, in order to address the ACID properties, specific support is required (e.g. a database). Therefore, the service runtime requirements are often extended with requirements of the ACID behaviour and separation of concerns. In this way, first, services can be enriched automatically with the ACID properties. Second, applications can be developed separating the different concerns of business logic and data storage.

Thus, it is important to separate the business logic runtime from the storage software artefacts involved in the architecture, taking into account the peculiarity of each of them when providing a useful environment for running service-oriented applications.

2.3.2 Functional Requirements

The diagram in the figure below introduces some actors involved into the scenarios describing the functionalities, the components and their relationships.

![Diagram](image)

**Figure 12 Functionalities of the Multi-Tier Transactional Service Runtime**

Apart from the functionalities offered by the pattern that is being specialized (in this case the Designer and Runtime Tools for E-SOA pattern, see Section 0), this pattern adds persistence functionalities. The specific components of this pattern allow either accessing data stored in a transactional persistent storage area or persisting data, improving the availability of data in the global infrastructure. So, the persistState/Data use case (shown as a filled oval in the figure above) has been added to the functional requirements of the extended pattern. This use case extends the functionality provided by the sendMessage use case of the Designer and Runtime Tools for E-SOA pattern.
sendMessage(a: Provider Agent, m: Message): it allows an Agent to send the Message m to the Provider Agent a running on the Multi-Tier Transactional Service Runtime.

deploy(ssc: Simple Service Component): it allows to deploy the Simple Service Component ssc into the Service Runtime.

undeploy(ssc: Simple Service Component): it allows to undeploy the Simple Service Component ssc from the Service Runtime.

startAgent(ssc: Simple Service Component): Provider Agent: it execute the Simple Service Component ssc by creating a correspondent Provider Agent running on the Multi-Tier Transactional Service Runtime.

stopAgent(pa: Provider Agent): it stop the execution of the Provider Agent pa running on the Multi-Tier Transactional Service Runtime.

getLoggedData(): it returns the data logged by this tool.

subscribeEvent(et: EventType): it allows a client to subscribe its interests to a type of Event. The client will be notified every time an Event of that type occurs.

configureMonitoringPolicy: it allows the monitoring tool to configure the monitoring policy for this component.

persistState/Data: This use case extends the functionality of the sendMessage use case in order to provide transactional read/write functionalities for state/data in a persistent database when a service is being realized. These functionalities are realized by the corresponding Software Services offered by the Provider Agents.

2.3.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>Buildability</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
<tr>
<td>Availability</td>
<td>-</td>
</tr>
<tr>
<td>Scalability</td>
<td>-</td>
</tr>
<tr>
<td>Reusability</td>
<td>+</td>
</tr>
<tr>
<td>Usability</td>
<td>+</td>
</tr>
<tr>
<td>Resource Efficiency</td>
<td>-</td>
</tr>
<tr>
<td>Reliability</td>
<td>+</td>
</tr>
</tbody>
</table>
2.3.3.1 Rational

The architectural choices made in this pattern correspond to a functional decomposition of the Service Runtime presented in the Designer and Runtime Tools for E-SOA pattern. These choices allow providing the same quality attributes as that pattern enhancing some of them. Additionally, Reliability also appears as a new non-functional quality attribute provided by this pattern.

- Buildability and Maintainability: Due to the separation of concerns made in this pattern, different servers are architected for each tier, avoiding a monolithic runtime more difficult to build and maintain.
- Availability: No availability is provided by the pattern since every component is a single point of failure for the architecture.
- Scalability: Every component becomes a single scalability bottleneck, therefore, no scalability is provided by the pattern.
- Usability: By enabling the separation of functions (business logic from data storage), the pattern eases the service/application development and the management of the underlying components of the multi-tier infrastructure.
- Resource Efficiency: The decomposition of the runtime components impact negatively on the resource efficiency. Indeed, each runtime component is not aware of the operative status of the other components and this can jeopardize the application of ad hoc resource allocation policies that allows the optimization of the resources usage.
- Reliability: By enforcing the ACID properties, data reliability is enforced in the architecture in the advent of concurrent accesses to data and failures.

2.3.4 Assumptions

This section describes the context where the proposed pattern is applicable.

Figure 13 Required functionalities

The following functionalities are required from the context (See Figure 13):
- **sendMessage**: the Multi-Tier Transactional Service Runtime requires this interface to forward the Message \( m \) sent by an Provider Agent running on it to the Agent \( a \).

- **notifyEvent**: the Multi-tier Transactional Service Runtime requires this functionality to notify events to the clients that have subscribed their interest though the subscribeEvent method.

- **assignCR**: this function is needed to assign computational resources to the different Agents that run on the Multi-Tier Transactional Service Runtime. It must be provided from other parts of the system.

- **isAuthorized**: the Multi-Tier Transactional Service Runtime requires this operation from the other parts of the system to check if the client has the rights to access provided functionalities of this component.

### 2.3.5 Solution

This pattern captures the specific properties of the different Software Components of a service runtime with a multi-tier structure and transactional persistence capabilities. As it has been stated before, it extends the functionalities of the Service Runtime component described in the Designer and Runtime Tools for E-SOA pattern.

Figure 14 shows the operations that the Multi-Tier Transactional Service Runtime subsystem provides (IP_MTTSR) to enable the external users or systems to interact with it. Similarly, the required operations (IR_MTTSR) enable the subsystem to interact with external systems. In this diagram the MTTSR subsystem is a black box and focus on its interfaces.
Figure 14 Multi-Tier Transactional Service Runtime Operations

Internally the Multi-Tier Transactional Service Runtime exposes the same interface than the Service Runtime component shown in Section 2.2.4. However, as it is shown in Figure 15, internally it includes two main components:

1. **Application Server**. It provides the same functionality as the Service Runtime, that is, it is the component that hosts Provider Agents implemented by Simple Service Component. It uses the functionalities provided by the Database component in order to transactionally read and persist state/data.

2. **Database**. This component provides the capabilities for reading/persisting data transactionally in a persistent storage. It is used by the Provider Agents running in the application server.

Each one of these components has a proxy [BMRS+96] on the client side (Application Server Proxy and Database Proxy in the figure respectively) that interfaces the real component.

The other two components shown in Figure 15 (Management Façade and Monitoring Façade) are already present in the Designer and Runtime Tools for E-SOA pattern and provide the same functionality as in that pattern. They will be described also here for the sake of readability.
Figure 15 Multi-Tier Transactional Service Runtime Components

Figure 15 also provides the relationships among these components. The following section focuses on each one of these components to clarify the role of them into the infrastructure and which are the functionalities they provide.

2.3.5.1 Component Description

The following figure (Figure 16) shows the structure of the solution provided by the Multi-Tier Transactional Service Runtime pattern. The Monitoring and Management components have been omitted from this diagram in order to highlight the tiered architecture.
Figure 16 Internal structure of the pattern
Figure 17 Application Server component and the provided and required interfaces

This component provides and requires the same interfaces as the Service Runtime specified in the Designer and Runtime Tools for E-SOA pattern. The Application Server is the component that hosts Provider Agents implemented by Simple Service Components. This component allows to deploy and undeploy Simple Service Components and to create or destroy Provider Agents that implements Simple Service Components deployed on it. Finally, it enables the hosted Provider Agents to receive messages from and send messages to other Agents. The state/data managed by Provider Agents can be read or persisted transactionally using the Database component. The Application Server uses the functionalities of the Transaction Manager in order to demarcate the transactions and take control of them at this level of the Multi-Tier Transactional Service Runtime (MTTSR).

Provided Operations

The Application Server provides the following operations:

- sendMessage(a: Provider Agent, m: Message)
  It allows an Agent to send the Message m to the Provider Agent a running on the Service Runtime.

- deploy(ssc: Simple Service Component)
  It allows to deploy the Simple Service Component ssc into the Service Runtime.

- undeploy(ssc: Simple Service Component)
  It allows to undeploy the Simple Service Component ssc from the Service Runtime.

- startAgent(ssc: Simple Service Component): Provider Agent.
  It execute the Simple Service Component ssc by creating a correspondent Provider Agent running on the Service Runtime.
- **stopAgent(pa: Provider Agent)**
  It stop the execution of the Provider Agent pa running on the Service Runtime.

- **getLoggedData()**
  It returns the data logged by this tool.

- **subscribeEvent(et: EventType)**
  It allows a client to subscribe its interests to a type of Event. The client will be notified every time an Event of that type occurs.

- **configureMonitoringPolicy()**
  It allows the monitoring tool to configure the monitoring policy for this component.

**Required Operations**

- **sendMessage(a: Agent, m: Message)**
  The Service Runtime requires this interface to forward the Message m sent by an Provider Agent running on it to the Agent a.

- **notifyEvent(e: Event)**
  The Service Runtime requires this functionality to notify events to the clients that have subscribed their interest though the subscribeEvent method.

- **assignCR()**
  This function is needed to assign computational resources to the different Agents that run on the Service Runtime. It must be provided from other parts of the system.

- **isAuthorized()**
  The Service Runtime requires this operation from the other parts of the system to check if the client has the rights to access provided functionalities of this component.

**Application Server Proxy**

This component interfaces the real application server component on the client side. It provides the same operations as the application server component described in the previous section.

**Database**
Figure 18 Database component and the provided and required interfaces

This component is in charge of persisting or retrieving state/data transactionally on the persistent storage depending on the instructions dictated by the Provider Agents running on the Application Server. So, It is in charge of correlating the Provider Agents with their corresponding transactions. This is done in order to guarantee the ACID properties for the state/data managed by the Provider Agents in the context of the MTTSR.

Provided Operations

The Database provides these operations:

- **beginTx(pa: Provider Agent): TransactionId**
  
  The Database component starts a new transaction triggered by the Provider Agent pa returning the transaction identifier that univocally identifies the transaction in the Multi-tier Transactional Service Runtime.

- **commitTx(tid: TransactionId)**
  
  The Database finishes successfully the transaction identified by the corresponding tid, persisting the changes in state/data performed during the transaction. The changes in state/data performed during the transaction are accepted and must be visible in the Database for other concurrent transactions.

- **abortTx(tid: TransactionId)**
  
  The Database finishes successfully the transaction identified by the corresponding tid discarding the changes in state/data performed during the transaction.

- **read(pa: Provider Agent, tid: TransactionId, q: Query): Data**
It retrieves the Data specified by the ProviderAgent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. The required data is specified by means of a query q. It associates the data read in the transaction with TID tid with the corresponding Provider Agent.

- write(pa: Provider Agent, tid: TransactionId, d: Data)

It persists the Data d specified by the Provider Agent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. It associates the data written in the transaction with TID tid with the corresponding Provider Agent.

- setTrigger(name: TriggerName, def: TriggerDefinition, code: TriggerCode)

It allows to define a trigger with name name. A trigger is a procedure that is executed when a condition is fulfilled on performing insert, update or delete operations in the database. The concrete conditions and operations are defined in the def parameter, and the procedure itself is specified by the code in the code parameter.

- unsetTrigger(name: TriggerName)

It allows to remove the trigger with name name from the database.

- inspectLog(range: LogRange): LogData

This operation allows performing log mining. That is, to retrieve a part of the database log specified in the range parameter.

Database Proxy

This component interfaces the real database component on the client side. It provides the same operations as the database component described in the previous section.

Management Façade
Figure 19 Management Façade component and the provided and required interfaces

The Management Façade is the component that is responsible to dispatch an operation to the specific component. Dispatch of operation means that when a client ask for an operation provided by the Management Façade, this component process the request and select the specific runtime that have to execute such operation. In particular, it manage the deployment and undeployment policies of Software Components, and the starting and stopping policies of the Agents into the runtime components. To accomplish its tasks the Management Façade asks to runtime components to deploy/undeploy Software Components and to start/stop Agents. For example, a user that have to deploy a Simple Service Component ask to the Management façade to do that. The Management Façade, according to its internal policies, recognizes that the service runtime is responsible for the Simple Service Component and ask to it for the deployment.

The exposed operations are:

**Provided Operations**

- deploy(sc: Software Component)
  
  It allows deploying the Software Component sc into the corresponding runtime.

- undeploy(sc: Software Component)
  
  It allows undeploying the Software Component sc from the corresponding runtime.

- startAgent(sc: Software Component): Agent

- stopAgent(a: Agent)
It allows executing the *Software Component* $sc$ on the corresponding runtime. The Management Façade ask to the runtime to create the *Agent*.

- **stopAgent**($a$: *Agent*)

It allows stopping the execution of the *Agent* $a$ running on a runtime. The Management Façade asks to the corresponding runtime to stop the *Agent*.

**Required Operations**

- **deploy**($sc$: *Software Component*)

The Management Façade requires this operation from all the runtime components to deploy the *Software Component* $sc$ into the appropriate runtime.

- **undeploy**($sc$: *Software Component*)

The Management Façade requires this operation from all the runtime components to undeploy the *Software Component* $sc$ from the appropriate runtime.

- **startAgent**($sc$: *Software Component*): *Agent*

The Management Façade requires this operation from all the runtime components to ask them for the execution of the *Software Component* $sc$ to the appropriate runtime.

- **stopAgent**($a$: *Agent*)

The Management Façade requires this operation from all the runtime components to ask them for stopping the execution of the *Agent* $a$ running into the appropriate runtime.

**Monitoring Façade**
The Monitoring Façade is the component that is responsible to collect monitoring data from the components that must be monitored according to the configured monitoring policies. Moreover, it manages the subscription policies of the events. This means that when a client ask for a subscription of an event, the monitoring façade is responsible to dispatch the event subscription to the specific runtime component, according to the defined monitoring policies.

The exposed operations are:

**Provided Operations**

- `getLoggedData()`
  
  It allows to retrieve the monitoring data provided by the components that must be monitored.

- `subscribeEvent(et: EventType)`
  
  This functionality allows a client to subscribe its interests to a type of Event. The client will be notified every time an Event of that type occurs.

- `configureMonitoringPolicy()`
  
  This functionality allows to configure the monitoring policies and features of the Enterprise SOA infrastructure and business components.

**Required Operations**

- `getLoggedData()`
  
  The Monitoring Façade requires this functionality to collect monitoring data from the tools that must be monitored.

- `subscribeEvent(et: EventType)`
It allows a client to subscribe its interests to a type of Event. The client will be notified every time an Event of that type occurs.

- `configureMonitoringPolicy()`

  This functionality is required by the Monitoring Tool in order to configure the monitoring policies of the components that need to be observed.

### 2.3.5.2 Functionality Descriptions

In this section is described how the different components collaborate to realize the provided functionalities of the Multi-Tier Transactional Service Runtime. The realization of some internal functionalities is described as well, to clarify the behaviours of the system when they can raise some ambiguity or they can be useful for the refinement of this pattern.

#### Send message involving management of persistent data

In this section are discussed the behaviours of the subsystem when an exchange of messages between *Agents* that imply to persist and retrieve data transactionally takes place. As reported in the diagram in Figure 21, the focus is on the components and their interactions that participate to the realization of the external functionality `sendMessage()`.
Figure 21 Multi-Tier Transactional Runtime Components Participating in the Exchange of Messages

The sequence diagrams in Figure 22 and Figure 23 show how an Agent communicates with a Provider Agent that gets/persists data/state in a database.
using transactions. For the sake of clarity, the messages that are exchanged inside the Multi-Tier Service Runtime are depicted in Figure 23.

The Agent starting the communication can be external or internal to the NCI. In both cases the message is sent to the external system that is responsible for manage the message applying the required transformation and dispatch it to the right recipient (in this case will be the Application Server). The External System also performs security check to look for authorization to forward the message.

If the Agent is not authorized, the message is not sent. Otherwise, the external system detect if the recipient is running into the NCI or outside of it. If it is running outside, the message is forwarded to the external provider agent.

If the recipient is running into the NCI, the external system sends the message to the Application Server component by means of its proxy. The Application Server constitutes the entry point for the Multi-Tier Transactional Service Runtime.
Figure 22 A Generic Agent Sends a Message to a Provider Agent Running on the Multi-tier Transactional Service Runtime
Figure 23 Messages Exchanged Inside the Multi-Tier Transactional Service Runtime
Then, the Application Server starts a transaction using the functionality provided by the Database component. This message is sent through the database proxy. The Transaction Manager of the database (not shown in the picture) returns a transaction identifier (TID) to the Provider Agent. This TID will be used to identify the subsequent read/write operations performed on the Database component and to commit or abort the changes produced in the transaction.

The scenario shows the Provider Agent performing the two alternatives for state/data persistence (read or write) and the two alternatives that exist for finishing a transaction (commit/abort).

When the Provider Agent must read state/data stored in the Database component in order to provide the required service, it retrieves it from the Database by means of a query. Otherwise, when the Provider Agent must persist some state/data in the Database, it specifies the state/data to write as a parameter in the message. It is important to note that, in order to preserve consistency, the changes in state/data produced in a transaction by write operations should only be visible to that transaction. This means that the TID must be specified in every read or write message sent to the Database.

Once the required read/write operations have been done, the transaction can be either committed or aborted. If the transaction is committed, the changes in that transaction are validated and persisted in the database in order to avoid inconsistencies with other concurrent transactions. On the other hand, if the transaction is aborted, the changes are discarded. As it can be observed in the figure, the Database is informed in order to persist or discard the changes when the transaction finishes.

**Setting a trigger on an update operation**

In this section is discussed the behaviour of the subsystem when a trigger is set on an update operation. The process is shown in Figure 24.

First of all, the trigger is registered on the database through the database proxy. The trigger name allows the database kernel to reference the trigger and locate its code when required. In the trigger definition is specified the kind of operation that will execute this trigger as well as other features if required. The trigger code includes the sentences that form the trigger itself.

Then, a transaction is started and two operations are performed on the database, first a read operation and second a write (update) operation. Upon the transaction is committed, the database checks if there is a trigger that must be executed. The database finds the previously registered trigger fulfilling the conditions, and finally executes it.
Figure 24 Trigger on an update operation
2.3.6 Relationships to other patterns
This section provides the relationships that the Multi-Tier Transactional Service Runtime pattern has with other patterns.

![Diagram of relationships between patterns]

Figure 25 Relationships of “Multi-Tier Transactional Service Runtime” pattern

The Multi-Tier Transactional Service Runtime pattern specializes the Service Runtime component of the Designer and Runtime Tools for E-SOA pattern.

2.3.7 Relationships to Components Catalogue
The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.3.7.1 Abstract Components
In many existing frameworks for building multi-tier applications is crucial the notion of application server. The application server contains the business logic of applications. This notion is has been captured in this pattern by the abstract component Application Server of the NEXOF-RA.

These frameworks use database management systems in order to persist the data managed by applications. Data persistence has been captured in this pattern by the abstract component Database of the NEXOF-RA.

2.3.7.2 Concrete Components
In real architecture implementations, the functionality of the application server abstract component can be provided by the so-called application servers, that also include other services for facilitating application development (e.g. JBoss
[JBo], Oracle Application Server [OAS], Neweaver [SAP], JOnAS [Jon], Weblogic [BEA], Websphere [IBM]) or by the so-called servlet containers (e.g. Tomcat Servlet Container [Tom], Jetty Servlet Container [Jet]). The original functionality of servlet containers is not to implement the business logic of applications but to provide the presentation layer for web applications.

Some of the most well-known concrete database system components available these days are:

- Microsoft’s SQL Server [MSS],
- Oracle [Ora],
- IBM’s DB2 [DB2], and
- Sybase [Syb],
- PostgreSQL [Pos]
- MySql [MyS]
- ...

Object-oriented databases or ad-hoc databases can also be used in addition to relational databases.

2.3.8 Relationships to the Standards Catalogue

Nowadays, there are many different standards that currently allow the construction of multi-tier architectures. We can distinguish between de iure standards and de facto standards. In the first group, are included the CORBA [Cor] standard from the Object Management Group (OMG) and the J2EE/JEE [JEE] from the Java Community Process (JCP). Some examples of the second group are frameworks based on the model-view-controller such as Ruby on Rails [RoR] or Struts [Str], and other alternatives such as Spring [Spr] or .NET [Net] [Str]. Many of these frameworks such as J2EE/JEE or Struts, define open architectures (by means of open-source projects) that allow the community to improve the products. Nowadays, the most used standard in the enterprises is J2EE/JEE [Can07].

Regarding to databases, the most well-known and used currently in enterprises are relational databases. In relational databases, the SQL standard is the de iure standard. The SQL standard includes support for triggers. The interfaces for examining the log are de facto standards provided by some database vendors such as Oracle or IBM. Open-source databases such as PostgreSQL also provide external tools for examining the log.

2.3.9 Application Examples

This pattern is applied in current multi-tier architectures. This kind of architectures is widely used in current enterprise information systems. Currently, many web-based applications are built using this approach. A multi-
tier architecture is a client/server-based infrastructure in which an application is split in different layers (tiers), each one providing well-defined interfaces that are executed by distinct software elements.

The three-tier architecture is a well-known example of this kind of architectures. In a three-tier architecture an application is split in presentation, application and data tiers. The presentation tier represents the user interface of the application. In the application tier, also known as middle, business or logic tier, it is built the logic part of the application. The functionality in this tier corresponds to the functionality provided by the Application Server component presented in this pattern. Finally, the data or back-end tier allows the access to the persistent information required by the application. The functionality in this tier corresponds to the functionality provided by the Database component of this pattern. Each tier can be deployed in the same centralized node or in different nodes connected through a network. Figure 26 shows an example of a three-tier architecture in which each tier is distributed in a different node.

![Three-Tier Architecture Diagram]

**Figure 26 Example of a Three-Tier Architecture**

2.3.10 References


2.4 Security in Enterprise SOA Pattern (Thales)

2.4.1 Problem Description

Nowadays, the value of SOA is not discussed any more. In this context, the security has become a crucial property for service oriented architectures (SOA) and its underlying infrastructure.

One can distinguish two kinds of security:
1. the first one is inherent to the infrastructure itself (how users can trust the infrastructure),

2. the second one deals with the security services (guaranteed by a SLA) provided by the infrastructure itself.

However, the overall security architecture relies on three fundamental IT infrastructure components: network services, host operating systems and target application.

All these techniques will use state of the art technologies such as:

- Intrusion detection and connection control
- Protection against malicious code
- Cryptography
  - Strong authentication
- Public Key infrastructure (PKI)
- Identity Management (IM)

In the context of ESOA, the base functionalities rely on:

- authentication,
- authorization,
- security policy management (configuration of the security policy),
- security proactive management (which treats the policy of security events),
- identity management,
- trusted timestamping,
- non-repudiation.

The aim of this document is to provide generic patterns securing an SOA infrastructure based on the assumptions listed above.

### 2.4.2 Functional Requirements

The following use-case diagram shows the functional requirements related to Security that this pattern meets. This use-case elaborates a set of security functions which can help other patterns of ESOA to manage their security. It depends on the application context; each functionality can be used separately or combined with other functionalities. To know what security is adequate for the system, a complete study of expression of needs must be conducted.
Figure 27 Functionalities provided by the Security Tool for ESOA

- **authenticate**: offers the functionality to identify and to authenticate an external agent or an entity. Authentication Service should offer the possibility to change the authentication protocol.

- **isAuthorized**: offers the functionality to control access of the external actor to the resources. Authorization service offers tools and techniques to make authorization decisions based on the functions, role or task performed by individual actor, to grant access rights to service or process.
In the context of ESOA, this functionality permits to authorize an agent or an entity to access to a resource, or more precisely, to perform a specific action on this resource.

- **configureSecurityPolicy:** The Security Policy Configuration products tools and data base to introduce security requirements that a NEXOF framework instantiation (for example: Transportation critical infrastructure) must ensure the security of information and information technology (IT) assets under their control. An Operational Security policy is never generic. It stems from the security requirements and the strategic issues specific to a particular activities domain.

As a security policy is never generic, this functionality permits to modify an established policy according to the requirements. In the context of the authorization, it permits to modify the access control policy. Concerning the authentication, it permits to configure the protocol.

- **subscribeEvent:** This functionality is already defined into Monitoring Pattern document. It allows an agent to subscribe its interests to different types of events produced in components.

- **getLoggedData:** This functionality is already defined into Monitoring Pattern document. This functionality must be provided by the components that are able to be monitored explicitly in order to extract monitoring data. The functionality must return the data specified by the consumer from applications and services running on top of a NCI, the NCI components and the computational resources of the E-SOA infrastructure.

- **configureMonitoringPolicy:** This functionality is already defined into Monitoring Pattern document. This functionality is required by the Monitoring Tool in order to configure the monitoring policies of the components that need to be observed.

- **suspend:** Offers the functionality to suspend the security agent. In mode automatic, the Proactive Security Management can decide to suspend an agent depending to the security policy decided by the enterprise. In manual mode, the administrator can suspend an agent which is subjected to a breach.

- **reactivate:** Offers the functionality to reactivate an agent. This action is often done by an administrator which is a security expert. Before reactivating an agent, he must analyse a breach subjected by this agent.

- **displayEvent:** Offers the functionality to display events which are notified by other security components (for example: Authentication Service is subjected to a breach, it notifies event to Security Component in order to display this event).

- **storeIdentity:** Offers the functionality to store or update the properties of an entity.
• **getIdentityAttributes**: offers the functionality to get properties of an entity.

• **deleteIdentityAttributes**: offers the functionality to delete properties of an entity.

• **createTimestamp**: offers the functionality to create a trusted timestamped message. A trusted timestamped message is achieved by a complex mechanism which will be described a section of the solution and also be expressed more details in the separate file [Trusted Timestamping Pattern].

• **checkTimestamp**: offers the functionality to check if the time of the message and also the document is corrupted or not. [Functionality of Trusted Timestamping Pattern].

• **sign**: offers the functionality to digitally sign the data (document or message). The digital signature is a process that permits to crypt the data with the private key and after to attach the certificate to this crypted message. [Functionality of Non-Repudiation Pattern].

• **isNonRepudiation**: offers the functionality to check if the data is corrupted or not. [Functionality of Non-Repudiation Pattern].

The following functionalities are showed in the solution section of the Trusted Timestamping and Non-Repudiation patterns.

• **unsign**: offers the functionality to decrypt digitally the signature of the data (document or message). [Functionality of Non-Repudiation Pattern].

• **hash**: offers the functionality to hash a message or a document. The hash can use different mechanisms of hash. [Functionality of Non-Repudiation Pattern].

• **getKey**: offers the functionality to get the cryptographic key. This functionality is one of provided functionalities. Others can be described in the XKMS². [Functionality of Non-Repudiation and Trusted Timestamping Patterns].

• **verifyCertificate**: offers the functionality to get a status of the certificate. The verifyCertificate is often mandatory to check the revocation of the certificate. [Functionality of Non-Repudiation Pattern].

• **timestamp**: offers the functionality to put a time and date on document or a message. [Functionality of Trusted Timestamping Pattern].

### 2.4.3 Non-Functional Qualities (Quality Attributes)

In general manner, the Security In Enterprise SOA pattern will affect positively or negatively the following quality attributes.

---

² XKMS : XML Key Management Specification
Generally, the CIA triad (confidentiality, integrity, and availability) must be the atomic quality attributes.

For the context of Enterprise SOA, only the quality attributes directly linked with the provided functionalities and the quoted security components are stressed below:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>+</td>
</tr>
<tr>
<td>Integrity</td>
<td>+</td>
</tr>
<tr>
<td>Authenticity</td>
<td>+</td>
</tr>
<tr>
<td>Accountability</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
<tr>
<td>Performance (efficiency)</td>
<td>-</td>
</tr>
</tbody>
</table>

2.4.3.1 Rational

All security functions will decrease the performance of the system. For this reason, an adequate choice between the needs of the security and the performance must be considered each time to implement the security on a system. For example, a set of files can be read by all public users, the SSL\(^3\) is not necessary to secure the network between the end user and the server.

Confidentiality ➔ In the general context or particularly in the context of ESOA, the confidentiality is managed by two security components “Authorization and Encryption”. If a message or sensitive data is encrypted, the message can be guaranteed against the thief of information. As the consequence, the confidentiality of the sensitive data is assured because the data is accessible or modifiable only with the authorization and the owner decryption key.

Integrity ➔ The aim of integrity is to guarantee that a data is not modified either accidentally or maliciously. The integrity can be obtained with different manners. The encryption of a document protects against to the unauthorized access. The digital signature guarantees that the document has not been modified since the signature. Trusted time-stamping adds the information of the time of the signature. The Authorization Service controls the access to the external agent (resources, software system...) and assures that the modification on a document is made by the authorized entity. The architectural choice should respect the standard specification such as XACML\(^4\) of OASIS. Based on the standard specification permits to gain a large use of the ESOA and adopts the best practices to guarantee only an authorized access to the resource or to the provider agent.

\(^3\) SSL Secure Socket Layer

\(^4\) XACML: eXtensible Access Control Markup Language
**Authenticity**  ➔ A data is considered as authentic when it can be used as a legal proof. Digital signature and time stamping are good tools to constitute such legal proofs.

**Accountability**  ➔ Provides the ability to enforce accountability or imputability, guarantying the transparency of the operations. Several security components enable accountability. The logging of all events in an audit trail is the most direct way. Authentication contributes to give evidence of accountability. In a more sophisticated way, the architectural choice with the TSA\(^5\) permits to trust the time of a document or a message. This choice should implement the best practices concerning the trusted timestamping which can be found in some standard specifications such as RFC 3161.

**Performance**  ➔ performance of a security function is acceptable if its implementation does not impact in a significant way the global performance of the infrastructure.

**Maintainability**  ➔ the overall security architecture is based on the SOA. All security components are seen as a service. By this way, they inherit the benefits of SOA: interoperability, loose-coupling. And the maintenance of the components is eased by decomposing the Security Tool into loosely coupled sub-components. Each sub-component fulfils a functionality, so it is possible to modify the implementation of a particular functionality without affecting the rest of the components.

### 2.4.4 Assumptions

This section describes the context where the proposed pattern is applicable.

---

**Figure 28 Functionalities required by the Security Tool**

The diagram in Figure 2 shows the functionalities required by the Security In Enterprise SOA to the other parts of the system before it makes sense to apply the pattern.

The proposed pattern is applicable in a context where the following functionalities are provided:

---

\(^5\) TSA: Timestamping Authority
notifyEvent: this functionality is required to notify events to the external clients that have subscribed their interest though the subscribeEvent method. E.g. an organization can be interested in the publication of new Agent Descriptions in the registry of another organization.

Within this pattern description, the functionalities described above are assumed to be provided by an actor named: **External Provider Agent**.

### 2.4.5 Solution

The security solution provides a set of functionalities which can be used in the top-level pattern such as the case of ESOA. These functionalities come from the work of WP4 and also from the Investigation Teams. Certain functionalities are frequently used by the other concerns as the Authentication, Authorization. Others remain optional for the advanced use.

Before detailing the scenarios of provided functionalities, the following section describes all main components which participated in the building of this solution.

### 2.4.5.1 Security Tool Components

This pattern captures the distinctiveness of the different Software Components managed by the Enterprise SOA and provides a specific Security Tool Component for each of them.

The diagram below shows the provided and required interfaces the system exposes to connect to other systems.
The following picture depicts the components of the system and their interconnections.

**Figure 29 Required and provided interfaces of the Security In ESOA**

**Figure 30 Components and their interrelations of Security In ESOA**
The diagram in Figure 30 shows the components and their interrelations that the “Security in ESOA” provides for each specific Software Component.

Before describing all components, their interrelations are explained here.

The “Security Management” offers the possibility to configure the policy of the following components: Authentication, Authorization, Trusted Timestamping, Non-Repudiation.

The Security Proactive Business offers the possibility to suspend and reactivate the following components: Authentication, Authorization, Trusted Timestamping, Non-Repudiation.

The Identity Management manage the provisioning and the administration of identity (entity) of the following components: Authentication, Authorization.

The Authentication Authorization Storage permits to persist users and security policy in the data base, LDAP or other solution.

The details of the interrelations are described from section 6.2. Security Tool Scenarios.

The introduced components are described as follows:

**Authentication Service**

![Diagram](image)

**Figure 31 Required and provided interfaces of the Authentication Service**

Authentication Service provides interfaces to authenticate an actor. In other words the Authentication Service enables to check that the actor is who he claims to be.

These interfaces permit also to configure protocols and manage identity (user or agent). They offer a possibility to support the proactive functionalities.
The exposed operations are:

**IP_Authentication Provided Operations**

- authenticate(Credentials): AuthenticationToken

  The method returns an authentication token. This token should be used to authenticate again the actor (agent or entity).

  The credentials can be a couple of login/password or a certificate. It supports, for example, authentication methods such as username/password authentication, or X.509 certificate authentication.

- configureSecurityPolicy(Policy): boolean

  This method permits to configure the authentication protocol of Authentication Service.

  The Protocol is the authentication protocol. If the authentication service uses a basic authentication, this method permits to change to a strong authentication with the certificate.

  It returns true if the operation is successful.

- suspend()

  This method suspends the Authentication Service.

- reactivate()

  This methodreactivates the Authentication Service.

**IP_AuthStorage Provided Operations**

- createUser(Entity): boolean

  This method permits to create an Entity into the authentication service. The store of this entity can be managed with a low component as LDAP.

  It returns true if the operation is successful.

- deleteUser(Entity): boolean

  This method permits to delete an Entity.

  It returns true if the operation is successful.

- updateUser(Entity): boolean

  This method permits to update an existing Entity.

  It returns true if the operation is successful.

- getUser(Entity Identifier): Entity

  This method permits to retrieve the information elements attached to an Entity.

  It returns an Entity if the operation is successfully get. Otherwise, it returns null.
Required Operations

- notifyEvent(Event)
  Notify an event when an event is occurred.
- displayEvent(Event)
  This method is required by this service in order to display an event by a GUI solution.
- getIdentityAttributes(Entity): Entity Properties
  This method is needed by this service in order to provide the identity of an actor.
- storeIdentity(Entity, Entity Properties)
  This method is needed by this service in order to store or update the identity of an actor.
- deleteIdentity(Entity, Entity Properties)
  This method is needed by this service in order to delete the identity of an actor.
- configureSecurityPolicy(Policy): boolean
  This method is needed by this service in order to configure the Protocol of Authentication.
- suspend()
  This method is needed in order to suspend a service.
- reactivate()
  This method is needed in order to reactivate a service.

Authorization Service

```
+notifyEvent(Event)
+displayEvent()
+getIdentityAttributes(Entity): Entity Properties
+storeIdentity(Entity, Entity Properties)
+deleteIdentity(Entity, Entity Properties)
+configureSecurityPolicy(Policy)
+suspend()
+reactivate()
```

```
+isAuthorized(Entity, Resources, Action, Conditions): Decision
+configureSecurityPolicy(Policy): boolean
+suspend()
+reactivate()
```

```
+createAttributes(Entity): boolean
+deleteAttributes(Entity): boolean
+updateAttributes(Entity): boolean
+getAuthorization(): Attributes Properties
```
Authorization Service provides interfaces to validate that an actor is authorized to perform the action it is willing to perform.

These interfaces permit also to configure the control access policy and manage attributes of the access control. They offer a possibility to support the proactive functionalities.

Authorization privileges (set by the Security Management) are expressed by defining the authorized actions on resources for each actor. The definition of roles can help to map permissions (authorized actions on specific resources) to entities/subjects for a homogeneous group of entities by simply linking an entity to a role. Authorization is validated against an authorization server that stores the authorization privileges or access control list.

The exposed operations are:

**IP_Authorization Provided Operations**

- isAuthorized(Entity, Resources, Action, Conditions): Decision (eg Permit/Deny)
  - The authorization is based on the entity (user).
  - What resources are accessible for this entity?
  - What action this entity can do?
  - Condition may be “only if user has role manager”.
- configureSecurityPolicy(Policy): boolean
  - This method permits to configure the Access control Policy of Authorization Service.
  - The Policy is a list of granted access. By default, an entity can’t access to anything.
  - It returns true if the operation is successful.
- suspend()
  - This method suspends the Authentication Service.
- reactivate()
  - This method reactivates the Authentication Service.

**IP_AuthZStorage Provided Operations**

- createAttributes(Entity): boolean
  - This method creates attributes of access control.
  - It returns true if the operation is successful.
• deleteAttributes(Entity): boolean
  This method deletes all attributes of an entity.
  It returns true if the operation is successful.
• updateAttributes(Entity, Attributes): boolean
  This method permits to update an existing Attributes.
  It returns true if the operation is successful.
• getAttributes(Entity): Properties
  This method permits to get attributes of access control of an entity.
  It returns Properties if the operation is successfully get.

Required Operations
• notifyEvent(Event)
  Notify an event when an event is occurred.
• displayEvent(Event)
  This method is required by this service in order to display an event by a GUI solution.
• getIdentityAttributes(Entity): Entity Properties
  This method is needed by this service in order to provide the identity of an actor.
• storeIdentity(Entity, Entity Properties)
  This method is needed by this service in order to store or update the identity of an actor.
• deleteIdentity(Entity, Entity Properties)
  This method is needed by this service in order to delete the identity of an actor.
• configureSecurityPolicy(Policy): boolean
  This method is needed by this service in order to configure the Protocol of Authentication.
• suspend()
  This method is needed in order to suspend a service.
• reactivate()
  This method is needed in order to reactivate a service.

Identity Management
Identity Management provides an interface to store and delete/get the identifiers of the entities in relation with their identities described by other properties.

The exposed operation is:

Provided Operation
- `getIdentityAttributes(Entity): Entity Properties`
  This method permits to get properties of an entity.
- `storeIdentity(Entity, Entity Properties)`
  This method stores and updates the properties of an entity.
- `deleteIdentity(Entity, Entity Properties)`
  This method deletes the properties of an entity.

Required Operations
- `notifyEvent(Event)`
  Notify an event when an event is occurred.
- `createUser(Entity): boolean`
  This method permits to create an Entity into a persistence solution. This action is done by authentication service.
- `deleteUser(Entity): boolean`
  This method permits to delete an Entity from a persistence solution.

These required Operations are often offered by the low level components such as a database, LDAP or other persistence solution.
- updateUser(Entity): boolean
  This method permits to update an existing Entity from a persistence solution.
- getUser(Entity Identifier): Entity
  This method permits to retrieve the information elements attached to an Entity.
- createAttributes(Entity): boolean
  This method creates attributes of access control into a persistence solution. This action is done by authentication service.
- deleteAttributes(Entity): boolean
  This method deletes all attributes of an entity.
- updateAttributes(Entity, Attributes): boolean
  This method permits to update an existing Attributes.
- getAttributes(Entity): Properties
  This method permits to get attributes of access control of an entity.

Security Management

Manage Security interface provides interface to address Security Policy configuration and Security Implementation.

The exposed operations are:

Provided Operations
- configureSecurityPolicy(Policy)
  This method permits to configure the security components (Authorization Service, Authentication Service)

This method manages the following interface required of this solution:

Required Operations

Figure 34 Required and provided interfaces of the Security Management
- notifyEvent(Event)
  Notify an event when an event is occurred.

**Proactive Security Management**

The functionality works at runtime, automatically or simulated. From the security event it receives, the pattern sends documented alerts and “all clear” to the monitoring pattern. In simulation mode, it advises temporary suspension or permanent disability of compromised services, and late occurrence of reactivation of suspended services. In automatic mode, it can suspend service temporarily or permanently and to reactivate a suspended service when the alert is closed.

The exposed operations are:

**Provided Operations**

- suspend(Security Service)
  This function permits to suspend service temporarily or permanently.
- reactivate(Security Service)
  This function permits to reactivate a suspended service when the alert is closed.
- notifyEvent(Event)
  This method permits other security components to notify the security events.
- displayEvent(Event)
  This method permits to display the current event in order to do analysis about the event.

**Required Operations**

- notifyEvent(Event)
  Notify an event when an event is occurred.

*Figure 35 Required and provided interfaces of the Proactive Security Management*

IR_SecurityPB
+notifyEvent(Event)

IP_SecurityPB
+suspend()
+reactivate()
+notifyEvent(Event)
+displayEvent()
Trusted Timestamping

Trusted timestamping is the process of securely keeping track of the creation and modification time of a document. Security here means that no one, not even the owner of the document, should be able to change it once it has been recorded provided that the timestamper's integrity is never compromised. By consequence, this component trusts the timestamp of a document or of a message.

In the ESOA, this component exposes essentially two functionalities which permit to trust a message and to verify the trusted timestamp attached to this message.

In the ESOA, this component exposes essentially two functionalities which permit to trust a message and to verify the trusted timestamp attached to this message.

The exposed operations are:

Provided Operations

- createTimestamp(MessageDigest, DigestAlgorithm): TS Message
  
The method returns a Trusted Timestamping message. This message is trusted by a TSA.
  
The MessageDigest is hashed message. And the DigestAlgorithm is a cryptographic algorithm.
- checkTimestamp(TSACertificate, Timestamped Message, TS Message): boolean
  
  This method permits to check if the message and the timestamp are not modified.
  
The TSACertificate is the certificate of the TSA. Timestamped Message is a hashed message + a timestamp. And the TS Message is a trusted timestamping message.

It returns true if the result of comparison is equal.

Figure 36 Required and provided interfaces of the Trusted Timestamping
• hash(Message, HashAlgorithm): MessageDigest
  This method permits to hash a message with a hash algorithm.
• timestamp(MessageDigest, Time): Timestamped Message
  This method permits to stamp the hour and date on a hashed message (MessageDigest).
  It returns a Timestamped Message.
• getKey(): Key
  This method permits to get a cryptographic key.
• configureSecurityPolicy(Policy): boolean
  This method permits to configure the certificate used by this service (TSA Timestamping Authority).
  It returns true if the operation is successful.
• suspend()
  This method suspends the service.
• reactivate()
  This method reactivates the service.

Required Operations
• notifyEvent(Event)
  Notify an event when an event is occurred.
• displayEvent(Event)
  This method is required by this service in order to display an event by a GUI solution.
• configureSecurityPolicy(Policy): boolean
  This method is needed by this service in order to configure the Protocol of Authentication.
• suspend()
  This method is needed in order to suspend a service.
• reactivate()
  This method is needed in order to reactivate a service.

Non-Repudiation
Non-repudiation is the concept of ensuring that a party in a dispute cannot repudiate, or refute the validity of a statement or contract. Although this concept can be applied to any transmission, including television and radio, by far the most common application is in the verification and trust of signatures.

In the ESOA, the Non-Repudiation component exposes how to guarantee the integrity of a message and how to verify the signature.

The exposed operations are:

**Provided Operations**

- **sign(MessageDigest, PrivateKey, Algorithm): Signed Message**
  
  This method permits to sign a hashed message (MessageDigest) by using a PrivateKey and an Algorithm.
  
  An algorithm is a cryptographic algorithm.
  
  It returns a signed message.

- **isNonRepudiation(SignedMessage, MessageDigest, PublicKey, Algorithm): boolean**

  This method permits to check the non-repudiation of the message.
  
  Signed Message is a message which is signed by the “sign” method.
  
  The PublicKey is linked with the PrivateKey.
  
  The Algorithm permits to this method how to make the signed message to a MessageDigest.
  
  At the end, the MessageDigest (come from the “unsign” method) is compared with the MessageDigest (in parameter of this method).
  
  The result is true if the comparison is equal.
• unsign(Signed Message, PublicKey, Algorithm): Message
This method permits to make the signed message to a clear message.
Signed Message is a message which is signed by the “sign” method.
The PublicKey is linked with the PrivateKey.
The Algorithm permits to this method how to make the signed message to a clear message.
It returns a clear message.

• hash(Message, HashAlgorithm): MessageDigest
This method permits to hash a message with a hash algorithm.
It returns a hashed message called MessageDigest.

• getKey(): Key
This method permits to get a cryptographic key.

• verifyCertificate(Certificate): Status
This method permits to the status of the certificate.
It returns a status of the certificate. It can be “the certificate is expired”.

• configureSecurityPolicy(Policy): boolean
This method permits to configure the certificate used by this service and to set the mode (check or not the validity of the certificate).
It returns true if the operation is successful.

• suspend()
This method suspends the service.

• reactivate()
This method reactivates the service.

Required Operations
• notifyEvent(Event)
Notify an event when an event is occurred.

• displayEvent(Event)
This method is required by this service in order to display an event by a GUI solution.

• configureSecurityPolicy(Policy): boolean
This method is needed by this service in order to configure the Protocol of Authentication.
• suspend()
  This method is needed in order to suspend a service.
• reactivate()
  This method is needed in order to reactivate a service.

2.4.5.2 Security Tool Scenarios
The security tool scenarios show how to use functionalities in general context and the particular context of Enterprise SOA. These scenarios were studied in the atomic form. They are represented under sequence diagrams or other forms (collaboration diagrams …). In essence, each scenario depicts a security component which provides one or more functionalities. These scenarios are designed independently (the most possible) from the technology and not taken into account different possibilities (for example different protocols of authentication such as certificate, retinal scan…). If all these possibilities will be studied, the number of cases will be infinite. The best way is to take case by case when the need of security is identified with the precise protocol. In this case, the identified sequences will be additional to the atomic scenario.

The next sections will represent three main scenarios (the most used in the security context and in the other concerns of ESOA).

Authentication of External Agents scenario

Figure 38 Required and provided interfaces of the Authentication Service

The following sequence diagrams represent scenario of authentication of an External Consumer Agent and an External Provider Agent.
The External Consumer Agent requests the authentication from the Authentication Service. The Security service checks if the existing agent is known from the system.

**Figure 39 Failed Authentication sequence diagram**

The above sequence diagram lightens a sequence of authentication which stresses on the fact that the authentication is failed.

**Figure 40 Successful Authentication sequence diagram**

This sequence shows when the authentication is successfully checked, the communication between the External Agents are granted.

Description of the scenario

The External Consumer Agent requests the authentication from the Authentication Service. The Security service checks if the existing agent is known from the system.
As the same, the External Provider Agent requests the authentication from the Authentication Service. The Security service checks if the existing agent is known from the system.

The following tables sum all possibilities:

<table>
<thead>
<tr>
<th>Ex. Consumer Agent</th>
<th>Ex. Provider Agent</th>
<th>Communication between two agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>unauthenticated</td>
<td>unauthenticated</td>
<td>Both are not possible</td>
</tr>
<tr>
<td>authenticated</td>
<td>unauthenticated</td>
<td>Single way : Consumer to Provider</td>
</tr>
<tr>
<td>unauthenticated</td>
<td>authenticated</td>
<td>Single way : Provider to Consumer</td>
</tr>
<tr>
<td>authenticated</td>
<td>authenticated</td>
<td>Dual way is possible</td>
</tr>
</tbody>
</table>

**Authorization to the External Provider Agent Scenario**

The following sequence diagrams represent an authorization scenario. The Authorization service a high level component of the Security Tool. This component contains many low level components which aren’t quoted in the sequence diagram.

The complete solution can be inspired from the XACML Specification⁶. The next picture describes only a compact building block of ESOA in order to guarantee the authorization.

---

⁶ [http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf](http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf)
The complete set of components of Authorization Service will be provided in the separate template pattern document [referenced to Authorization Pattern].

Figure 42 Failed Authorization sequence diagram
This above sequence diagram lightens a sequence of authorization which stresses on the fact that the authorization is failed.

Figure 43 Successful Authorization sequence diagram
The second sequence shows when the authorization is successfully checked, the access to the External Provider Agent is granted.

Description of the scenario:
The External Consumer Agent requests the authorization to access to the External Provider Agent from the Authorization Service. If the authorization is confirmed, the access to the External Provider Agent is possible.
Configuration of Security Policy of Authentication, Authorization, Non-Repudiation and Trusted Timestaping Scenario

The role permits to configure the security policy which is played by Security Management tool. This tool provides to the ESOA the way to change the protocol authentication or authorization.

![Diagram](image)

**Figure 44 Required and provided interfaces of the Security Management**

The following sequence diagrams represent how to configure the security policy of Authorization and Authentication Services.

![Diagram](image)

**Figure 45 Failed Configuration Security Policy sequence diagram**

This above sequence diagram lightens a sequence of security policy configuration which stresses on the fact that the authentication is failed.
The second sequence shows when the authentication and authorization are successfully checked, the configuration of the authentication and authorization services is possible.

For the moment, there is any specification concerning how to configure automatically the security policy. Each application proposes a web interface in order to configure the security policy.

Description of the scenario:
The External Provider Agent requests the configuration from the Security Management.

The Configuration Management checks the authentication and authorization before granting this action.

The both check is successful, the configuration can be gone to the next step.

If the External Consumer Agent wants to change the Authentication Protocol, it is possible now.

If the External Consumer Agent wants to change the access control policy, it is possible now.

One time the configuration is successfully done, a positive value is returned to the External Consumer Agent.
Figure 47 Configuration Security Policy of Trusted Timestamping and Non-Repudiation sequence diagram

The below sequence diagram shows when the configuration of Trusted Timestamping and Non-Repudiation services is called by an External Provider Agent.

For the Trusted Timestamping, the External Agent can control what certificate is used to encrypt a message.

For the Non-Repudiation, the External Agent can control what certificate is used and what mode of certificate verification is possible. The verification of certificate is optional. The option must be set to use the certificate verification. Moreover, the External Agent has a choice between CRL\(^7\) or OCSP\(^8\) to check the status of the certificate [report to Trusted Timestamping Pattern for more details].

Security Proactive Approach Scenario

The component to prevent a breach is assumed by the Security Proactive Business.

---

\(^7\) CRL Certificate Revocation List

\(^8\) OCSP Online Certificate Status Protocol
Figure 48 Required and provided interfaces of the Security Proactive Business

The following sequence diagrams represent two sequences which describe how the Security Management interacts with the Entity and the security components (Authentication and Authorization Services).

Figure 49 authentication breach sequence diagram

Description of the scenario:

When the Authentication Service is subjected to a security breach, it notifies the Security Management. The Security Management decides (it depends the policy established for the different levels of the events and type of events) to suspend the Authentication Service.

After analysing the breach, correcting, he can reactivate the service.
This event is displayed in order to show to the Administrator.
The Administrator (expert security) analyzes the event. Following his analysis, he decides to reactivate or not the service.

After the analysis, the breach is now secured for example; he can reactivate the Authentication Service.

Figure 50 authorization breach sequence diagram

For the Authorization sequence, the scenario is the same as described previously.

Presently, we know how the Administrator reactivates Trusted Timestamping or Non-Repudiation when they are subjected to breaches.

Identity Management Scenario
The following scenario presents how to provision a user by setting also the access control and modify the name of user or the rules of access control concerning this same user. It contains three sequences which describe:

- User provisioning and the access control (creation)
- Deletion of the user and the attributes of access control
- Get user information

**Description of the scenario:**

The Administrator wants to create a new user with corresponding access rights. He starts to interact with the Identity Management. The Identity Management requests for the authentication and the authorization of this user.
If the authentication and the authorization are successfully granted:

- the user is created inside the Authentication Service,
- the access rights are created inside the Authorization Service.

One time, the Administrator is granted (the authentication and the authorization have already checked. The authentication token and authorization tokens permit to this administrator to use the “delete functionality” without requesting again the Authentication and the Authorization). He wants now to delete the user. The deletion is done. The user is suppressed from the Authentication Service. For the data coherence between different components, he also deletes all attributes (access rights stored in the Authorization Service) concerning this user.
Concerning the getting of user information, he call the “getIdentityAttributes”, The Security Management requests the user information from the Authentication Service and the Authorization Service. The results are returned to the Administrator.

Trust a message and its verification Scenario

The “Trust in a message and its verification scenario” use jointly two following components in order to achieve this solution.

![Figure 55 Required and provided interfaces of the Trusted Timestamping](image1)

![Figure 56 Required and provided interfaces of the Non-Repudiation](image2)

![Figure 57 Trust a message with Trusted Timestamping and Non-Repudiation](image3)
The below sequence diagram expresses in high level how to achieve a trust solution. It shows also a combination of two components. This combination is often used to combine different patterns in order to obtain a wished solution.

This solution can be used to trust a message only or to trust it on time. In fact, the message is signed with the private key of the certificate coming from the Trusted Timestamping. The signature of this message can be decrypted only by the public key of the owner’s certificate. Thus we can trust that this message is well certificated by the right owner. Moreover, the time put on this message permits to guarantee what moment is done this message. This certificated time is used to prove what moment, for example, a citizen declares his tax online.

![Sequence Diagram](image)

**Figure 58 Verification a trusted message with Trusted Timestamping and Non-Repudiation**

The below sequence diagram permits to check if the message or/and the timestamp is modified or not. In the case that the message or the timestamp is modified, the result of this verification alerts that there is a modification. It means that we cannot trust this message and its timestamp.

**2.4.6 Relationships to other patterns**

This section provides the relationships that the Security In ESOA pattern has with the Enterprise SOA pattern. The following picture is just a reminder.
The security in ESOA functionalities are transversal to other patterns of the Enterprise SOA top level pattern. The security tool uses the Monitoring Tool to supervise the Security. The Security Tool reuses as possible other components provided in ESOA.

In this case, this pattern can be used for all other top-level patterns.

### 2.4.7 Application Examples with the security standards on the WS*

<table>
<thead>
<tr>
<th>Service</th>
<th>Standards used</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication</td>
<td>WS-Security: UsernameToken or X509Token</td>
</tr>
<tr>
<td></td>
<td>(certificate)</td>
</tr>
<tr>
<td>authorization</td>
<td>XACML or RBAC</td>
</tr>
<tr>
<td>Trusted timestamping</td>
<td>RFC 3161</td>
</tr>
</tbody>
</table>

### 2.4.7.1 Example of SSO authentication with LemonLDAP

The following picture expresses how to use the current technology in order to achieve the authentication service.

In the following picture, comparing with the authentication scenario:

- The applications are the External Provider Agent.
- The Users are the External Consumer Agent. The agent can be replaced by the entity.
- The Portal contains the Authentication Service.
- The Session DB and the LDAP Server are the low-level components which participate to the SSO Authentication Service.

*Figure 59 Relationship between “Security in ESOA” and “Enterprise SOA” pattern*
Figure 60: Example with a concrete use of Authentication Service

Detail of operations:

- 1 and 2: non-authenticated users (i.e. without valid cookie) are redirected to the portal,
- 3: authentication request (login password validated on LDAP server or other mechanism),
- 4: recovery ok user attributes,
- 5: calculation of the additional attributes asked in the configuration (macros and groups) and storage of the user data in the sessions database,
- 6: cookie generation and redirection to the asked URL,
- 7: interception of the cookie by the agent of protection (Apache module) and recovery of the user data,
8: checking of the access authorization to the asked URL and transmission to the application (real application or reverse-proxy) if granted,
9 and 10: other requests are treated directly with the use data stored in the local cache. Access grant is calculated at each request.

When the authenticated user tries to access to another protected server, only phases 7 to 10 are replayed in all transparency for the user.

2.4.8 References

[LemonLDAP] Supports the SSO Mechanism
http://wiki.lemonldap.ow2.org/xwiki/bin/view/NG/Presentation

http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

2.4.9 Appendix A: Acronyms

LDAP: Lightweight Directory Access Protocol is an application protocol for querying and modifying directory services running over TCP/IP. A directory is a set of objects with attributes organized in a logical and hierarchical manner. A simple example is the telephone directory, which consists of a list of names (of either persons or organizations) organized alphabetically, with each name having an address and phone number associated with it.

RBAC: Role-Based Access Control is an approach to restricting system access to authorized users. It is a newer alternative approach to mandatory access control (MAC) and discretionary access control (DAC). RBAC is sometimes referred to as role-based security.

SSL: Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols that provide security for communications over networks such as the Internet. TLS and SSL encrypt the segments of network connections at the Transport Layer end-to-end.

XACML: eXtensible Access Control Markup Language. It is a declarative access control policy language implemented in XML and a processing model, describing how to interpret the policies.

2.5 Trigger Writeset Extraction Pattern (UPM)

2.5.1 Problem Description

Data replication is the main technique for providing highly available and scalable data services. Different architectural patterns have been proposed for attaining data replication from databases (See the Black-Box, White-Box and Gray-Box DB Replication patterns). In order to extract the data to be replicated into a set of replicas (called writesets), these patterns for database replication require either the use of standard DB interfaces (e.g. Black-Box and White-Box
DB Replication patterns) or the implementation of some minimal interface within the DB (Gray-Box DB Replication pattern).

In this pattern is described how to implement writeset extraction based on one of the standard DB mechanisms called triggers.

2.5.2 Functional Requirements
This section is not applicable to non-functional patterns.

2.5.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>Triggers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>-</td>
</tr>
<tr>
<td>Applicability</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
</tbody>
</table>

2.5.3.1 Rational
Data replication is mainly used to provide high availability to the data and scalability to the applications using those data. The patterns related to writeset extraction have a dramatic impact on the performance of database replication patterns (See Black-Box, Gray-Box and White-Box DB Replication patterns). The resulting system throughput and response times perceived by the clients when the load increases are critical to measure the performance. The applicability of the pattern –that is, the number and strength of assumptions that must be taken into account in order to apply the pattern- must be also considered. Finally, how the writeset extraction pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into account.

In the following text is described how the current pattern for writeset extraction affects the quality attributes identified in the table above.

The main strength of the Trigger Writeset Extraction pattern is that relies on the facilities provided by the trigger mechanism, present in almost all database (either open-source or commercial). This impacts positively the applicability of the pattern, as the writeset extraction can be implemented in almost any database. Moreover, as the database component does not need to be modified, the maintainability is also improved. However, triggers are heavy-weight and when activated frequently as in the case of writeset extraction they consume excessive computing resources, impacting negatively the performance of the system.
2.5.4 Assumptions

The pattern assumes a database component with support for triggers that enables to extract the updates performed by transactions.

The Standard Database interface provides the following operations:

- **beginTx(pa: Provider Agent): TransactionId**
  
  The Database component starts a new transaction triggered by the Provider Agent `pa` returning the transaction identifier that univocally identifies the transaction in the Multi-tier Transactional Service Runtime.

- **commitTx(tid: TransactionId)**
  
  The Database finishes successfully the transaction identified by the corresponding `tid`, persisting the changes in state/data performed during the transaction. The changes in state/data performed during the transaction are accepted and must be visible in the Database for other concurrent transactions.

- **abortTx(tid: TransactionId)**
  
  The Database finishes successfully the transaction identified by the corresponding `tid` discarding the changes in state/data performed during the transaction.

- **read(pa: Provider Agent, tid: TransactionId, q: Query): Data**
  
  It retrieves the Data specified by the Provider Agent `pa` running on the Service Runtime in the context of the transaction identified by the transaction identifier `tid`. The required data is specified by means of a query `q`. It associates the data read in the transaction with TID `tid` with the corresponding Provider Agent.

- **write(pa: Provider Agent, tid: TransactionId, d: Data)**
  
  It persists the Data `d` specified by the Provider Agent `pa` running on the Service Runtime in the context of the transaction identified by the
transaction identifier tid. It associates the data written in the transaction with TID tid with the corresponding Provider Agent.

- setTrigger(name: TriggerName, def: TriggerDefinition, code: TriggerCode)

It allows to define a trigger with name name. A trigger is a procedure that is executed when a condition is fulfilled on performing insert, update or delete operations in the database. The concrete conditions and operations are defined in the def parameter, and the procedure itself is specified by the code in the code parameter.

- unsetTrigger(name: TriggerName)

It allows to remove the trigger with name name from the database.

- inspectLog(range: LogRange): LogData

This operation allows performing log mining. That is, to retrieve a part of the database log specified in the range parameter.

2.5.5 Solution

In order to replicate data contained in databases, it is necessary an interface to extract/inject the desired data from/into the database. Data are collected in the form of writesets. Each writeset contains the modified records in the database in the context of a particular transaction. The transaction identifier (TID) is used to univocally identify a concrete writeset.

With regard to the format of the writesets, they can be obtained either in binary or textual forms. Every approach has its own benefits and drawbacks.

If writesets are collected in binary form, they might be seen as a black-box. That is, their contents might not be analyzed. This means that possible conflicts between different writesets could not be detected. However, this approach allows a faster extraction/injection of the writesets from/into the replicas of the database because it does not require any additional processing.

In the second form, the writeset is collected as textual information (e.g. as SQL sentences). In this case, the contents of the writeset can be analyzed, what is useful to detect conflicts. It is also possible to inject the writesets in database replicas of different vendors, increasing the portability of the solutions. However, the injection of the writesets in the target database replicas requires additional processing, what affects the performance.

Basically, the interface must provide the following two operations:

- getWriteset(tid: TransactionId) : Writeset

It allows an Agent to retrieve from the database the writeset for a particular transaction identified by its tid.

- setWriteset(tid: TransactionId, ws: Writeset)
It allows an **Agent** to apply into the database the writeset for a particular transaction identified by its tid.

The current state-of-the-art databases [Pos, Ora, MSS, MyS] provide some interfaces/mechanisms out of the box – that is, they do not require source code modification in the database - that can be used to perform data extraction/injection of writesets. These are basically the so-called triggers and log mining. Here, it is described the writeset extraction based on the trigger mechanism.

The easiest way of extracting/injecting the writesets of transactions from a database using standard mechanisms is by means of triggers. A trigger is a procedure that is executed upon a condition is fulfilled before/after performing an operation (INSERT, UPDATE, DELETE) in the database.

Therefore, in this case the interface described above is implemented by means of procedures. These procedures must access the database metadata in order to implement the desired functionality, that is, either to obtain or apply a writeset.

The main drawback of this approach is that it is computationally very expensive, so the performance when using this mechanism is affected negatively. For example, each update in a record requires two updates in the database, one for the original data and one for the writeset function that writes the writeset data in a special table.
When triggers are used, the writesets are usually extracted in textual form (that is, by means of SQL sentences). The trade-offs of this fact, are described above.

2.5.6 Relationships to other patterns

This pattern is an alternative to other writeset extraction patterns shown in the figure below. Each pattern has different assumptions and trade-offs in terms of non-functional attributes. The interface and the implementation described here allow the solutions described in those DB replication patterns to extract/inject the writesets from/into the database. At a higher level, the DB replication patterns are used to provide high availability and scalability to service runtimes architected via the Multi-Tier Transactional Service Runtime pattern.

2.5.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.5.7.1 Abstract Components

This pattern is related to the **Database** abstract component of the NEXOF-RA.

2.5.7.2 Concrete Components

Some of the most well-known concrete database system components available these days are:

- Microsoft’s SQL Server [MSS]
- Oracle [Ora]
- IBM’s DB2 [DB2]
- Sybase [Syb]
- PostgreSQL [Pos]
- MySql [MyS]
- ...

...
2.5.8 Relationships to Standards Catalogue

Many of today's databases follow the relational model. The SQL standard is the de facto standard for relational databases. This standard includes support for the trigger mechanism described in this pattern.

2.5.9 Application Examples

One of the main applications of this pattern is to enable efficient database replication. By extracting the writesets it becomes possible to avoid executing the update transactions at all the replicas. That is, with this pattern it is possible to fully execute each update transactions at any of the replicas, extract the resulting writeset, propagate it to the rest of replicas and apply it locally. This process is substantially cheaper than fully executing each update transaction at each replica.

Other applications of this pattern are online incremental backup utilities and geographic distribution of data. Using this pattern, these applications can extract the writesets from the databases and manage them accordingly to their specific functionality (e.g. create database backups without stopping the database system or distribute data near to clients depending on their localization).

2.5.10 References


2.6 Log Mining Writeset Extraction Pattern (UPM)

2.6.1 Problem Description

Data replication is the main technique for providing highly available and scalable data services. Different architectural patterns have been proposed for attaining data replication from databases (See the Black-Box, White-Box and Gray-Box DB Replication patterns). In order to extract the data to be replicated into a set of replicas (called writesets), these patterns for database replication require either the use of standard DB interfaces (e.g. Black-Box and White-Box DB Replication patterns) or the implementation of some minimal interface within the DB (Gray-Box DB Replication pattern).
In this pattern is described how to implement writeset extraction based on one of the standard DB mechanisms called log mining.

2.6.2 Functional Requirements
This section is not applicable to non-functional patterns.

2.6.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>Log Mining</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>-</td>
</tr>
<tr>
<td>Applicability</td>
<td>0</td>
</tr>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
</tbody>
</table>

2.6.3.1 Rational
Data replication is mainly used to provide high availability to the data and scalability to the applications using those data. The patterns related to writeset extraction have a dramatic impact on the performance of database replication patterns (See Black-Box, Gray-Box and White-Box DB Replication patterns). The resulting system throughput and response times perceived by the clients when the load increases are critical to measure the performance. The applicability of the pattern –that is, the number and strength of assumptions that must be taken into account in order to apply the pattern- must be also considered. Finally, how the writeset extraction pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into account.

In the following text is described how the current pattern for writeset extraction affects the quality attributes identified in the table above.

The applicability of this pattern depends on the tools provided by a specific database system in order to perform log mining. As not all databases provide these tools (although many of them do), applicability is considered neutral. When log mining tools are available to implement the writeset extraction, the maintainability of the solution is improved, as no modifications are required to the database. On the side of weaknesses, as it occurs with the Trigger Writeset Extraction pattern the mechanism is expensive in terms of computing resources consumption.
2.6.4 Assumptions

The pattern assumes a database component with log mining facilities that enable to extract the updates performed by transactions.

The Standard Database interface provides the following operations:

- `beginTx(pa: Provider Agent): TransactionId`

  The Database component starts a new transaction triggered by the Provider Agent `pa` returning the transaction identifier that univocally identifies the transaction in the Multi-tier Transactional Service Runtime.

- `commitTx(tid: TransactionId)`

  The Database finishes successfully the transaction identified by the corresponding `tid`, persisting the changes in state/data performed during the transaction. The changes in state/data performed during the transaction are accepted and must be visible in the Database for other concurrent transactions.

- `abortTx(tid: TransactionId)`

  The Database finishes successfully the transaction identified by the corresponding `tid` discarding the changes in state/data performed during the transaction.

- `read(pa: Provider Agent, tid: TransactionId, q: Query): Data`
It retrieves the Data specified by the ProviderAgent $pa$ running on the Service Runtime in the context of the transaction identified by the transaction identifier $tid$. The required data is specified by means of a query $q$. It associates the data read in the transaction with TID $tid$ with the corresponding Provider Agent.

- write($pa$: Provider Agent, $tid$: TransactionId, $d$: Data)

It persists the Data $d$ specified by the Provider Agent $pa$ running on the Service Runtime in the context of the transaction identified by the transaction identifier $tid$. It associates the data written in the transaction with TID $tid$ with the corresponding Provider Agent.

- setTrigger(name: TriggerName, def: TriggerDefinition, code: TriggerCode)

It allows to define a trigger with name $name$. A trigger is a procedure that is executed when a condition is fulfilled on performing insert, update or delete operations in the database. The concrete conditions and operations are defined in the $def$ parameter, and the procedure itself is specified by the code in the $code$ parameter.

- unsetTrigger(name: TriggerName)

It allows to remove the trigger with name $name$ from the database.

- inspectLog(range: LogRange): LogData

This operation allows performing log mining. That is, to retrieve a part of the database log specified in the $range$ parameter.

### 2.6.5 Solution

In order to replicate data contained in databases, it is necessary an interface to extract/inject the desired data from/into the database. Data are collected in the form of writesets. Each writeset contains the modified records in the database in the context of a particular transaction. The transaction identifier (TID) is used to univocally identify a concrete writeset.

With regard to the format of the writesets, they can be obtained either in binary or textual forms. Every approach has its own benefits and drawbacks.

If writesets are collected in binary form, they might be seen as a black-box. That is, their contents might not be analyzed. This means that possible conflicts between different writesets could not be detected. However, this approach allows a faster extraction/injection of the writesets from/into the replicas of the database because it does not require any additional processing.

In the second form, the writeset is collected as textual information (e.g. as SQL sentences). In this case, the contents of the writeset can be analyzed, what is useful to detect conflicts. It is also possible to inject the writesets in database replicas of different vendors, increasing the portability of the solutions. However, the injection of the writesets in the target database replicas requires additional processing, what affects the performance.
Basically, the interface must provide the following two operations:

- `getWriteset(tid: TransactionId) : Writeset`
  It allows an Agent to retrieve from the database the writeset for a particular transaction identified by its tid.

- `setWriteset(tid: TransactionId, ws: Writeset)`
  It allows an Agent to apply into the database the writeset for a particular transaction identified by its tid.

The current state-of-the-art databases [Pos, Ora, MSS, MyS] provide some interfaces/mechanisms out of the box—that is, they do not require source code modification in the database that can be used to perform data extraction/injection of writesets. These are basically the so-called triggers and log mining. Here, it is described the writeset extraction based on the log mining mechanism.

Database logs contain the records of the updates executed in the database in order to perform redo/undo operations to guarantee database consistency in the advent of failures. Using log mining techniques and provided tools, these logs can be analyzed in order to extract information about the operations performed by the different transactions.
The main problem of using log mining is that the log is used to be stored in a separate disk for efficiency reasons, and reading in parallel this file by another process can decrease the performance. Moreover, the log mining is not useful when an eager replication protocol/algorithm must be used. In eager replication, updates are propagated within the transaction boundaries. However, log mining can only examine the records for a concrete transaction when the transaction has already been committed.

Using log mining the writesets are usually extracted in textual form, that is, by means of SQL sentences. The trade-offs of this fact, are described above.

2.6.6 Relationships to other patterns

This pattern is an alternative to other writeset extraction patterns shown in the figure below. Each pattern has different assumptions and trade-offs in terms of non-functional attributes. The interface and the implementation described here allow the solutions described in those DB replication patterns to extract/inject the writesets from/into the database. At a higher level, the DB replication patterns are used to provide high availability and scalability to service runtimes architected via the Multi-Tier Transactional Service Runtime pattern.

2.6.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.6.7.1 Abstract Components

This pattern is related to the Database abstract component of the NEXOF-RA.

2.6.7.2 Concrete Components

Some of the most well-known concrete database system components available these days are:

- Microsoft’s SQL Server [MSS]
2.6.8 Relationships to Standards Catalogue

The most well-known databases used currently in enterprises are relational databases. In relational databases, the SQL standard is the de iure standard. The interfaces for examining the database log are not included in SQL. They are de facto standards provided by some database vendors such as Oracle or IBM. Open-source databases such as PostgreSQL also provide external tools/plugins for examining the log.

2.6.9 Application Examples

One of the main applications of this pattern is to enable efficient database replication. By extracting the writesets it becomes possible to avoid executing the update transactions at all the replicas. That is, with this pattern is possible to fully execute each update transactions at any of the replicas, extract the resulting writeset, propagate it to the rest of replicas and apply it locally. This process is substantially cheaper than fully executing each update transaction at each replica.

Other applications of this pattern are online incremental backup utilities and geographic distribution of data. Using this pattern, these applications can extract the writesets from the databases and manage them accordingly to their specific functionality (e.g. create database backups without stopping the database system or distribute data near to clients depending on their localization).

2.6.10 References


- Oracle [Ora]
- IBM’s DB2 [DB2]
- Sybase [Syb]
- PostgreSQL [Pos]
- MySql [MyS]
- …
2.7 Writeset Extraction Based on Extended DB Interfaces Pattern (UPM)

2.7.1 Problem Description

Data replication is the main technique for providing highly available and scalable data services. Different architectural patterns have been proposed for attaining data replication from databases (See the Black-Box, White-Box and Gray-Box DB Replication patterns). In order to extract the data to be replicated into a set of replicas (called writesets), these patterns for database replication require either the use of standard DB interfaces (e.g. Black-Box and White-Box DB Replication patterns) or the implementation of some minimal interface within the DB (Gray-Box DB Replication pattern).

This writeset extraction pattern describes how to extend the standard DB interface with some minimal functionality to dramatically improve the performance of the extraction.

2.7.2 Functional Requirements

This section is not applicable to non-functional patterns.

2.7.3 Non-Functional Qualities (Quality Attributes)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>+</td>
</tr>
<tr>
<td>Applicability</td>
<td>-</td>
</tr>
<tr>
<td>Maintainability</td>
<td>0</td>
</tr>
</tbody>
</table>

2.7.3.1 Rational

Data replication is mainly used to provide high availability to the data and scalability to the applications using those data. The patterns related to writeset extraction have a dramatic impact on the performance of database replication patterns (See Black-Box, Gray-Box and White-Box DB Replication patterns). The resulting system throughput and response times perceived by the clients when the load increases are critical to measure the performance. The applicability of the pattern—that is, the number and strength of assumptions that must be taken into account in order to apply the pattern—must be also considered. Finally, how the writeset extraction pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into account.
In the following text is described how the current pattern for writeset extraction affects the quality attributes identified in the table above.

The strong point of this pattern is the performance obtained in writeset extraction/injection. By implementing this well-defined interface in the database code, the performance is dramatically improved with regard to the use of standard mechanisms provided by databases such as triggers or log mining. On the other hand, the applicability is restricted only to those open-source databases that provide the source code to perform the required modifications or to databases provided by database vendors that offer facilities to extend the database functionalities on demand. The maintainability is considered as neutral because once the writeset extraction/injection mechanism is implemented, the replication middleware that use it on top of the database is not affected when maintenance changes must be done in the database internals.

2.7.4 Assumptions

This pattern assumes a database component with a standard interface. Since the pattern extends it, it requires to have access to the DB source code.

The Standard Database interface provides the following operations:

- `beginTx(pa: Provider Agent): TransactionId`
  
The Database component starts a new transaction triggered by the Provider Agent `pa` returning the transaction identifier that univocally identifies the transaction in the Multi-tier Transactional Service Runtime.

- `commitTx(tid: TransactionId)`
  
The Database finishes successfully the transaction identified by the corresponding `tid`, persisting the changes in state/data performed during the transaction. The changes in state/data performed during the transaction are accepted and must be visible in the Database for other concurrent transactions.

- `abortTx(tid: TransactionId)`
The Database finishes successfully the transaction identified by the corresponding tid discarding the changes in state/data performed during the transaction.

- **read**(pa: Provider Agent, tid: TransactionId, q: Query): Data
  It retrieves the Data specified by the ProviderAgent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. The required data is specified by means of a query q. It associates the data read in the transaction with TID tid with the corresponding Provider Agent.

- **write**(pa: Provider Agent, tid: TransactionId, d: Data)
  It persists the Data d specified by the Provider Agent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. It associates the data written in the transaction with TID tid with the corresponding Provider Agent.

- **setTrigger**(name: TriggerName, def: TriggerDefinition, code: TriggerCode)
  It allows to define a trigger with name name. A trigger is a procedure that is executed when a condition is fulfilled on performing insert, update or delete operations in the database. The concrete conditions and operations are defined in the def parameter, and the procedure itself is specified by the code in the code parameter.

- **unsetTrigger**(name: TriggerName)
  It allows to remove the trigger with name name from the database.

- **inspectLog**(range: LogRange): LogData
  This operation allows performing log mining. That is, to retrieve a part of the database log specified in the range parameter.

### 2.7.5 Solution

In order to replicate data contained in databases, it is necessary an interface to extract/inject the desired data from/into the database. Data are collected in the form of writesets. Each writeset contains the modified records in the database in the context of a particular transaction. The transaction identifier (TID) is used to univocally identify a concrete writeset.

With regard to the format of the writesets, they can be obtained either in **binary** or **textual** forms. Every approach has its own benefits and drawbacks.

If writesets are collected in binary form, they might be seen as a **black-box**. That is, their contents might not be analyzed. This means that possible conflicts between different writesets could not be detected. However, this approach allows a faster extraction/injection of the writesets from/into the replicas of the database because it does not require any additional processing.
In the second form, the writeset is collected as textual information (e.g. as SQL sentences). In this case, the contents of the writeset can be analyzed, what is useful to detect conflicts. It is also possible to inject the writesets in database replicas of different vendors, increasing the portability of the solutions. However, the injection of the writesets in the target database replicas requires additional processing, what affects the performance.

Basically, the interface must provide the following two operations:

- **getWriteset(tid: TransactionId) : Writeset**
  It allows an **Agent** to retrieve from the database the writeset for a particular transaction identified by its tid.

- **setWriteset(tid: TransactionId, ws: Writeset)**
  It allows an **Agent** to apply into the database the writeset for a particular transaction identified by its tid.

Using standard interfaces/mechanisms (*Triggers* and *Log Mining*), data replication has a high performance penalty and results in solutions with little or null scalability. Therefore, instead of using the standard interfaces for data extraction provided by the current database management systems, the solution proposed here extends the functionality of the database providing extended meta-interfaces [Kem00, KA00, SJPK06] that implement the specific operations that allow to extract/inject writesets from/into databases.

The solution implies to access the source code of the database and implement the required additional modules that perform the required functionality for writeset extraction/injection without influence the normal behaviour of the database.
As it has been stated before, there are two different ways of implementing the extended meta-interfaces depending on whether the interface provides the writesets in binary or textual form. The architect must take this fact into account when deciding the replication protocol to implement because it may impact/restrict the final solution. When the writesets are collected as binary data, their content might not be analyzed. With this approach, the writesets can be propagated among replicas but conflicts could not be detected among concurrent transactions. On the other hand, if the writesets are collected as textual information (e.g. as SQL sentences), the contents of the writeset can be analyzed, what allows for example to access the primary keys of the records that have been updated. This information can be useful for certain replication protocols in order to detect conflicts among concurrent transactions.

2.7.6 Relationships to other patterns

This pattern is an alternative to other writeset extraction patterns shown in the figure below. Each pattern has different assumptions and tradeoffs in terms of non-functional attributes. The interface and the implementation described here allow the solutions described in those DB replication patterns to extract/inject the writesets from/into the database. At a higher level, the DB replication patterns are used to provide high availability and scalability to service runtimes architected via the Multi-Tier Transactional Service Runtime pattern.
2.7.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.7.7.1 Abstract Components

This pattern is related to the Database abstract component of the NEXOF-RA.

2.7.7.2 Concrete Components

Some of the most well-known concrete database system components available these days are:

- Microsoft’s SQL Server [MSS]
- Oracle [Ora]
- IBM’s DB2 [DB2]
- Sybase [Syb]
- PostgreSQL [Pos]
- MySql [MyS]
- ...

2.7.8 Relationships to Standards Catalogue

The most well-known databases used currently in enterprises are relational databases. In relational databases, the SQL standard is the de iure standard. Extended interfaces for writset extraction/injection are not a standard feature in current databases. They require the access to the source code of the database in order to be implemented. So, these interfaces can be either implemented in open-source databases or must be ordered on demand the database vendors.
2.7.9 Application Examples

One of the main applications of this pattern is to enable efficient database replication. By extracting the writesets it becomes possible to avoid executing the update transactions at all the replicas. That is, with this pattern it is possible to fully execute each update transactions at any of the replicas, extract the resulting writeset, propagate it to the rest of replicas and apply it locally. This process is substantially cheaper than fully executing each update transaction at each replica.

Other applications of this pattern are online incremental backup utilities and geographic distribution of data. Using this pattern, these applications can extract the writesets from the databases and manage them accordingly to their specific functionality (e.g. create database backups without stopping the database system or distribute data near to clients depending on their localization).

2.7.10 References

[KA00] Bettina Kemme, Gustavo Alonso: Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. VLDB 2000

2.8 Gray-Box DB Replication Pattern (UPM)

2.8.1 Problem Description

High availability and scalability are common requirements in current service-oriented infrastructures (SOI). In order to provide high availability and scalability for stateful services, it is necessary to provide these two features for the persistent data at the service infrastructure level. The central component that manages persistent data in SOIs is a database (See Multi-Tier Transactional Service Runtime pattern). They store and provide data that are crucial for many services running in the upper tiers of the service-based infrastructure.
Regular databases fail to provide both availability and scalability. The failure of the database component results in the unavailability of the data, and therefore, of all the services that rely on this data. Regarding scalability, a database has a limited capacity in terms of throughput. If this capacity is exceeded, then the quality of service is drastically reduced. If the excessive load continues over time, it results in the failure of the database component or the shedding of the load, what results in the unavailability of the service.

The patterns related to DB replication (White-Box, Gray-Box and Black-Box DB Replication patterns) precisely address these two requirements, availability and scalability at the data tier of SOIs.

This pattern describes the gray-box database replication approach.

### 2.8.2 Functional Requirements

This section is not applicable to non-functional patterns.

### 2.8.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>Gray-Box Database Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Availability</td>
</tr>
<tr>
<td>Applicability</td>
</tr>
<tr>
<td>Maintainability</td>
</tr>
</tbody>
</table>

#### 2.8.3.1 Rational

The main goal of this pattern is to use multiple instances of the database running on different nodes coordinated by a DB replication protocol. By running multiple instances on different nodes, it becomes possible to tolerate node crashes and attain high availability. If the DB replication is performed in a smart way, it also enables to scale. In order to attain scalability, first a read-one write-all (ROWA) strategy is used. ROWA allows executing read only queries at one of the replicas, consuming 1/N th (N being the total number of replicas) of the replicated system capacity. ROWA enables scaling for read-intensive workloads. Additionally, updates can be processed asymmetrically, that is, an update transaction is fully executed only at one of the replicas, and the resulting update tuples (a.k.a. writeset) are extracted and propagated to the other replicas. Asymmetric update processing enables to scale for update workloads.
The applicability of the pattern – that is, the number and strength of assumptions that must be taken into account in order to apply the pattern – must be also considered. Finally, how the database replication pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into account.

Each particular DB replication pattern differs from each other in the tradeoffs among the degree of scalability, applicability and maintainability.

The degree of scalability of this pattern is very high because the writeset extraction is performed very efficiently (See *Writeset Extraction Based on Extended DB Interfaces* pattern). This allows to attain a low ratio between the cost of fully executing a transaction and only applying the updates for it, what enables to scale update workloads.

The applicability of this pattern is considered neutral. This pattern requires implementing the interfaces for writeset extraction in the database (See also *Writeset Extraction Based on Extended DB Interfaces* pattern), so the database source code must be available. This is possible in open-source databases. In commercial databases, these interfaces must be ordered on demand to the specific database vendor. On the other hand, the required extensions are quite localized and small, what makes it quite feasible in most cases.

Maintainability is also considered as neutral. With this pattern, only the DB replication code (e.g. an external middleware) needs to be updated, what eases maintainability. The database kernel is not impacted by the replication functionality. However, the writeset extraction mechanism needs to be introduced/adapted accordingly in order to be coherent with the rest of the DB kernel.

### 2.8.4 Assumptions

The main assumption for this pattern is that data are stored in a database component.

The database should provide an extended interface for efficient writeset extraction.
Additionally, the DB source code should be available since it requires to be modified.

The Standard Database interface provides the following operations:

- `beginTx(pa: Provider Agent): TransactionId`
  
  The Database component starts a new transaction triggered by the Provider Agent `pa` returning the transaction identifier that univocally identifies the transaction in the Multi-tier Transactional Service Runtime.

- `commitTx(tid: TransactionId)`
  
  The Database finishes successfully the transaction identified by the corresponding `tid`, persisting the changes in state/data performed during the transaction. The changes in state/data performed during the transaction are accepted and must be visible in the Database for other concurrent transactions.

- `abortTx(tid: TransactionId)`
  
  The Database finishes successfully the transaction identified by the corresponding `tid` discarding the changes in state/data performed during the transaction.

- `read(pa: Provider Agent, tid: TransactionId, q: Query): Data`
  
  It retrieves the Data specified by the ProviderAgent `pa` running on the Service Runtime in the context of the transaction identified by the
transaction identifier `tid`. The required data is specified by means of a query `q`. It associates the data read in the transaction with TID `tid` with the corresponding Provider Agent.

- `write(pa: Provider Agent, tid: TransactionId, d: Data)`
  It persists the Data `d` specified by the Provider Agent `pa` running on the Service Runtime in the context of the transaction identified by the transaction identifier `tid`. It associates the data written in the transaction with TID `tid` with the corresponding Provider Agent.

- `setTrigger(name: TriggerName, def: TriggerDefinition, code: TriggerCode)`
  It allows to define a trigger with name `name`. A trigger is a procedure that is executed when a condition is fulfilled on performing insert, update or delete operations in the database. The concrete conditions and operations are defined in the `def` parameter, and the procedure itself is specified by the code in the `code` parameter.

- `unsetTrigger(name: TriggerName)`
  It allows to remove the trigger with name `name` from the database.

- `inspectLog(range: LogRange): LogData`
  This operation allows performing log mining. That is, to retrieve a part of the database log specified in the `range` parameter.

The interface for writeset extraction must provide the following two operations:

- `getWriteset(tid: TransactionId): Writeset`
  It allows an `Agent` to retrieve from the database the writeset for a particular transaction identified by its tid.

- `setWriteset(tid: TransactionId, ws: Writeset)`
  It allows an `Agent` to apply into the database the writeset for a particular transaction identified by its tid.

### 2.8.5 Solution

In this pattern, in order to achieve high availability and scalability of the database component, replication is proposed. Database replication lies in running the database at different nodes, coordinating them by means of a replication protocol. There are many different replica control protocols but their categorization is out of the scope of this pattern description.

High availability is attained by means of the redundancy introduced by the data replication. In order to attain scalability, the redundancy of processing transactions should be minimized. If all replicas execute everything, no scalability could be obtained. The proposed pattern increases scalability by combining two techniques: the read-one write-all (ROWA) approach and
asymmetric processing of updates. The former targets the improvement of scalability for read-intensive workloads and the latter the scalability of update workloads.

In the ROWA approach, read-only transactions are executed at any replica locally without any further coordination with other replicas. Updates are applied everywhere.

In the asymmetric processing of updates, an update transaction is executed at one replica, and its updates are propagated and executed at the other replicas either sometime during the transaction or after its commit. As transactions run concurrently at the different replicas accessing the different copies of data items, replica control needs to be coupled with concurrency control protocol in order to provide the desired level of isolation for the concurrent transactions.

Clients, before accessing the database to inject transactional requests, need to discover one of the replicas. This is done transparently by means of a replica discovery mechanism (See Registry-Based Replica Discovery or Multicast-Based Replica Discovery patterns). The basic steps of the DB replication are summarized in Figure 62. The database proxy has been omitted for the sake of clarity. Given a transaction, all its operations are submitted to one of the replicas. Different transactions can be submitted to different replicas. When a database replica receives an operation request, it executes it locally and returns the response. At the end of transaction, if the transaction is read-only, it is committed locally. If the transaction has updated the database, the updates (and depending on the replica control protocol, also the identifiers of the objects read), are propagated to the other replicas. A global serialization order must be determined among concurrent transactions. This can be achieved by generating a total order among all update transactions and then only allow transactions to commit, if they can be serialized according to this total order. The mechanisms that determine how this can be achieved can vary from setting distributed locks to optimistic protocols that use a total order multicast and perform certification of transactions.

---

9 Note that while different replica control protocols might differ in the order and the details of the steps, the described actions can be found in most protocols.
The Gray-Box DB Replication pattern lies in between the White-Box and the Black-Box DB Replication patterns. The pattern combines the good performance of the White-Box DB Replication pattern and the advantages of a middleware-based approach for replicating the data.

The middleware-based approach encapsulates the replication logic in a new middleware component that is interposed between the client and the database and called DB Replication Middleware. Furthermore, concurrency control is now often implemented in both, the DB Replication Middleware and in the database kernel, leading to its own complexity.

In this pattern, the DB Replication Middleware can exploit the performance of an extended writeset extraction/injection interface implementation to perform its tasks more efficiently (See Writeset Extraction Based on Extended DB...
Interfaces pattern). The solution for this pattern is depicted in the following figure.

Additionally, this pattern also allows for an efficient execution of write operations if updates are propagated by simply indicating the physical after-images of updated records and by providing specialized methods to apply these changes on remote replicas. This means that the writesets must be extracted in binary form. Then, only the local replica has to execute the complete write operation while the other replicas perform more efficient apply operations, freeing up additional processing capacity.

The deployment diagram shown in Figure 63 depicts a scenario that includes a replicated database following the gray-box approach. The database proxies are distributed in the corresponding client nodes. In order to discover and connect to the different DB Replication Middleware component, the database proxy depends on a discovery component.
In order to hide database replication to client applications, what is called replication transparency, this pattern relies on a combination of the Transparent Replication Proxy pattern and the patterns related to replica discovery (either the Registry-Based Replica Discovery pattern or the Multicast-Based Replica Discovery pattern). For example, the database connectivity component of the client (e.g., a JDBC/ODBC driver) may implement the Transparent Replication Proxy pattern. In this way, a client interacts with the database through this enhanced database connectivity component. The proxy is able to transparently discover the database replicas and connecting to one of them. From that point on, that replica will execute the transactions locally for that client.

This DB replication pattern together with the Transparent Replication Proxy pattern also handles failures and guarantee exactly-once execution. Upon a failure of the replica processing the transaction for a client, the proxy detects it and performs failover to another replica. We want to outline solutions for two cases; first, when each operation is a single transaction; and second, when a transaction consists of several operations. In both cases, we assume that replica control propagates all updates a transaction has performed to remote replicas in a single message at the end of transaction just before the transaction is committed locally.

Figure 63 Gray-box replication pattern
For single-operation transactions we can distinguish several cases. (1) The failure occurred when the client had no outstanding request. Then, nothing special has to be done. The proxy simply sends the next request to the new replica. (2) If the failure occurred when there was an outstanding request the proxy resubmits the request to the new replica. The new replica might have received the updates before the failure or not. If it has, it should simply return the outcome of the transaction (it either committed or aborted due to conflicts). If it has not received the updates, it will simply execute the transaction as a new transaction. In all cases, each request leads to one transaction executed in the system.

For multi-operation transactions, there exist several solutions. Here it is outlined only one. The proxy keeps track of all operations of the current transaction and the responses it receives. It can discard them when a transaction terminates. In case of a failure, we can distinguish the following cases. (1) The failure occurred when no transaction was active, i.e., the last response the client received was a commit confirmation. Then, the proxy simply submits the operations of the next transaction to the new replica. (2) The failure occurred while a transaction was active, i.e., the client has an outstanding read or update request. In this case, the proxy can resubmit all requests to the new replica, replaying transaction execution. If it receives the same responses as in the original execution, execution can resume where it stopped before the crash. If not, the transaction must be aborted. (3) The failure occurred when the client had submitted a commit request but before the failed replica responded. In this case, the proxy resubmits the commit request to the new replica. The new replica might have received or not received the updates from the failed replica before the failure. In the first case it can simply return the outcome of the transaction (which was either committed or aborted) to the proxy. In the second case it should return an abort as it does not know about this transaction. Then, the proxy will continue as under (2).

2.8.6 Relationships to other patterns

This pattern is an alternative to other database replication patterns shown in the figure below. Each pattern has different assumptions and tradeoffs in terms of non-functional attributes.
As we have outlined in the previous section, this pattern relies on the *Transparent Replication Proxy* pattern applied to the database connectivity component (the DBProxy, for example a JDBC or ODBC driver) to provide replication transparency.

Additionally, the connectivity component implements the connection to the replicas based on any replica discovery pattern (that is, *Registry-Based Replica Discovery* or *Multicast-Based Replica Discovery*).

The pattern relies on writeset extraction/injection interface (exposed by the *Trigger Writeset Extraction, Log Mining Writeset Extraction and Writeset Extraction Based on Extended DB Interfaces* patterns) for getting the data to be replicated.

As it has been pointed out, this pattern relies on the features provided by the *Writeset Extraction Based on Extended DB Interfaces* pattern.

### 2.8.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

#### 2.8.7.1 Abstract Components

This pattern is related to the *Database* abstract component of the NEXOF-RA.

The gray-box database replication approach requires an external middleware component connected to the database that includes the replication logic. This component is called *Database Replication Middleware* in the NEXOF-RA.

#### 2.8.7.2 Concrete Components

Some of the most well-known concrete database system components available these days are:

- Microsoft’s SQL Server [MSS]
- Oracle [Ora]
- IBM’s DB2 [DB2]
- Sybase [Syb]
- PostgreSQL [Pos, PosR]
- MySQL [MyS]
- …

However, the external interfaces required to perform writeset extraction are not provided by database vendors in order to avoid other companies to develop replication solutions on top of their products.

With regard to scientific/technical related work, examples of gray-box database replication components performing replication on top of open-source databases with extended interfaces can be found in [LKPJ05, PJKA05].
2.8.8 Relationships to Standards Catalogue

Regarding to databases, the most well-known and used currently in enterprises are relational databases. In relational databases, the SQL standard is the de iure standard.

2.8.9 Application Examples

DB replication can be used in any scenario where availability and/or scalability of data management is needed.

One of the applications of this pattern is to increase the availability/scalability of service registries.

Another application is in the context of multi-tier architectures in order to overcome the typical DB bottleneck. This scenario is described in detail in the following figure (Figure 64).

The configuration presented below models a gray-box or a black-box replication approach.

Clients connect to a single instance of an application server providing the application services. The services running in this component extract/inject data from/into a replicated database (its bounds are marked with a dotted line). However, for the application server instance the data tier does not appear as replicated. On the one hand, this is achieved by means of the Transparent Replication Proxy pattern implemented for example at the ODBC/JDBC driver. Read-only requests can be redirected to any database replica. Update requests are redirected to a particular database instance. On the other hand, the replication logic of the database instance –that is, the particular protocol or algorithm located in the replication middleware- replicates the changes produced in updated transactions to the other instances in the data tier. Thus, by means of the features provided by both, the driver and the replication middleware, the application server is virtually connected to all the instances of the database, despite it is connected to only one.
Figure 64 Database Replication Example (Gray-Box/Black-Box Approach)

2.8.10 References


2.9 Black-Box DB Replication Pattern (UPM)

2.9.1 Problem Description
High availability and scalability are common requirements in current service-oriented infrastructures (SOI). In order to provide high availability and scalability
for stateful services, it is necessary to provide these two features for the persistent data at the service infrastructure level. The central component that manages persistent data in SOIs is a database (See Multi-Tier Transactional Service Runtime pattern). They store and provide data that are crucial for many services running in the upper tiers of the service-based infrastructure.

Regular databases fail to provide both availability and scalability. The failure of the database component results in the unavailability of the data, and therefore, of all the services that rely on this data. Regarding scalability, a database has a limited capacity in terms of throughput. If this capacity is exceeded, then the quality of service is drastically reduced. If the excessive load continues over time, it results in the failure of the database component or the shedding of the load, what results in the unavailability of the service.

The patterns related to DB replication (White-Box, Gray-Box and Black-Box DB Replication patterns) precisely address these two requirements, availability and scalability at the data tier of SOIs.

This pattern describes the black-box database replication approach.

2.9.2 Functional Requirements

This section is not applicable to non-functional patterns.

2.9.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>Black-Box Database Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Availability</td>
</tr>
<tr>
<td>Applicability</td>
</tr>
<tr>
<td>Maintainability</td>
</tr>
</tbody>
</table>

2.9.3.1 Rational

The main goal of this pattern is to use multiple instances of the database running on different nodes coordinated by a DB replication protocol. By running multiple instances on different nodes, it becomes possible to tolerate node crashes and attain high availability. If the DB replication is performed in a smart way, it also enables to scale. In order to attain scalability, first a read-one write-all (ROWA) strategy is used. ROWA allows executing read only queries at one of the replicas, consuming 1/N th (N being the total number of replicas) of the replicated system capacity. ROWA enables scaling for read-intensive
workloads. Additionally, updates can be processed asymmetrically, that is, an update transaction is fully executed only at one of the replicas, and the resulting update tuples (a.k.a. writeset) are extracted and propagated to the other replicas. Asymmetric update processing enables to scale for update workloads.

The applicability of the pattern— that is, the number and strength of assumptions that must be taken into account in order to apply the pattern— must be also considered. Finally, how the database replication pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into account.

Each particular DB replication pattern differs from each other in the tradeoffs among the degree of scalability, applicability and maintainability.

The degree of scalability of this pattern is modest due to the writeset extraction is performed using standard mechanisms (See Trigger Writeset Extraction and Log Mining Writeset Extraction patterns). These mechanisms are too heavyweight, resulting in saving very low computing capacity when using asymmetric update transaction processing and therefore this pattern allows low scalability for update workloads.

The applicability is high due to it can be applied to any standard database that provides either triggers or log mining and does not require access to the DB source code.

The maintainability is high, since only the DB replication code (e.g. an external middleware) needs to be updated and it is independent of changes in the underlying database system.

2.9.4 Assumptions

The main assumption for this pattern is that data are stored in a database component.

The database should provide either triggers or log mining facilities for performing writeset extraction.
Figure 65 Non-replicated data tier

The Standard Database interface provides the following operations:

- `beginTx(pa: Provider Agent): TransactionId`

  The Database component starts a new transaction triggered by the Provider Agent `pa` returning the transaction identifier that univocally identifies the transaction in the Multi-tier Transactional Service Runtime.
commitTx(tid: TransactionId)

The Database finishes successfully the transaction identified by the corresponding tid, persisting the changes in state/data performed during the transaction. The changes in state/data performed during the transaction are accepted and must be visible in the Database for other concurrent transactions.

abortTx(tid: TransactionId)

The Database finishes successfully the transaction identified by the corresponding tid discarding the changes in state/data performed during the transaction.

read(pa: Provider Agent, tid: TransactionId, q: Query): Data

It retrieves the Data specified by the ProviderAgent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. The required data is specified by means of a query q. It associates the data read in the transaction with TID tid with the corresponding Provider Agent.

write(pa: Provider Agent, tid: TransactionId, d: Data)

It persists the Data d specified by the Provider Agent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. It associates the data written in the transaction with TID tid with the corresponding Provider Agent.

setTrigger(name: TriggerName, def: TriggerDefinition, code: TriggerCode)

It allows to define a trigger with name name. A trigger is a procedure that is executed when a condition is fulfilled on performing insert, update or delete operations in the database. The concrete conditions and operations are defined in the def parameter, and the procedure itself is specified by the code in the code parameter.

unsetTrigger(name: TriggerName)

It allows to remove the trigger with name name from the database.

inspectLog(range: LogRange): LogData

This operation allows performing log mining. That is, to retrieve a part of the database log specified in the range parameter.

The interface for writeset extraction must provide the following two operations:

getWriteset(tid: TransactionId) : Writeset

It allows an Agent to retrieve from the database the writeset for a particular transaction identified by its tid.
- `setWriteset(tid: TransactionId, ws: Writeset)`

  It allows an Agent to apply into the database the writeset for a particular transaction identified by its tid.

### 2.9.5 Solution

In this pattern, in order to achieve high availability and scalability of the database component, replication is proposed. Database replication lies in running the database at different nodes, coordinating them by means of a replication protocol. There are many different replica control protocols but their categorization is out of the scope of this pattern description.

High availability is attained by means of the redundancy introduced by the data replication. In order to attain scalability, the redundancy of processing transactions should be minimized. If all replicas execute everything, no scalability could be obtained. The proposed pattern increases scalability by combining two techniques: the read-one write-all (ROWA) approach and asymmetric processing of updates. The former targets the improvement of scalability for read-intensive workloads and the latter the scalability of update workloads.

In the ROWA approach, read-only transactions are executed at any replica locally without any further coordination with other replicas. Updates are applied everywhere.

In the asymmetric processing of updates, an update transaction is executed at one replica, and its updates are propagated and executed at the other replicas either sometime during the transaction or after its commit. As transactions run concurrently at the different replicas accessing the different copies of data items, replica control needs to be coupled with concurrency control protocol in order to provide the desired level of isolation for the concurrent transactions.

Clients, before accessing the database to inject transactional requests, need to discover one of the replicas. This is done transparently by means of a replica discovery mechanism (See Registry-Based Replica Discovery or Multicast-Based Replica Discovery patterns). The basic steps of the DB replication are summarized in Figure 62\(^\text{10}\). The database proxy has been omitted for the sake of clarity. Given a transaction, all its operations are submitted to one of the replicas. Different transactions can be submitted to different replicas. When a database replica receives an operation request, it executes it locally and returns the response. At the end of transaction, if the transaction is read-only, it is committed locally. If the transaction has updated the database, the updates (and depending on the replica control protocol, also the identifiers of the objects read), are propagated to the other replicas. A global serialization order must be determined among concurrent transactions. This can be achieved by generating a total order among all update transactions and then only allow transactions to commit, if they can be serialized according to this total order. The mechanisms

\(^{10}\) Note that while different replica control protocols might differ in the order and the details of the steps, the described actions can be found in most protocols.
that determine how this can be achieved can vary from setting distributed locks to optimistic protocols that use a total order multicast and perform certification of transactions.

The **Black-Box DB Replication** pattern uses a standard database server without any additional non-standard replication functionality. On top of the standard database, it provides a DB Replication Middleware component. This component encapsulates the replication logic. Furthermore, concurrency control is now often implemented in both, the DB Replication Middleware component and in the database kernel, leading to its own complexity.

Thus, a challenge with this pattern is to capture the update operations and to apply them efficiently at the remote replicas. As a standard database is used, the writeset extraction/injection interface must be implemented with the functionality provided by the standard database. This means that only the **Trigger Writeset Extraction** and the **Log Mining Writeset Extraction** patterns can
be used with this pattern. These interfaces are very heavyweight, what negatively impacts the performance and resulting scalability.

The black-box approach is shown in the following figure.

The deployment diagram shown in Figure 67 depicts a scenario that includes a replicated database following the black-box approach. The database proxies are distributed in the corresponding client nodes. In order to discover and connect to the different DB Replication Middleware component, the database proxy depends on a discovery component.
In order to hide database replication to client applications, what is called replication transparency, this pattern relies on a combination of the Transparent Replication Proxy pattern and the patterns related to replica discovery (either the Registry-Based Replica Discovery pattern or the Multicast-Based Replica Discovery pattern). For example, the database connectivity component of the client (e.g., a JDBC/ODBC driver) may implement the Transparent Replication Proxy pattern. In this way, a client interacts with the database through this enhanced database connectivity component. The proxy is able to transparently discover the database replicas and connecting to one of them. From that point on, that replica will execute the transactions locally for that client.

This DB replication pattern together with the Transparent Replication Proxy pattern also handles failures and guarantee exactly-once execution. Upon a failure of the replica processing the transaction for a client, the proxy detects it and performs failover to another replica. We want to outline solutions for two cases; first, when each operation is a single transaction; and second, when a transaction consists of several operations. In both cases, we assume that replica control propagates all updates a transaction has performed to remote replicas in a single message at the end of transaction just before the transaction is committed locally.
For single-operation transactions we can distinguish several cases. (1) The failure occurred when the client had no outstanding request. Then, nothing special has to be done. The proxy simply sends the next request to the new replica. (2) If the failure occurred when there was an outstanding request the proxy resubmits the request to the new replica. The new replica might have received the updates before the failure or not. If it has, it should simply return the outcome of the transaction (it either committed or aborted due to conflicts). If it has not received the updates, it will simply execute the transaction as a new transaction. In all cases, each request leads to one transaction executed in the system.

For multi-operation transactions, there exist several solutions. Here it is outlined only one. The proxy keeps track of all operations of the current transaction and the responses it receives. It can discard them when a transaction terminates. In case of a failure, we can distinguish the following cases. (1) The failure occurred when no transaction was active, i.e., the last response the client received was a commit confirmation. Then, the proxy simply submits the operations of the next transaction to the new replica. (2) The failure occurred while a transaction was active, i.e., the client has an outstanding read or update request. In this case, the proxy can resubmit all requests to the new replica, replaying transaction execution. If it receives the same responses as in the original execution, execution can resume where it stopped before the crash. If not, the transaction must be aborted. (3) The failure occurred when the client had submitted a commit request but before the failed replica responded. In this case, the proxy resubmits the commit request to the new replica. The new replica might have received or not received the updates from the failed replica before the failure. In the first case it can simply return the outcome of the transaction (which was either committed or aborted) to the proxy. In the second case it should return an abort as it does not know about this transaction. Then, the proxy will continue as under (2).

2.9.6 Relationships to other patterns

This pattern is an alternative to other database replication patterns shown in the figure below. Each pattern has different assumptions and tradeoffs in terms of non-functional attributes.
As we have outlined in the previous section, this pattern relies on the Transparent Replication Proxy pattern applied to the database connectivity component (the DBProxy, for example a JDBC or ODBC driver) to provide replication transparency.

Additionally, the connectivity component implements the connection to the replicas based on any replica discovery pattern (that is, Registry-Based Replica Discovery or Multicast-Based Replica Discovery).

The pattern relies on writeset extraction/injection interface (exposed by the Trigger Writeset Extraction, Log Mining Writeset Extraction and Writeset Extraction Based on Extended DB Interfaces patterns) for getting the data to be replicated. Specifically, as it has been pointed out, this pattern may rely on the features provided by either the Trigger Writeset Extraction pattern or the Log Mining Writeset Extraction pattern.

2.9.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.9.7.1 Abstract Components

<table>
<thead>
<tr>
<th>Name</th>
<th>Description of the main features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Replication Middleware</td>
<td>It refers to an out-of-the box middleware for databases that includes replication logic.</td>
</tr>
<tr>
<td>Database</td>
<td>It is a component that provides support the use of integrated and persistent collection of data records known as databases. It allows different user application programs to easily access the same data.</td>
</tr>
</tbody>
</table>

2.9.7.2 Concrete Components

<table>
<thead>
<tr>
<th>Name</th>
<th>Description of the main features</th>
<th>Related abstract component</th>
<th>Reference to the concrete implementation</th>
</tr>
</thead>
</table>
2.9.8 Relationships to Standards Catalogue

Regarding to databases, the most well-known and used currently in enterprises are relational databases. In relational databases, the SQL standard is the de facto standard.

The SQL standard includes support for triggers. As this pattern has shown, they can be used for performing writeset extraction. In this sense, the interfaces for examining the log are de facto standards provided by some database vendors such as Oracle or IBM. Open-source databases such as PostgreSQL also provide external tools for examining the log.

2.9.9 Application Examples

DB replication can be used in any scenario where availability and/or scalability of data management are needed.

One of the applications of this pattern is to increase the availability/scalability of service registries.

Another application is in the context of multi-tier architectures in order to overcome the typical DB bottleneck. This scenario is described in detail in the following figure (Figure 64).

The configuration presented below models a gray-box or a black-box replication approach.

Clients connect to a single instance of an application server providing the application services. The services running in this component extract/inject data from/into a replicated database (its bounds are marked with a dotted line). However, for the application server instance the data tier does not appear as replicated. On the one hand, this is achieved by means of the Transparent Replication Proxy pattern implemented for example at the ODBC/JDBC driver. Read-only requests can be redirected to any database replica. Update requests are redirected to a particular database instance. On the other hand, the replication logic of the database instance –that is, the particular protocol or

<table>
<thead>
<tr>
<th>MySQL</th>
<th>See Database description</th>
<th>Database</th>
<th><a href="http://mysql.com/?bydis_dis_index=1">http://mysql.com/?bydis_dis_index=1</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>See Database description</td>
<td>Database</td>
<td><a href="http://www.postgresql.org/">http://www.postgresql.org/</a></td>
</tr>
</tbody>
</table>
algorithm located in the replication middleware replicates the changes produced in updated transactions to the other instances in the data tier. Thus, by means of the features provided by both, the driver and the replication middleware, the application server is virtually connected to all the instances of the database, despite it is connected to only one.

![Database replication example](image)

**Figure 68 Database Replication Example (Gray-Box/Black-Box Approach)**

### 2.9.10 References

No references.

### 2.10 White-Box DB Replication Pattern (UPM)

#### 2.10.1 Problem Description

High availability and scalability are common requirements in current service-oriented infrastructures (SOI). In order to provide high availability and scalability for stateful services, it is necessary to provide these two features for the persistent data at the service infrastructure level. The central component that manages persistent data in SOIs is a database (See *Multi-Tier Transactional Service Runtime* pattern). They store and provide data that are crucial for many services running in the upper tiers of the service-based infrastructure.

Regular databases fail to provide both availability and scalability. The failure of the database component results in the unavailability of the data, and therefore, of all the services that rely on this data. Regarding scalability, a database has a limited capacity in terms of throughput. If this capacity is exceeded, then the
quality of service is drastically reduced. If the excessive load continues over time, it results in the failure of the database component or the shedding of the load, what results in the unavailability of the service.

The patterns related to DB replication (White-Box, Gray-Box and Black-Box DB Replication patterns) precisely address these two requirements, availability and scalability at the data tier of SOIs.

This pattern describes the white-box database replication approach.

2.10.2 Functional Requirements
This section is not applicable to non-functional patterns.

2.10.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>White-Box Database Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Availability</td>
</tr>
<tr>
<td>Applicability</td>
</tr>
<tr>
<td>Maintainability</td>
</tr>
</tbody>
</table>

2.10.3.1 Rational
The main goal of this pattern is to use multiple instances of the database running on different nodes coordinated by a DB replication protocol. By running multiple instances on different nodes, it becomes possible to tolerate node crashes and attain high availability. If the DB replication is performed in a smart way, it also enables to scale. In order to attain scalability, first a read-one write-all (ROWA) strategy is used. ROWA allows executing read only queries at one of the replicas, consuming 1/N th (N being the total number of replicas) of the replicated system capacity. ROWA enables scaling for read-intensive workloads. Additionally, updates can be processed asymmetrically, that is, an update transaction is fully executed only at one of the replicas, and the resulting update tuples (a.k.a. writeset) are extracted and propagated to the other replicas. Asymmetric update processing enables to scale for update workloads.

The applicability of the pattern –that is, the number and strength of assumptions that must be taken into account in order to apply the pattern- must be also considered. Finally, how the database replication pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into
Each particular DB replication pattern differs from each other in the tradeoffs among the degree of scalability, applicability and maintainability.

The degree of scalability of this pattern is very high because the writeset extraction is performed very efficiently (See **Writeset Extraction Based on Extended DB Interfaces** pattern). This allows to attain a low ratio between the cost of fully executing a transaction and only applying the updates for it, what enables to scale update workloads.

The applicability is low because, first, it requires the DB source code to be available, and second, it requires to modify large sections of the DB code what is difficult and requires highly skilled engineers. So this pattern can be applied mainly in open-source databases. In commercial databases, these interfaces must be ordered on demand to the specific database vendor.

Maintainability is low because it requires to keep consistent the DB code with respect the DB kernel, what is a very expensive task.

### 2.10.4 Assumptions

The main assumption for this pattern is that data are stored in a database component.

![Figure 69 Non-replicated data tier](image)

Additionally, the DB source code should be available since it requires to be modified.

The **Standard Database interface** provides the following operations:

- `beginTransaction(pa: Provider Agent): TransactionId`
  
  The Database component starts a new transaction triggered by the Provider Agent `pa` returning the transaction identifier that univocally identifies the transaction in the Multi-tier Transactional Service Runtime.

- `commitTransaction(tid: TransactionId)`
The Database finishes successfully the transaction identified by the corresponding tid, persisting the changes in state/data performed during the transaction. The changes in state/data performed during the transaction are accepted and must be visible in the Database for other concurrent transactions.

- `abortTx(tid: TransactionId)`
  The Database finishes successfully the transaction identified by the corresponding tid discarding the changes in state/data performed during the transaction.

- `read(pa: Provider Agent, tid: TransactionId, q: Query): Data`
  It retrieves the Data specified by the ProviderAgent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. The required data is specified by means of a query q. It associates the data read in the transaction with TID tid with the corresponding Provider Agent.

- `write(pa: Provider Agent, tid: TransactionId, d: Data)`
  It persists the Data d specified by the Provider Agent pa running on the Service Runtime in the context of the transaction identified by the transaction identifier tid. It associates the data written in the transaction with TID tid with the corresponding Provider Agent.

- `setTrigger(name: TriggerName, def: TriggerDefinition, code: TriggerCode)`
  It allows to define a trigger with name name. A trigger is a procedure that is executed when a condition is fulfilled on performing insert, update or delete operations in the database. The concrete conditions and operations are defined in the def parameter, and the procedure itself is specified by the code in the code parameter.

- `unsetTrigger(name: TriggerName)`
  It allows to remove the trigger with name name from the database.

- `inspectLog(range: LogRange): LogData`
  This operation allows performing log mining. That is, to retrieve a part of the database log specified in the range parameter.

- `getWriteset(tid: TransactionId) : Writeset`
  It allows an Agent to retrieve from the database the writeset for a particular transaction identified by its tid.

- `setWriteset(tid: TransactionId, ws: Writeset)`
  It allows an Agent to apply into the database the writeset for a particular transaction identified by its tid.
2.10.5 Solution

In this pattern, in order to achieve high availability and scalability of the database component, replication is proposed. Database replication lies in running the database at different nodes, coordinating them by means of a replication protocol. There are many different replica control protocols but their categorization is out of the scope of this pattern description.

High availability is attained by means of the redundancy introduced by the data replication. In order to attain scalability, the redundancy of processing transactions should be minimized. If all replicas execute everything, no scalability could be obtained. The proposed pattern increases scalability by combining two techniques: the read-one write-all (ROWA) approach and asymmetric processing of updates. The former targets the improvement of scalability for read-intensive workloads and the latter the scalability of update workloads.

In the ROWA approach, read-only transactions are executed at any replica locally without any further coordination with other replicas. Updates are applied everywhere.

In the asymmetric processing of updates, an update transaction is executed at one replica, and its updates are propagated and executed at the other replicas either sometime during the transaction or after its commit. As transactions run concurrently at the different replicas accessing the different copies of data items, replica control needs to be coupled with concurrency control protocol in order to provide the desired level of isolation for the concurrent transactions.

Clients, before accessing the database to inject transactional requests, need to discover one of the replicas. This is done transparently by means of a replica discovery mechanism (See Registry-Based Replica Discovery or Multicast-Based Replica Discovery patterns). The basic steps of the DB replication are summarized in Figure 62\(^{11}\). The database proxy has been omitted for the sake of clarity. Given a transaction, all its operations are submitted to one of the replicas. Different transactions can be submitted to different replicas. When a database replica receives an operation request, it executes it locally and returns the response. At the end of transaction, if the transaction is read-only, it is committed locally. If the transaction has updated the database, the updates (and depending on the replica control protocol, also the identifiers of the objects read), are propagated to the other replicas. A global serialization order must be determined among concurrent transactions. This can be achieved by generating a total order among all update transactions and then only allow transactions to commit, if they can be serialized according to this total order. The mechanisms that determine how this can be achieved can vary from setting distributed locks to optimistic protocols that use a total order multicast and perform certification of transactions.

\(^{11}\) Note that while different replica control protocols might differ in the order and the details of the steps, the described actions can be found in most protocols.
The White-Box DB Replication pattern integrates replication into the database kernel and it enables to do so very efficiently, at the cost of applicability and maintainability. The solution is shown in the following figure.
While introducing replication in the database kernel can exploit the existing concurrency control mechanism of the database system, and thus, can be strongly optimized, the strong coupling of database engine and replication logic leads to increased complexity and maintenance costs. Additionally, it is necessarily vendor specific support.

This pattern also allows for an efficient execution of write operations if updates are propagated by simply indicating the physical after-images of updated records and by providing specialized methods to apply these changes on remote replicas (e.g. using the *Writeset Extraction Based on Extended DB Interfaces* pattern for extracting the writesets in binary form). Then, only the local replica has to execute the complete write operation while the other replicas perform more efficient apply operations, freeing up additional processing capacity.

The deployment diagram shown in Figure 71 depicts a scenario that includes a replicated database following the white-box approach. The database proxies are distributed in the corresponding client nodes. In order to discover and connect to the different database replicas, the database proxy depends on a discovery component.
Figure 71 White-box replication pattern

In order to hide database replication to client applications, what is called replication transparency, this pattern relies on a combination of the Transparent Replication Proxy pattern and the patterns related to replica discovery (either the Registry-Based Replica Discovery pattern or the Multicast-Based Replica Discovery pattern). For example, the database connectivity component of the client (e.g., a JDBC/ODBC driver) may implement the Transparent Replication Proxy pattern. In this way, a client interacts with the database through this enhanced database connectivity component. The proxy is able to transparently discover the database replicas and connecting to one of them. From that point on, that replica will execute the transactions locally for that client.

This DB replication pattern together with the Transparent Replication Proxy pattern also handles failures and guarantee exactly-once execution. Upon a failure of the replica processing the transaction for a client, the proxy detects it and performs failover to another replica. We want to outline solutions for two cases; first, when each operation is a single transaction; and second, when a transaction consists of several operations. In both cases, we assume that replica control propagates all updates a transaction has performed to remote replicas in a single message at the end of transaction just before the transaction is committed locally.

For single-operation transactions we can distinguish several cases. (1) The failure occurred when the client had no outstanding request. Then, nothing special has to be done. The proxy simply sends the next request to the new
replica. (2) If the failure occurred when there was an outstanding request the proxy resubmits the request to the new replica. The new replica might have received the updates before the failure or not. If it has, it should simply return the outcome of the transaction (it either committed or aborted due to conflicts). If it has not received the updates, it will simply execute the transaction as a new transaction. In all cases, each request leads to one transaction executed in the system.

For multi-operation transactions, there exist several solutions. Here it is outlined only one. The proxy keeps track of all operations of the current transaction and the responses it receives. It can discard them when a transaction terminates. In case of a failure, we can distinguish the following cases. (1) The failure occurred when no transaction was active, i.e., the last response the client received was a commit confirmation. Then, the proxy simply submits the operations of the next transaction to the new replica. (2) The failure occurred while a transaction was active, i.e., the client has an outstanding read or update request. In this case, the proxy can resubmit all requests to the new replica, replaying transaction execution. If it receives the same responses as in the original execution, execution can resume where it stopped before the crash. If not, the transaction must be aborted. (3) The failure occurred when the client had submitted a commit request but before the failed replica responded. In this case, the proxy resubmits the commit request to the new replica. The new replica might have received or not received the updates from the failed replica before the failure. In the first case it can simply return the outcome of the transaction (which was either committed or aborted) to the proxy. In the second case it should return an abort as it does not know about this transaction. Then, the proxy will continue as under (2).

2.10.6 Relationships to other patterns

This pattern is an alternative to other database replication patterns shown in the figure below. Each pattern has different assumptions and tradeoffs in terms of non-functional attributes.

As we have outlined in the previous section, this pattern relies on the Transparent Replication Proxy pattern applied to the database connectivity
component (the DBProxy, for example a JDBC or ODBC driver) to provide replication transparency.

Additionally, the connectivity component implements the connection to the replicas based on any replica discovery pattern (that is, Registry-Based Replica Discovery or Multicast-Based Replica Discovery).

2.10.7 Relationships to Components Catalogue
The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.10.7.1 Abstract Components

<table>
<thead>
<tr>
<th>Name</th>
<th>Description of the main features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>It is a component that provides support the use of integrated and persistent collection of data records known as databases. It allows different user application programs to easily access the same data.</td>
</tr>
</tbody>
</table>

2.10.7.2 Concrete Components

<table>
<thead>
<tr>
<th>Name</th>
<th>Description of the main features</th>
<th>Related abstract component</th>
<th>Reference to the concrete implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>See Database description</td>
<td>Database</td>
<td><a href="http://mysql.com/?bydis_dis_index=1">http://mysql.com/?bydis_dis_index=1</a></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>See Database description</td>
<td>Database</td>
<td><a href="http://www.postgresql.org/">http://www.postgresql.org/</a></td>
</tr>
<tr>
<td>Sybase</td>
<td>See Database description</td>
<td>Database</td>
<td><a href="http://www.sybase.com/">http://www.sybase.com/</a></td>
</tr>
<tr>
<td>DB2</td>
<td>See Database description</td>
<td>Database</td>
<td><a href="http://www.ibm.com/db2/">http://www.ibm.com/db2/</a></td>
</tr>
</tbody>
</table>
2.10.8 Relationships to Standards Catalogue

Regarding to databases, the most well-known and used currently in enterprises are relational databases. In relational databases, the SQL standard is the de facto standard.

2.10.9 Application Examples

DB replication can be used in any scenario where availability and/or scalability of data management is needed.

One of the applications of this pattern is to increase the availability/scalability of service registries.

Another application is in the context of multi-tier architectures in order to overcome the typical DB bottleneck. This scenario is described in detail in the following figure (Figure 64).

The configuration presented below models a white-box replication approach.

Clients connect to a single instance of an application server providing the application services. The services running in this component extract/inject data from/into a replicated database (bounds are marked with a dotted line). However, for the application server instance the data tier does not appear as replicated. This is achieved by means of the Transparent Replication Proxy pattern implemented for example at the ODBC/JDBC driver. Read-only requests can be redirected to any database replica. Update requests are redirected to a particular database instance. When receiving the requests, the database instances replicate the changes when necessary to the other replicas.

![Figure 72 Database Replication Example (White-Box Approach)](image)

2.10.10 References

No references.
2.11 Vertical Replication Pattern (UPM)

2.11.1 Problem Description

Current state-of-the-art for service-oriented infrastructures supporting highly available and scalable services architected in multiple tiers mainly address the replication of session state at the application server tier (e.g. see the Session Replication pattern) or at the database tier (e.g. see the three patterns for database replication), but not both tiers at the same time.

Many commercial and open-source solutions rely on the Session Replication architectural pattern. This pattern consists in replicating the session state held at the application server tier, sharing a single database to store the persistent state. The main drawback of this solution is that the database becomes a single point of failure and a possible bottleneck for scalability. The other way around, that is, replicating the database and providing a single instance of the application server, transforms the application server in the bottleneck and the single point of failure.

It is possible to architect the system in order to replicate independently the application server tier and the database tier, coordinating both replication mechanisms (e.g. see the Horizontal Replication pattern). However, building such a solution introduces inconsistencies in the advent of failures and dynamic load balancing.

The proposed Vertical Replication pattern goes beyond current state-of-the-art by enabling to provide holistically high available and scalable services architected in multiple tiers, whilst guaranteeing full consistency of the solution.

2.11.2 Functional Requirements

This section is not applicable to non-functional patterns.

2.11.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>Vertical Replication</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>+</td>
</tr>
<tr>
<td>Availability</td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td>0</td>
</tr>
</tbody>
</table>
2.11.3.1 Rational

The main goal of this pattern is to enhance the availability and scalability of multi-tier service platforms.

In order to attain high availability and scalability, it results to a holistic replication approach that replicates all tiers simultaneously. Transactions are processed following the read-one write-all (ROWA) strategy and asymmetric update processing, what reduces redundancy of transaction processing across replicas. ROWA allows executing read only queries at one of the replicas, consuming 1/N th (N being the total number of replicas) of the replicated system capacity. ROWA enables scaling for read-intensive workloads. With asymmetric update processing of transactions, an update transaction is fully executed only at one of the replicas, and the resulting update tuples (a.k.a. writeset) are extracted and propagated to the other replicas. Asymmetric update processing enables to scale for update workloads.

The applicability of the pattern –that is, the number and strength of assumptions that must be taken into account in order to apply the pattern- must be also considered. The approach encapsulates the replication logic in the application server, what results in a high applicability since it is based on a standard database.

Finally, how the pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into account. The maintainability is considered neutral since it does not require maintaining database code, but it requires to maintain the replication code within the application server.

2.11.4 Assumptions

The pattern assumes a Multi-Tier Transactional Service Runtime that will be enhanced with high availability and scalability. The components of the Multi-Tier Transactional Service Runtime pattern\(^{12}\) are shown in Figure 73. The figure shows the two tiers of the internal multi-tier architecture with their respective proxies: the application server (middle tier) and database (data tier) respectively.

\(^{12}\) See the Multi-Tier Transactional Service Runtime pattern for additional information.
It also assumes that application server code is available in order to introduce the replication logic.

2.11.5 Solution

In this pattern, in order to achieve high availability and scalability of a multi-tier architecture, replication is proposed. The proposed solution consists in replicating the application server and the database components as a single unit, encapsulating the replication logic in the application server. This unit of replication is called a vertical replica. A set of vertical replicas conform a cluster. The application server component of the replica drives the replication process, extracting/injecting data from/into its associated database. The architecture of the solution is shown in Figure 74.
Each application server contains a replication logic component. This component allows to achieve three objectives; first, it enables to replicate the possible changes made in the state of the services running at the middle-tier; second, it guarantees the consistency of the replicated data in all the application server replicas; and third, it guarantees the data consistency in both tiers of the vertical replica at the same time (middle-tier and data-tier).

In order to accommodate different technical requirements, the replication logic included in the application server component may implement different replication protocols (e.g. based on primary-backup, update-everywhere, etc.). In all of them, the database is considered a black box without any awareness of replication. Thus, given with the fact that there is only one replica control protocol for the entire system allowing for a simplified coordination, makes this approach look very appealing. In this replication architecture, the number of
application server and database instances is always the same, as always one instance of each type builds a vertical replica (the atomic replication unit). The following deployment diagram shows a possible distribution of the components for a possible solution. The right most vertical replica is deployed in two different nodes, one for the application server and the second for the database. The other replicas are deployed in a single node including both components. The proxies of each component are distributed in their corresponding nodes. In order to discover and connect to the different vertical replicas, the application server proxy depends on a discovery component.

Thus, high availability is attained by means of the redundancy introduced by replicating the vertical replicas. In order to attain scalability, the redundancy of processing transactions should be minimized. If all replicas execute everything, no scalability could be obtained. The proposed pattern increases scalability by combining two techniques: the read-one write-all (ROWA) approach and asymmetric processing of updates. The former targets the improvement of scalability for read-intensive workloads and the latter the scalability of update workloads.

In the ROWA approach, read-only transactions are executed at any replica locally without any further coordination with other replicas. Updates are applied everywhere.

In the asymmetric processing of updates, an update transaction is executed at one replica, and its updates are propagated and executed at the other replicas either sometime during the transaction or after its commit. As transactions run concurrently at the different replicas accessing the different copies of data items, replica control needs to be coupled with concurrency control protocol in order to provide the desired level of isolation for the concurrent transactions.

At first glance, this solution does not seem to handle unbalanced workloads that
are either compute or data-intensive. However, as the replicas are installed on one machine, such an approach provides natural adaptation. If the workload is compute intensive, the application server will take most of the resources of the node, while if the workload turns data intensive, the database will claim the resources. Automatic adaptation can also be introduced.

In this pattern a failure happens whenever a failure in an individual component of the vertical replica occurs (either the application server or database). When a replica fails, a new one can be introduced into the system in order to not degrade the current performance of the cluster. As it has been commented above, if the replication protocol is properly designed, new replicas can be also introduced in order to scale-out the cluster. In order to guarantee consistency when new replicas are added to the cluster, a recovery process must be applied before the new replica can start processing new client requests.

With regard to the replication transparency, upon a failure is detected in a replica, the application server proxies in the client resubmit outstanding requests to the new replica they have discovered. In order to do so, this pattern relies on a combination of the Transparent Replication Proxy pattern and the patterns related to replica discovery (either the Registry-Based Replica Discovery pattern or the Multicast-Based Replica Discovery pattern).

Different to the Session Replication pattern, no multi-tier coordination is needed. Each application server has full control and knowledge about what is happening at its local database. Either it has received an update message and decided on the outcome of the transaction or it has not received it and the transaction has to be considered aborted.

An example of the replication process is outlined in Figure 75 (failure free scenario). The scenario presents two replicas that consist in an application server –that includes a certification-based replication protocol- and a database. The application server and database proxies have been omitted for the sake of clarity. The client connects to one of the application server replicas. From that point on, that replica will be the local replica for the client. The request is executed locally. When the associated transaction is about to commit the certification protocol is executed. At the end of transaction all database records updated by the request plus the changed session state are propagated to the other replicas. Depending on the concurrency control algorithm it is possible that also the identifiers of the read data records need to be propagated. All update messages need to be globally ordered, e.g., by using a total order multicast. With this, all replicas will deliver these messages in the same order. Every replica acknowledges the reception of the update to all other replicas (e.g. using a uniform multicast).
Each replica then executes the certification protocol to decide about the outcome of each transaction according to the global order. If the transaction under certification conflicts with any other concurrent transaction (it has updated a tuple updated by another transaction) that has already committed, then the transaction is aborted. Otherwise, the session state is applied and the database changes are applied before the transaction is committed. At the local replica, once the other replicas have acknowledged the reception of the message, a response is sent to the client.

2.11.6 Relationships to other patterns

This pattern can be applied to the *Multi-Tier Transactional Service Runtime* pattern in order to provide high availability and scalability.

The main alternatives to this pattern are the patterns devoted to the replication of individual tiers, either the application server (See *Session Replication* pattern) or the database tier (See *White-Box, Gray-Box and Black-Box DB
Replication patterns). The Horizontal Replication pattern replicates both tiers independently, ensuring consistency in both tiers. With respect to them, this pattern provides high availability and scalability of both tiers in an easy way, because the replication of the middle tier and the data tier are done at the same time.

For failover purposes, this pattern may use the Transparent Replication Proxy in order to provide replication transparency to the clients.

Moreover, the clients can implement the connection to the replicas based on any replica discovery pattern (that is, Registry-Based Replica Discovery or Multicast-Based Replica Discovery).

Additionally, if the application server must maintain some session state for the duration of each client connection, the Session Replication pattern can be used to replicate this client-related information. This is useful when failures occur and is valuable to keep the session data highly available.

2.11.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.11.7.1 Abstract Components

In many of the frameworks for developing multi-tier applications is crucial the notion of application server. The application server contains the business logic of applications. This notion is has been captured in this pattern by the abstract component Application Server of the NEXOF-RA.

With regard to the data tier, the notion of data storage has been captured in this pattern by the abstract component Database of the NEXOF-RA.

2.11.7.2 Concrete Components

In real architecture implementations, the functionality of current state-of-the-art application server products does not include vertical replication facilities. This support can be added ad-hoc to open-source application server components based on the J2EE/JEE standard (such as JBoss [JBo], JOnAS [Jon], Geronimo [Gen] or Glassfish [Gla]).

With regard to databases, some examples of concrete databases that can be used to implement this pattern are:

- Microsoft’s SQL Server [MSS]
- Oracle [Ora]
- IBM’s DB2 [DB2]
- Sybase [Syb]
- PostgreSQL [Pos]
- MySql [MyS]
Object-oriented databases or ad-hoc databases can also be used in addition to relational databases.

With regard to scientific/technical related work, some examples of the Vertical Replication pattern can be found in [PVPJ06, PPJK07].

2.11.8 Relationships to Standards Catalogue

Nowadays, there are many different standards that currently allow the construction of multi-tier architectures. With regard to the middle-tier, not all the frameworks/standards -such as J2EE/JEE [JEE], .NET [Net], CORBA [Cor] or SAP [SAP]- can be used for implementing this pattern. As this pattern requires to modify the application server code, open-source application server components based on the J2EE/JEE standard are recommended.

Regarding to databases, the most well-known and used currently in enterprises are relational databases. In relational databases, the SQL standard is the de iure standard. Moreover, in the current market, there are standard interfaces to allow application servers to connect with them (e.g. ODBC, JDBC).

2.11.9 Application Examples

Figure 76 shows a vertical replication configuration that uses a JEE application server and a relational database in each replica.

![Vertical replication example](image)

Figure 76 Vertical replication example

The middle tier includes a replicated JEE application server, whilst the data tier includes several instances of the database. The replication and communication facilities are provided only at the application server. Clients connect to the application server components (called EJBs) by means of a proxy. Each application server is in charge of loading/injecting the data in its associated database using the JDBC driver (proxy). The communication among the application server replicas can be done using peer-to-peer communication (either using specific communication protocols or standard ones) or using a multicast service. The communication among the replicas will depend on the
requirements of the services and applications running on top of the replicated infrastructure.

2.11.10 References


2.12 Session Replication Pattern (UPM)

2.12.1 Problem Description

Many of the current applications deployed over multi-tier architectures are computational intensive. When load increases, these applications must able to scale in order to provide the desired quality of service to the final users. Thus, scalability is a desirable feature for this kind of infrastructures at the middle tier. At this level, application servers hold the business components that form the business logic that consumes CPU cycles on client requests.
Application servers also keep track of the temporal information related to a particular client, the so-called session state. The session state is used in conversations between the particular client and the business logic of the application. In a non-replicated multi-tier infrastructure, the session state is lost if the application server fails. This means that when the application server is re-started, all the pending conversations with the clients must be re-created. This re-execution can degrade the quality of service perceived by the clients, so it is better to provide high availability for those sessions at the middle-tier.

An important challenge for applications is that request execution does not only change the state at the application server but might also access the data tier (e.g. a database). Therefore, any solution that aims to provide scalability and availability in such an infrastructure has to be aware of this fact.

The Session Replication pattern deals with this scenario, achieving scalability and high availability for session state stored in the middle tier.

### 2.12.2 Functional Requirements

This section is not applicable to non-functional patterns.

### 2.12.3 Non-Functional Qualities (Quality Attributes)

<table>
<thead>
<tr>
<th>Vertical Replication</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>+</td>
</tr>
<tr>
<td>Availability</td>
<td>+</td>
</tr>
<tr>
<td>Applicability</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability</td>
<td>0</td>
</tr>
</tbody>
</table>

### 2.12.3.1 Rational

The main goal of this pattern is to enhance the availability and scalability of the middle-tier of applications deployed in multi-tier service platforms.

In order to attain high availability, the pattern results to a replication approach that replicates session information across the middle tier in a cluster of application servers that share a common database. Different application servers might serve different clients leading to load-distribution, and thus, potential for scalability.

The applicability of the pattern –that is, the number and strength of assumptions that must be taken into account in order to apply the pattern- must be also considered. The approach encapsulates the replication logic for the session components in the application server, what results in a high applicability since it
is based on a standard database.

Finally, how the pattern affects maintainability (measured as the number of components that is required to modify and maintain to apply the pattern) is another important quality attribute to take into account. The maintainability is considered neutral since it does not require maintaining database code. It requires only maintaining the replication code within the application server.

2.12.4 Assumptions

The pattern assumes a *Multi-Tier Transactional Service Runtime* that will be enhanced with high availability and scalability at the middle-tier. The components of the *Multi-Tier Transactional Service Runtime* pattern\(^\text{13}\) are shown in Figure 73. The figure shows the two tiers of the internal multi-tier architecture with their respective proxies: the application server (middle-tier) and database (data-tier) respectively.

\(^{13}\) See the *Multi-Tier Transactional Service Runtime* pattern for additional information.
It also assumes that application server code is available in order to introduce the replication logic for session information.

2.12.5 Solution

In the Session Replication pattern, in order to achieve high availability and scalability for the middle-tier of a multi-tier architecture, replication is proposed at that tier. The architecture of the solution is shown in Figure 74. A set of application server replicas conform a cluster that share a single standard database instance. Both, the application servers and the databases have their corresponding proxies.

Figure 77 Multi-Tier Architecture
The proposed solution for attaining high availability consists in replicating the session information held at the application server and keep track of the versions of the data at the database. Session information refers to the data/components modified at the application server that are tied to a particular conversation (transactional set of related operations) from a particular client. Application servers drive the replication process for the session information of the clients, extracting/injecting data when required from/into the common database instance.

With regard to scalability, different servers might serve different clients leading to load-distribution, and thus, potential for scalability. The application server maintains some session state for the duration of the client connection, so once the client has contacted a particular application server replica for starting a session (called local replica), the client will be tied up to that replica till the end of the session.
For fault-tolerance purposes, the session state of each client should be replicated on several replicas (at least one). Session replication requires replica control to keep the session copies on the different servers consistent. That is, when client requests lead to a change of the session state the local replica of the client has to propagate the changes to the remote replicas. Concurrency control is not an issue as different clients have disjoint session state. When the local server replica of a client fails, the client will be transferred to another application server replica with a copy of its session state.

Replication transparency in this pattern requires that each request is executed exactly once in the system despite replication and possible failures. The tasks to achieve this are split between the Session Replication pattern and the Transparent Replication Proxy pattern with the use of the Multi-Tier Coordination pattern.

The following figure shows an example of deployment for the components of Session Replication pattern.

![Session Replication Deployment Example](image)

In Figure 79, the proxies of each component in the architecture (that is, application server replicas and the database instance) are distributed in either client nodes or application server nodes. In order to discover and connect to the different application server replicas, the application server proxy implements the Transparent Replication Proxy pattern for the interaction between clients and application server replicas. The application server proxy depends on a
discovery component that discovers through one of the replica discovery patterns (either Registry-Based Replica Discovery or Multicast-Based Replica Discovery) one of the application server replicas that will be its local replica and will execute all its requests. In the remainder of the pattern description, we assume that each client session has only one request that corresponds to a single database transaction (1 client request/1 transaction). Extensions are possible to deal with more complex transaction interaction patterns such as sessions with 1 client request/N transactions, N client request/1 transaction and N client request/M transactions (See [WK05,PPJV06]). In the deployment diagram, session information for the clients connecting to application server node 1 is replicated to node 2. The session information for the clients connecting to node 2 is replicated to node 3 and finally session information in node 3 is replicated to node 1.

The next paragraphs describe here a possible implementation of the Session Replication pattern depicted in Figure 80 and Figure 81. The example describes a scenario in which two application server replicas are connected to a shared database instance. The clients of such system are connected and perform requests to the application server replicas. The application server and database proxies have been omitted for the sake of clarity.

In Figure 80, when the local application server of a client receives a request, it executes the request locally, potentially interacting with the shared database instance (steps 1-10). Just before finishing executing (and committing the database transaction if one was started), the session state plus the response to be sent to the client are propagated to the server replicas that replicate that particular session. Furthermore, if an update transaction is associated with the request execution, the Multi-Tier Coordination pattern is applied in order to be able to keep track of the outcome of this transaction. Only when the remote replicas confirm the reception of these state changes is the database transaction committed and the response returned to the client. The remote replica keeps track of the last two state changes it receives for each session. Note that the tasks of the remote replica on behalf of this session are not compute intensive. Thus, its main processing power can be used to serve its own clients.

However, when a replica fails (step 11), the client proxy will find a new replica that has a copy of the session state through one of the replica discovery patterns (steps 12-13). If there was an outstanding request, the proxy will resubmit the request to the new replica. Several cases can be distinguished:

- The new replica had not received the state changes on the session and the response associated with this request before the failure. In this case, it is impossible that the database transaction had committed, and the new replica can safely execute the request as a new request.
- The new replica has received the state changes. It will use the Multi-Tier Coordination Pattern in order to determine whether the corresponding
database transaction committed (e.g., by checking if a marker exists in the database).

- If the marker is found, it will install the state changes locally and immediately return the response (see this scenario in Figure 80, steps 14-18). This guarantees exactly-once semantic across the application and database tiers.

- If the database transaction had not committed (see Figure 81, steps 13-14), it will not install the state changes but resort to the previous state of the session. Thus, no changes on behalf of this request remain in the system. The new replica will execute the request as if it were a new request (steps 15-22). This also guarantees exactly-once semantic across the application and database tiers.

Figure 80 Session Replication Failover: Marker Found
2.12.6 Relationships to other patterns

This pattern can be applied to the Multi-Tier Transactional Service Runtime pattern in order to provide high availability and scalability.

The main alternatives to this pattern are the patterns devoted to the replication of the database tier (See White-Box, Gray-Box and Black-Box DB Replication patterns), the Vertical Replication pattern that replicates both tiers at the same time or the Horizontal Replication pattern that replicates both tiers independently.
For failover purposes, this pattern may use the Transparency Replication Proxy in order to provide replication transparency to the clients. If exactly once execution of request is also required, the Multi-Tier Coordination pattern is needed.

Moreover, the clients can implement the connection to the replicas based on any replica discovery pattern (that is, Registry-Based Replica Discovery or Multicast-Based Replica Discovery).

2.12.7 Relationships to Components Catalogue

The following sections detail the relationships of the components of this pattern with regard to the NEXOF Reference Architecture.

2.12.7.1 Abstract Components

In many of the frameworks for developing multi-tier applications is crucial the notion of application server. The application server contains the business logic of applications. This notion is has been captured in this pattern by the abstract component Application Server of the NEXOF-RA.

With regard to the data tier, the notion of data storage has been captured in this pattern by the abstract component Database of the NEXOF-RA.

2.12.7.2 Concrete Components

In real architecture implementations, the functionality of some current state-of-the-art application server products includes session replication facilities. For example, JBoss [JBo] and JOnAS [Jon] include this support. This support can be added ad-hoc to any open-source application servers based on the J2EE/JEE standard.

With regard to databases, some examples of concrete databases that can be used to implement this pattern are:

- Microsoft’s SQL Server [MSS]
- Oracle [Ora]
- IBM’s DB2 [DB2]
- Sybase [Syb]
- PostgreSQL [Pos]
- MySql [MyS]
- …

Object-oriented databases or ad-hoc databases can also be used in addition to relational databases.

2.12.8 Relationships to Standards Catalogue

Nowadays, there are many different standards that currently allow the construction of multi-tier architectures. With regard to the middle-tier, not all the frameworks/standards -such as J2EE/JEE [JEE], .NET [Net], CORBA [Cor] or SAP [SAP]- can be used for implementing this pattern. As this pattern requires
modifying the application server code, open-source application server components based on the J2EE/JEE standard are recommended.

Regarding to databases, the most well-known and used currently in enterprises are relational databases. In relational databases, the SQL standard is the de-lure standard. Moreover, in the current market, there are standard interfaces to allow application servers to connect with them (e.g. ODBC, JDBC).

2.12.9 Application Examples

Figure 82 shows a configuration of the Session Replication pattern that uses a JEE based application server in the middle tier and a relational database in the data tier.

![Figure 82 Session Replication Example](image)

The middle tier includes a replicated application server, whilst the data tier includes a shared database. Clients connect to the application server components (called EJBs) by means of a proxy. Each application server is in charge of loading/injecting the data in the database using the JDBC driver (proxy). Replication and communication facilities are provided by the application server and the database replicas respectively. The communication among the application server replicas of the middle tier can be done using peer to peer communication (either using specific communication protocols or standard ones) or a group communication system (GCS). The communication among the replicas will depend on the requirements of the services and applications running on top of the replicated infrastructure.

2.12.10 References

2.13 Trusted Timestamping Pattern (Thales)

2.13.1 Problem Description

Trusted Timestamping is the process to track a document in the time, since its creation, and follow its changes. It is the means guaranteeing that nobody, even the owner, is capable of modifying it, once it has been timestamped, provided that the timestamper's integrity is never compromised. Consequently, this pattern strengthens the trust in the timestamp of a document.

The administrative aspect involves setting up a publicly available, trusted timestamp management infrastructure to collect process and renew timestamps.

The following picture is an interaction between building blocks. This represents at the same time a scenario in order to conceive a complete trusted timestamping solution.
This scenario lightens different abstract components which are participated to build the Trusted Timestamping solution. It also illustrates the design problem and how the pattern solves the problem.

### 2.13.2 Functional Requirements

The following use-case diagram shows the functional security requirements that this pattern meets. This use-case elaborates a set of security functions which can help other patterns of ESOA to manage their security. It depends on the application context; each functionality can be used separately or combined with other functionalities. To know what security is adequate for the system, a complete study of expression of needs must be conducted.
Figure 84 Functionalities provided by the Trusted Timestamping for Security in ESOA

- **createTimestamp**: offers the functionality to create a trusted timestamped message. A trusted timestamped message is achieved by a complex mechanism which will be described later in the solution section.

- **checkTimestamp**: offers the functionality to check if the time of the message or the document is corrupted or not.

- **hash**: offers the functionality to hash a message or a document. The hash can use different mechanisms of hash. Certain mechanisms are not reversible.

- **timestamp**: offers the functionality to put a time and date on document or a message.

- **getKey**: offers the functionality to get the cryptographic key. This functionality is one of provided functionalities. Others can be described in the XKMS\textsuperscript{14}

- **configureSecurityPolicy**: this functionality is provided by the Security In ESOA pattern and was already described [Security In ESOA Pattern].

\textsuperscript{14} XKMS : XML Key Management Specification
• **suspend**: this functionality is provided by the Security In ESOA pattern and was already described [Security In ESOA Pattern].

• **reactivate**: this functionality is provided by the Security In ESOA pattern and was already described [Security In ESOA Pattern].

2.13.3 Non-Functional Qualities (Quality Attributes)

Generally, the CIA triad (confidentiality, integrity, and availability) must be the atomic quality attributes.

For the context of Enterprise SOA, only the quality attributes directly linked with the provided functionalities and the quoted security components are stressed below:

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity</td>
<td>+</td>
</tr>
<tr>
<td>Accountability</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
<tr>
<td>Performance (efficiency)</td>
<td>-</td>
</tr>
</tbody>
</table>

2.13.3.1 Rational

All security functions will decrease the performance of the system. For this reason, an adequate choice between the needs of the security and the performance must be considered each time to implement the security on a system. For example, a set of files can be read by all public users, the SSL\(^{15}\) is not necessary to secure the network between the end user and the server.

**Integrity**: The aim of integrity is to guarantee that a data is not modified either accidentally or maliciously. The integrity can be obtained with different manners. The encryption of a document protects against the unauthorized access. The digital signature guarantees that the document has not been modified since the signature. Trusted time-stamping adds the information of the time of the signature. The Authorization Service controls the access to the external agent (resources, software system…) and assures that the modification on a document is made by the authorized entity. The architectural choice should respect the standard specification such as XACML\(^{16}\) of OASIS. Based on the standard specification permits to gain a large use of the ESOA and adopts the best practices to guarantee only an authorized access to the resource or to the provider agent.

\(^{15}\) SSL Secure Socket Layer
\(^{16}\) XACML: eXtensible Access Control Markup Language
Accountability  ➤ Provides the ability to enforce accountability or imputability, guarantying the transparency of the operations. Several security components enable accountability. The logging of all events in an audit trail is the most direct way. Authentication contributes to give evidence of accountability. In a more sophisticated way, the architectural choice with the TSA\textsuperscript{17} permits to trust the time of a document or a message. This choice should implement the best practices concerning the trusted timestamping which can be found in some standard specifications such as RFC 3161.

Performance  ➤ performance of a security function is acceptable if its implementation does not impact in a significant way the global performance of the infrastructure.

Maintainability  ➤ the overall security architecture is based on the SOA. All security components are seen as a service. By this way, they inherit the benefits of SOA: interoperability, loose-coupling. And the maintenance of the components is eased by decomposing the Security Tool into loosely coupled sub-components. Each sub-component fulfils functionality, so it is possible to modify the implementation of a particular functionality without affecting the rest of the components.

2.13.4 Assumptions

This section describes the context where the proposed pattern is applicable.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure85.png}
\caption{Functionalities required by the Trusted Timestamping}
\end{figure}

The diagram in Figure 2 shows the functionality required by the Trusted Timestamping to the other parts of the system before it makes sense to apply the pattern.

The proposed pattern is applicable in a context where the following functionality is provided:

- **notifyEvent**: this functionality is required to notify events to the external clients that have subscribed their interest though the `subscribeEvent`

\textsuperscript{17} TSA: Timestamping Authority
method. E.g. an organization can be interested in the publication of new Agent Descriptions in the registry of another organization.

- **displayEvent**: this functionality is required by Security In ESOA to display events.
- **configureSecurityPolicy**: this functionality is required by Security In ESOA to configure the Security Policy.
- **suspend**: this functionality is required by Security In ESOA to suspend a security component.
- **reactivate**: this functionality is required by Security In ESOA to reactivate a security component.

### 2.13.5 Solution

The trusted timestamping solution provides a set of functionalities which can be used in the top-level pattern such as the case of ESOA. These functionalities come from the work of WP4, demonstrated in the WP8 and also from the Investigation Teams.

Before detailing scenarios of provided functionalities, the following section describes all main components which participate in the building of this solution.

#### 2.13.5.1 Trusted Timestamping Components

This pattern captures the distinctiveness of the different Software Components managed by the Security In ESOA and provides a specific Security Component for each of them.

The diagram below shows the provided and required interfaces the system exposes to connect to other systems.

![Diagram of Trusted Timestamping Components](image)

**Figure 86 Required and provided interfaces of the Trusted Timestamping**
The following picture depicts the components of the system and their interconnections.

![Diagram of components and interconnections]

**Figure 87 Components and their interrelations of Trusted Timestamping**

The diagram in Figure 30 shows the components and their interrelations that the “Trusted Timestamping” provides for each specific Software Component.

Before describing all components, their interrelations are explained here.

The “Timestamping Authority” uses the following components Hash, Digital Signature and Timestamping to achieve the trusted timestamping solution.

The Digital Signature uses functionalities offered by the Key Management in order to sign a message or a document.

The details of the interrelations are described from section 6.2. Trusted Timestamping Scenarios.

The introduced components are described as follows:

**Timestamping Authority (TSA)**

![Diagram of Timestamping Authority interfaces]

**Figure 88 Required and provided interfaces of the Timestamping Authority**
Timestamping Authority provides interfaces to create a trusted timestamping message and to check the validity of this message.

The exposed operations are:

**Provided Operations**

- `createTimestamp(MessageDigest, DigestAlgorithm): TSMessage`
  
  The method returns a Trusted Timestamping message. This message is trusted by a TSA.
  
  The `MessageDigest` is hashed message. And the `DigestAlgorithm` is a cryptographic algorithm.

- `checkTimestamp(TSACertificate, TimestampedMessage, TSMessage): boolean`
  
  This method permits to check if the message and the timestamp are not modified.
  
  The `TSACertificate` is the certificate of the TSA. `TimestampedMessage` is a hashed message + a timestamp. And the `TSMessage` is a trusted timestamping message.
  
  It returns true if the result of comparison is equal.

**Required Operations**

- `notifyEvent(Event)`

  Notify an event when an event is occurred.

**Hash**

![Figure 89 Required and provided interfaces of the Hash component](image)

The Hash component permits to hash a message with a hash algorithm. This algorithm can be reversible or irreversible.

The exposed operations are:
Provided Operations

- hash(Message, HashAlgorithm): MessageDigest
  This method permits to hash a message with a hash algorithm.
  It returns a hashed message called MessageDigest.

Required Operations

- notifyEvent(Event)
  Notify an event when an event is occurred.

Digital Signature

Figure 90 Required interfaces of the Digital Signature

This component proposes to digitally sign a message and to unsign this same signed message.

The exposed operations are:

Provided Operations

- sign(MessageDigest, PrivateKey, Algorithm): Signed Message
  This method permits to sign a hashed message (MessageDigest) by using a PrivateKey and an Algorithm.
  An algorithm is a cryptographic algorithm.
  It returns a signed message.

- unsign(Signed Message, PublicKey, Algorithm): Message
  This method permits to make the signed message to a clear message.
  Signed Message is a message which is signed by the “sign” method.
  The PublicKey is linked with the PrivateKey.
The Algorithm permits to this method how to make the signed message to a clear message.

It returns a clear message.

- `isNonRepudiation(Signed Message, MessageDigest, PublicKey, Algorithm): boolean`

This method permits to check the non-repudiation of the message.

Signed Message is a message which is signed by the “sign” method.

The PublicKey is linked with the PrivateKey.

The Algorithm permits to this method how to make the signed message to a MessageDigest.

At the end, the MessageDigest (come from the “unsign” method) is compared with the MessageDigest (in parameter of this method).

The result is true if the comparison is equal.

**Required Operations**

- `notifyEvent(Event)`

  Notify an event when an event is occurred.

**Key Management**

![Figure 91 Required and provided interfaces of the Key Management](image)

Key Management provides an interface to manage cryptographic keys in the ESOA. This component should implement the XKMS\(^{18}\) specification.

The exposed operation is:

**Provided Operation**

- `getKey(): Key`

---

\(^{18}\) XKMS: XML Key Management Specification
This method permits to get a cryptographic key.

Required Operations

- `notifyEvent(Event)`
  
  Notify that an event has occurred.

**Timestamping**

![Timestamping Diagram](image)

**Figure 92 Required and provided interfaces of the Timestamping**

Timestamping interface provides interface to put a time on a message.

The exposed operations are:

**Provided Operations**

- `timestamp(MessageDigest, Time): Timestamped Message`
  
  This method permits to stamp the hour and date on a hashed message (MessageDigest).
  
  It returns a Timestamped Message.

This method manages the following interface required of this solution:

**Required Operations**

- `notifyEvent(Event)`
  
  Notify an event when an event is occurred.
2.13.5.2 Trusted Timestamping Scenarios

In this section, two scenarios are presented. The trust of timestamped message scenario depicts how a message becomes a trusted timestamped message. The second scenario shows how to check the integrity of the time of message.

Trust of timestamped message scenario

![Diagram of Trusted Timestamping](image)

Figure 93 Required and provided interfaces of the Trusted Timestamping

The following sequence diagrams represent scenario of a creation of trusted timestamped message.

![Sequence diagram of trusted timestamped message creation](image)

Figure 94 creation of a trusted timestamped message sequence diagram

Description of this creation

The External Consumer Agent hashes the message by using an algorithm of hash.

The result of hashed message (MessageDigest) is injected to the “createTimestamp” method. The Timestamping Authority cannot read the message because of the hash function.

The External Consumer Agent wants to make a trusted timestamped message by addressing it to the Timestamping Authority.
The Timestamping adds the time to the MessageDigest.
The Digital Signature signs the MessageDigest and the Timestamp.
Before signing the data, the Digital Signature gets the cryptographic key from the Key Management.

**Verification of the trusted timestamping of a message Scenario**
The following scenario describes how the trusted timestamped message permits to verify the validity of the time and the integrity of the message.

![Diagram showing the process of trusted timestamping]

**Figure 95 Verification of a trusted timestamping**

**Description of this verification**
The External Consumer Agent hashes the message by using an algorithm of hash.

The hashed message is added with a timestamp. This action is done by the Timestamp component. This hashed message + timestamp is injected to the Hash component. The result is sent to the External Consumer Agent.

The result of hashed message (MessageDigest) is injected to the “checkTimestamp” method.

The External Consumer Agent can verify now if the origin message is modified or not and also if the timestamp is the correct one.

Before doing the “checkTimestamp” method, the Timestamping Authority decrypts the trusted timestamped message (TS Message, see the previous scenario).

The decryption is done thanks to algorithm and to Key Management component which provides the PublicKey. We should verify also the revocation of the certificate of the TSA. This verification is done with the Non-Repudiation pattern.
The decrypted message is compared to MessageDigest. If these two messages are equal, the timestamp is correct and the message is not corrupted.

2.13.6 Relationships to other patterns

This section provides the relationships that the Trusted Timestamping pattern has with the Security In Enterprise SOA pattern and with the top-level pattern.

Figure 96 Relationship between “Trusted Timestamping”, “Security In ESOA” and “Enterprise SOA” pattern

The security In ESOA functionalities are transversal to other patterns of the Enterprise SOA top level pattern.

In this case, this pattern can be used for all other top-level patterns with certain aspects of the solution section. The objective of this pattern is already quoted in the problem description.

2.13.7 Application Examples with the security standards on the WS*

<table>
<thead>
<tr>
<th>Component</th>
<th>Standards used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Timestamping</td>
<td>RFC 3161</td>
</tr>
<tr>
<td>Key Management</td>
<td>XKMS</td>
</tr>
</tbody>
</table>

2.13.8 References

[XKMS Specification] XML Key Management Specification

2.13.9 Appendix A: Acronyms

SSL: Secure Socket Layer
XKMS: XML Key Management Specification
2.14 Non-Repudiation Pattern (Thales)

2.14.1 Problem Description

Non-repudiation is the concept of ensuring that a party in a dispute cannot repudiate, or refute the validity of an information, statement or contract. Although this concept can be applied to any transmission, including television and radio, by far the most common application is in the verification and trust of signatures.

According to traditional legal practice, a signature on a paper contract or memorandum is sometimes challenged by the signatory. Such contesting may take one of two forms: The signatory may claim fraud or forgery, such as “I did not sign that.” Alternately, he/she may accept the signature as authentic but dispute its validity due to coercion, as in the scenario of blackmail or confessions given under torture. In the latter case the digital signature does nothing more.

The legal burden of proof differs depending upon the repudiation reason. In the former scenario the burden of proof typically rests on the party claiming validity, while in the latter it shifts to the signatory claiming lack thereof.

The following picture is an interaction between building blocks. This represents at the same time a scenario in order to conceive a complete non-repudiation solution.

![Abstract Design Patterns](image)

**Figure 97 Non-Repudiation Scenario**

This scenario lightens different abstract components which are participated to build the Non-Repudiation solution. It also illustrates the design problem and how the pattern solves the problem.

2.14.2 Functional Requirements

The following use-case diagram shows the functional requirements related to Security that this pattern meets. This use-case elaborates a set of security functions which can help other patterns of ESOA to manage their security. It
depends on the application context; each functionality can be used separately or combined with other functionalities. To know what security is adequate for the system, a complete study of expression of needs must be conducted.

![Diagram showing functionalities](image)

**Figure 98 Functionalities provided by the Non-Repudiation for Security in ESOA**

- **isNonRepudiation**: offers the functionality to check if the data is corrupted or not.
- **sign**: offers the functionality to digitally sign the data (document or message). The digital signature is a process that permits to crypt the data with the private key and after to attach the certificate to this encrypted message.
- **unsign**: offers the functionality to decrypt digitally the signature of the data (document or message).
- **hash**: offers the functionality to hash a message or a document. The hash can use different mechanisms of hash.
• **getKey**: offers the functionality to get the cryptographic key. This functionality is one of provided functionalities. Others can be described in the XKMS\(^\text{19}\).

• **verifyCertificate**: offers the functionality to get a status of the certificate. The verifyCertificate is often mandatory to check the revocation of the certificate.

• **configureSecurityPolicy**: this functionality is provided by the Security In ESOA pattern and was already described [Security In ESOA Pattern].

• **suspend**: this functionality is provided by the Security In ESOA pattern and was already described [Security In ESOA Pattern].

• **reactivate**: this functionality is provided by the Security In ESOA pattern and was already described [Security In ESOA Pattern].

### 2.14.3 Non-Functional Qualities (Quality Attributes)

Generally, the CIA triad (confidentiality, integrity, and availability) must be the atomic quality attributes.

For the context of Enterprise SOA, only the quality attributes directly linked with the provided functionalities and the quoted security components are stressed below:

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity</td>
<td>+</td>
</tr>
<tr>
<td>Authenticity</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
<tr>
<td>Performance (efficiency)</td>
<td>-</td>
</tr>
</tbody>
</table>

#### 2.14.3.1 Rational

All security functions will decrease the performance of the system. For this reason, an adequate choice between the needs of the security and the performance must be considered each time to implement the security on a system. For example, a set of files can be read by all public users, the SSL\(^\text{20}\) is not necessary to secure the network between the end user and the server.

**Integrity** The aim of integrity is to guarantee that a data is not modified either accidentally or maliciously. The integrity can be obtained with different manners. The encryption of a document protects against to the unauthorized access. The digital signature guarantees that the document has not been modified since the signature. Trusted time-stamping adds the information of the time of the

---

\(^{19}\) XKMS : XML Key Management Specification

\(^{20}\) SSL Secure Socket Layer
The Authorization Service controls the access to the external agent (resources, software system…) and assures that the modification on a document is made by the authorized entity. The architectural choice should respect the standard specification such as XACML\textsuperscript{21} of OASIS. Based on the standard specification permits to gain a large use of the ESOA and adopts the best practices to guarantee only an authorized access to the resource or to the provider agent.

**Authenticity** A data is considered as authentic when it can be used as a legal proof. Digital signature and time stamping are good tools to constitute such legal proofs.

**Performance** performance of a security function is acceptable if its implementation does not impact in a significant way the global performance of the infrastructure.

**Maintainability** the overall security architecture is based on the SOA. All security components are seen as a service. By this way, they inherit the benefits of SOA: interoperability, loose-coupling. And the maintenance of the components is eased by decomposing the Security Tool into loosely coupled sub-components. Each sub-component fulfills functionality, so it is possible to modify the implementation of a particular functionality without affecting the rest of the components.

### 2.14.4 Assumptions

This section describes the context where the proposed pattern is applicable.

![Diagram](image)

**Figure 99 Functionalities required by the Non-Repudiation**

The diagram in Figure 2 shows the functionality required by the Non-Repudiation to the other parts of the system before it makes sense to apply the pattern.

The proposed pattern is applicable in a context where the following functionality is provided:

---

\textsuperscript{21} XACML: eXtensible Access Control Markup Language
The diagram below shows the provided and required interfaces the system managed by the Security in ESOA and provides a specific Security Component. This pattern captures the distinctiveness of the different Software Components managed by the Security in ESOA and provides a specific Security Component for each of them.

Before detailing the scenarios of provided functionalities, the following section describes all main components which participate in the building of this solution.

2.14.5 Solution

The Non-Repudiation solution provides a set of functionalities which can be used in the top-level pattern such as the case of ESOA. These functionalities come from the work of WP4, the security PoC of WP8 and also from the Investigation Teams.

The diagram below shows the provided and required interfaces the system exposes to connect to other systems.

![Diagram showing provided and required interfaces for Non-Repudiation components](image)

**Figure 100 Required and provided interfaces of the Non-Repudiation**

- **notifyEvent**: this functionality is required to notify events to the external clients that have subscribed their interest through the subscribeEvent method. E.g. an organization can be interested in the publication of new Agent Descriptions in the registry of another organization.
- **displayEvent**: this functionality is required by Security In ESOA to display events.
- **configureSecurityPolicy**: this functionality is required by Security In ESOA to configure the Security Policy.
- **suspend**: this functionality is required by Security In ESOA to suspend a security component.
- **reactivate**: this functionality is required by Security In ESOA to reactivate a security component.

2.14.5.1 Non-Repudiation Components

This pattern captures the distinctiveness of the different Software Components managed by the Security in ESOA and provides a specific Security Component for each of them.

The diagram below shows the provided and required interfaces the system exposes to connect to other systems.
The following picture depicts the components of the system and their interconnections.

![Diagram of system components and interconnections]

**Figure 101 Components and their interrelations of Non-Repudiation solution**

The diagram in Figure 30 shows the components and their interrelations that the “Non-Repudiation” provides for each specific Software Component.

Before describing all components, their interrelations are explained here.

The “Digital Signature” uses the following components Hash, Certificate Store and Key Management to achieve the non-repudiation solution.

The Certificate Store uses functionalities offered alternatively by the CRL\(^{22}\) or OCSP\(^ {23}\) in order to verify the status of certificates.

The details of the interrelations are described from section 6.2. Non-Repudiation Scenarios.

The introduced components are described as follows:

**Digital Signature**

---

\(^{22}\) CRL: certificate revocation list
\(^{23}\) OCSP: online certificate status protocol
Digital Signature provides interfaces to digitally sign/unsign the data and also to check the non-repudiation of this data.

The exposed operations are:

**Provided Operations**

- `sign(MessageDigest, PrivateKey, Algorithm): Signed Message`
  
  This method permits to sign a hashed message (MessageDigest) by using a PrivateKey and an Algorithm.

  An algorithm is a cryptographic algorithm.

  It returns a signed message.

- `unsign(Signed Message, PublicKey, Algorithm): Message`
  
  This method permits to make the signed message to a clear message.

  Signed Message is a message which is signed by the “sign” method.

  The PublicKey is linked with the PrivateKey.

  The Algorithm permits to this method how to make the signed message to a clear message.

  It returns a clear message.

- `isNonRepudiation(Signed Message, MessageDigest, PublicKey, Algorithm): boolean`
  
  This method permits to check the non-repudiation of the message.

  Signed Message is a message which is signed by the “sign” method.

  The PublicKey is linked with the PrivateKey.

  The Algorithm permits to this method how to make the signed message to a MessageDigest.

  At the end, the MessageDigest (come from the “unsign” method) is compared with the MessageDigest (in parameter of this method).

  The result is true if the comparison is equal.
**Required Operations**

- `notifyEvent(Event)`
  Notify an event when an event is occurred.

**Hash**

![Diagram of Hash component]

**Figure 103 Required and provided interfaces of the Hash**

The Hash component permits to hash a message with a hash algorithm.

The exposed operations are:

**Provided Operations**

- `hash(Message, HashAlgorithm): MessageDigest`
  This method permits to hash a message with a hash algorithm.
  It returns a hashed message called MessageDigest.

**Required Operations**

- `notifyEvent(Event)`
  Notify an event when an event is occurred.

**Key Management**

![Diagram of Key Management component]

**Figure 104 Required interfaces of the Key Management**
Key Management provides an interface to manage cryptographic keys in the ESOA. This component should implement the XKMS specification.

The exposed operation is:

Provided Operation

- `getKey()`: Key

  This method permits to get a cryptographic key.

Required Operations

- `notifyEvent(Event)`

  Notify an event when an event is occurred.

Certificate Store

![Certificate Store Diagram]

Figure 105 Required and provided interfaces of the Certificate Store

The Certificate Store interface provides firstly interface to verify the status of certificates.

The exposed operations are:

Provided Operations

- `verifyCertificate(Certificate): Status`

  This method permits to the status of the certificate.

  It returns a status of the certificate. It can be “the certificate is expired”.

Required Operations

- `notifyEvent(Event)`

---

24 XKMS: XML Key Management Specification
Notify an event when an event is occurred.

- `getStatus(Certificate): Status`

This method is required to get the status of the certificate.

**Online Certificate Status Protocol (OCSP)**

The Online Certificate Status Protocol (OCSP) is an Internet protocol used for obtaining the revocation status of an X.509 digital certificate. It is described in RFC 2560 and is on the Internet standards track. It was created as an alternative to certificate revocation lists (CRL), specifically addressing certain problems associated with using CRLs in a public key infrastructure (PKI).

In the ESOA context, the OCSP interface provides firstly interface to get the status of certificate.

The exposed operations are:

**Provided Operations**

- `getStatus(Certificate): Status`

  This method permits to get the status of the certificate.

  It returns a status of the certificate. It can be “the certificate is expired”.

**Required Operations**

- `notifyEvent(Event)`

  Notify an event when an event is occurred.

**Certificate Revocation List (CRL)**
In the operation of some cryptosystems, usually public key infrastructures (PKIs), a certificate revocation list (CRL) is a list of certificates (or more specifically, a list of serial numbers for certificates) that have been revoked or are no longer valid, and therefore should not be relied upon.

In the ESOA context, the CRL interface provides firstly interface to get the status of certificate.

The exposed operations are:

Provided Operations

- `getStatus(Certificate): Status`
  
  This method permits to get the status of the certificate.
  
  It returns a status of the certificate. It can be “the certificate is expired”.

Required Operations

- `notifyEvent(Event)`
  
  Notify an event when an event is occurred.

2.14.5.2 Non-Repudiation Scenarios

In this section, two scenarios are presented. The first scenario depicts how the signature can trust a message and guarantee its integrity. The second scenario shows how to verify the signature.

Trust a message and guarantee its integrity, signature scenario
The following sequence diagrams represent scenario of a creation of signature on a message.

![Sequence Diagram]

**Figure 108 Required and provided interfaces of the Non-Repudiation**

**Figure 109 creation of a signature on a message sequence diagram**

Description of this creation

The External Consumer Agent hashes the message by using an algorithm of hash.

The result of hashed message (MessageDigest) is injected to the “sign” method. And Key Management returns the cryptographic key in order that the Digital Signature signs the MessageDigest by encrypting it with the private key and after by attaching a certificate to it.

The Digital Signed Message is returned to the External Consumer Agent.

**Verification of a signature Scenario**

The following scenario describes how to verify the signature of a message.
The status can be returned to the External Consumer Agent for information.

The verification of the certificate revocation status is mandatory for the validity of the non-repudiation verification. For performance optimization, we can allow not to verify for a reasonable period. The following descriptions concern the two sub sequences verification of CRL or OCSP.

The Digital Signature verifies the status of the certificate. This verification is done with the Certificate Store. And this component gets the status of the certificate from the remote service of OCSP or CRL.

The status is returned directly in the parameter of the “unsign” method.

And then, the message digest computed by the External Consumer Agent is compared to the received MessageDigest. If these two message digests are equal, the message is not corrupted and has been signed by the sender. This process permits to guarantee the integrity of the message. The verification of the certificate trusts the authenticity of the message.

**Figure 110 Verification of a signature sequence diagram**

Description of this verification

The External Consumer Agent hashes the message by using an algorithm of hash.

The hashed message (MessageDigest) is injected to the “isNonRepudiation” method in the objective to compare with the signed message (one time, it is decrypted).

The External Consumer Agent can verify now the non-repudiation of the origin message.

The Digital Signature gets the PublicKey from the Key Management and decrypts the signed message by using the “unsign” method.

The verification of the certificate revocation status is mandatory for the validity of the non-repudiation verification. For performance optimization, we can allow not to verify for a reasonable period. The following descriptions concern the two sub sequences verification of CRL or OCSP.

The Digital Signature verifies the status of the certificate. This verification is done with the Certificate Store. And this component gets the status of the certificate from the remote service of OCSP or CRL.

The status is returned directly in the parameter of the “unsign” method.

And then, the message digest computed by the External Consumer Agent is compared to the received MessageDigest. If these two message digests are equal, the message is not corrupted and has been signed by the sender. This process permits to guarantee the integrity of the message. The verification of the certificate trusts the authenticity of the message.
2.14.6 Relationships to other patterns

This section provides the relationships that the Non-Repudiation pattern has with the Security In Enterprise SOA pattern and with the top-level pattern.

![Diagram of relationships]

**Figure 111 Relationship between “Non-Repudiation”, “Security In ESOA” and “Enterprise SOA” pattern**

The Non-Repudiation functionalities are transversal to other patterns of the Enterprise SOA top level pattern.

In this case, this pattern can be used for all other top-level patterns with certain aspects of the solution section. The objective of this pattern is already quoted in the problem description.

2.14.7 Application Examples with the security standards on the WS*

<table>
<thead>
<tr>
<th>Component</th>
<th>Standards used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Management</td>
<td>XKMS</td>
</tr>
<tr>
<td>OCSP</td>
<td>RFC 2560</td>
</tr>
</tbody>
</table>

2.14.8 References

[XKMS Specification] XML Key Management Specification

2.14.9 Appendix A: Acronyms

CRL: Certificate Revocation List
OCSP: Online Certificate Status Protocol
SSL: Secure Socket Layer
XKMS: XML Key Management Specification
3 CONTRIBUTIONS TO PoCs

WP4 has been one of the main contributors to the first stage of PoCs. The first stage of PoC aimed at providing the evaluation of individual or small sets of interrelated patterns that were identified early in the project. These PoCs have as main goal to evaluate the tradeoffs between architectural choices (among patterns solving the same problem) and/or evaluate the non-functional attributes resulting for some architectural choices (specific patterns). The PoCs are presented in two sets. The one devoted to availability and scalability patterns of multi-tier service runtimes and the one devoted to security patterns.

We present both the revised PoC description and the PoC results.

3.1 PoC Descriptions Related to High Availability and Scalability

The following PoCs have been selected and setup based on a proposal coming from WP4 and took over by UPM based on work performed and achieved by WP4 in close cooperation with WP7. [This was reported in D8.1.] The patterns evaluated in each PoC are the following:

- **PoC: Database Replication Interfaces for Highly Available Stateful Services**
  - Trigger Writeset Extraction Pattern
  - Log Mining Writeset Extraction Pattern
  - Writeset Extraction Based on Extended DB Interfaces Pattern

- **PoC: Gray-Box Database Replication Architectural Pattern For Highly Available Stateful Services**
  - Gray-Box Database Replication Pattern

- **PoC: Vertical Replication Architectural Pattern For Scalable and Highly Available Stateful Services**
  - Vertical Replication Pattern

- **PoC: Gray-Box Database Replication Architectural Pattern in WANs**
  - Gray-Box Database Replication Pattern applied in WANs

### 3.1.1 Database Replication Interfaces for Highly Available Stateful Services

<table>
<thead>
<tr>
<th>POC Name</th>
<th>Database Replication Interfaces for Highly Available Stateful Services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What</strong></td>
<td><strong>Universidad Politécnica de Madrid (UPM)</strong></td>
</tr>
<tr>
<td>Owners</td>
<td><strong>In order to provide highly available stateful services, it is necessary to provide high availability at the service infrastructure level for persistent data stored in databases. Data replication is the main technique for providing highly available and scalable data services. Different architectural patterns have been proposed for replicating databases (See the White-Box, Gray-Box and Black-Box Database Replication patterns). Two of them are generic due to they implement database replication as a middleware component, outside the database, and therefore are</strong></td>
</tr>
</tbody>
</table>
Amenable to be used with any database. In order to replicate transactions across nodes, after executing an update transaction locally at a node, the updates produced by it needs to be obtained in order to propagate them to the other replicas and therefore have a consistent state across replicas. The main difference across the architectural choices for replicating databases lies precisely on how transaction updates are obtained from the database. Black-box database replication implies to use the standard interfaces of the database since the database is considered as a black box. On the other hand, gray-box replication, can extend minimally the database interface to provide this functionality more efficiently.

This PoC evaluates the different tradeoffs involved between the different patterns to acquire transaction updates to enable indirectly infer which are the tradeoffs between the middleware-based database replication patterns.

<table>
<thead>
<tr>
<th>ACPs involved</th>
<th>The main architectural patterns validated in this PoC are <em>Trigger Writeset Extraction, Log Mining Writeset Extraction and Writeset Extraction based on Extended Interfaces</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective of the PoC</td>
<td>The main objective of this PoC is to evaluate the tradeoffs in terms of non-functional aspects of the different writeset extraction/injection interfaces for databases. We will evaluate the following quality attributes: performance, applicability and maintainability. The aim is to compare DB replication patterns (See the <em>White-Box, Gray-Box and Black-Box DB Replication</em> patterns) indirectly by means of micro benchmarking their main difference, that lies in the writeset extraction pattern used. With the results obtained, it will be possible to decide the most appropriate pattern to extract/inject writesets with a DB replication pattern.</td>
</tr>
<tr>
<td>Functionalities</td>
<td>The PoC focus on the functionalities that must provide databases in order to extract/inject writesets from/into them. In particular, it examines the differences in using approaches based on <em>Standard Interfaces/Mechanisms</em> provided by current databases (mainly <em>Triggers and Log Mining</em>) with regard to the use of ad-hoc <em>Extended Interfaces</em> that allow the extraction of writesets directly from the database either in binary or textual form.</td>
</tr>
<tr>
<td>Dependencies</td>
<td>The architectural patterns used in this PoC are strongly related to the black-box, white-box and gray-box database replication approaches described in the corresponding patterns. The interfaces described in the patterns evaluated in this PoC allow the solutions described in the different database replication patterns to extract/inject the required writesets from the database, either in binary or textual form.</td>
</tr>
<tr>
<td>Why</td>
<td>As aforementioned in order to decide which database replication pattern should be used, one has to take into consideration which are the different tradeoffs among them in terms of quality attributes. It turns out that most of the quality attributes derived from the way transaction updates are extracted from the database since this strongly affects the resulting quality attributes.</td>
</tr>
<tr>
<td>Architecture Component(s) affected</td>
<td>The main architectural components affected are databases. They are the architectural components that must provide these writeset extraction/injection interfaces.</td>
</tr>
</tbody>
</table>
Alternatives

The different alternatives of extracting data from a database are described in depth in the patterns related to database interfaces for writeset extraction and evaluated in this PoC. There are two main approaches:

- **Standard DB Interfaces (Triggers and Log Mining)** – They are general purpose interfaces provided by current databases that can be exploited to extract/inject the required writesets. These interfaces allow implementing portable solutions due to the fact that the information provided by the interfaces can be analyzed.

- **Extended DB Interfaces** – They are ad-hoc interfaces exhibited specifically to extract/inject writesets from outside the database. The writesets can either be extracted in binary or textual (SQL form).

Relationship with the NEXOF-RA

High availability and scalability are two requirements of today’s SOI. Databases are a key component of service-oriented infrastructures and replicating them necessary for attaining high available and scalable NEXOF-RA compliant architectures.

The service runtime component is many times architected using the Multi-Tier Transactional Service Runtime pattern. A database component is used to provide persistency and data coherence guarantees so called ACID properties. The database replication patterns are used to provide high availability and scalability to service runtimes architected via the multi-tier pattern. The writeset extraction patterns are used by the database replication patterns to extract transaction updates that need to be propagated across replicas.
How

Scenarios for validation

This PoC addresses the evaluation of different architectural alternatives for a specific component of database replication patterns. As such it is not directly linked to any scenario and the evaluation performs specific microbenchmarks. However, it is indirectly linked with any scenario with high availability and/or scalability requirements that might rely on a replicated database such as the e-commerce scenario in D10.1 (S12). So, the evaluation of the different injection/extraction mechanisms is going to be done directly by means of microbenchmarks on different database management systems, both commercial and open-source: Microsoft SQLServer, PostgreSQL and MySQL. The microbenchmarks will inject an increasing number of transactions in configured databases.

Suggested Architecture

A set of clients will inject transactional load on the different database systems compared. So, a client machine will inject the load to a database running in a separate machine. The environment for the tests is a LAN.
### Environment
The hardware used for the POC consists of computers with AMD Athlon dual CPUs (2GHz), with 1 GB of RAM and two 320 GB hard disks. The running operating system is a Fedora Linux distribution.

The writeset extraction/injection interfaces will be evaluated in three database management systems: Microsoft SQLServer, PostgreSQL and MySQL.

The binary writeset capture is implemented in Postgres.

The textual writeset capture is implemented in MySQL.

Triggers and Log Mining are implemented in MS SQL Server.

### Estimated Effort
12 persons-months. Main efforts: setup the systems (install all server software and configure them), run the evaluations and finally assess them.

### Methods
The method used to measure the effects of data extraction/injection of the different methods proposed consists in comparing the results obtained in a baseline of a particular configuration (without using any writeset extraction/injection method) against the same database system configured with a particular extraction/injection method for an increasing number of transactions.

### Assessment criteria & metrics & way to document
As this PoC aims at evaluating a very technical set of patterns and therefore is far from any business scenario from which extract business requirements, we depart from the technical requirements of quality attributes addressed by the architectural patterns. We use the ATAM utility tree for matching the technical quality attributes (performance, applicability and maintainability) with metrics to be used in the evaluation. The utility tree for the patterns applied in this PoC is presented in the following figure:
For evaluating the quality attributes we have used the following evaluation methodology. We will use specific microbenchmarks to evaluate the throughput and the response time of the different writset extraction methods. Since each pattern can only be applied to one or more databases (due to their different assumptions) and there is no single database supporting the implementation of all the patterns, we need to use different databases (described above). We compare the relative performance of transaction execution with a particular writset extraction pattern with the performance of the same database without writset extraction. This enables us to quantify the overhead of the different patterns and compare them despite having been exercised on different databases. The applicability is measured as the number and strength of assumptions and the maintainability as the number of components we require to modify and maintain.

The generic quality attributes identified for each pattern are (See the documents for each pattern description):

**Trigger Writeset Extraction**
- Performance \( \rightarrow \) -
- Applicability \( \rightarrow \) +
- Maintainability \( \rightarrow \) +

**Log Mining Writeset Extraction**
- Performance \( \rightarrow \) -
- Applicability \( \rightarrow \) 0
- Maintainability \( \rightarrow \) +

**Writset Extraction Based on Extended Interfaces**
- Performance \( \rightarrow \) +
- Applicability \( \rightarrow \) -
- Maintainability \( \rightarrow \) 0

### Further Information

<table>
<thead>
<tr>
<th>POC Name</th>
<th>Gray-Box Database Replication Architectural Pattern For Highly Available Stateful Services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What</strong></td>
<td></td>
</tr>
<tr>
<td>Owners</td>
<td>Universidad Politécnica de Madrid (UPM)</td>
</tr>
<tr>
<td><strong>Description of the PoC</strong></td>
<td>In current service oriented infrastructures, high availability and scalability are common requirements. When considered for stateful services based on databases, they imply to implement some form of database replication. High availability provides 24/7 access to data using clusters of nodes that contain replicas of the data. In this way, if one of the nodes fails, the data is not lost remains available to other requests. Scalability is another key feature required by many service infrastructures. Scalability enables to cope with increasing loads by using a higher number of nodes. Several architectural patterns are available for attaining highly available data: white, gray and black box data replication (See White-Box, Gray-Box and Black-Box Database Replication patterns for further information). The White-Box DB Replication pattern addresses availability by implementing data replication within the database kernel. Black-Box DB Replication pattern takes the database as a black box and using only its standard interface implement replication outside the database. The White-Box approach</td>
</tr>
</tbody>
</table>
might be more performant and scalable but requires access to the database code what in many cases is not possible, and when possible too complex. The Black-Box approach can be applied to any database without access to its code however it might be more costly. The *Gray-Box DB Replication* pattern has been proposed to have the best of black and white architectural patterns. The core idea of the Gray-Box approach is to identify some minimal functionality to be exposed by the database that can enable to provide a solution as performant and scalable as the White-Box architectural pattern but without access to the database code and using a standard database except for the new identified interface.

This PoC aims to demonstrate the validity of architectural choices and architectural patterns described in NEXOF-RA with regard to data replication of data sources, in particular databases.

<table>
<thead>
<tr>
<th>ACPs involved</th>
<th><em>Gray-Box Database Replication</em> architectural pattern.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective of the PoC</td>
<td>The goal of this PoC is to evaluate the scalability of the Gray-Box database replication pattern. The aim is to quantify the scalability in terms of how many times the peak throughput of non-replicated database can be multiplied by using a replicated database with an increasing number of nodes and in this way determine the limits of the approach in terms of throughput scale out. The other quality attributes evaluated for this pattern are applicability and maintainability.</td>
</tr>
<tr>
<td>Functionalities</td>
<td>The PoC focus on the functionality of replicated databases in order to provide high availability and scalability in the data tier.</td>
</tr>
<tr>
<td>Dependencies</td>
<td>The architectural pattern used In this PoC depend on the writeset extraction interfaces described in the <em>Writeset Extraction based on Extended DB Interfaces</em> pattern. More concretely on the extended DB interface writeset extraction patterns. Moreover, for failover purposes, this pattern uses the <em>Transparent Replication Proxy</em> in order to provide replication transparency to the clients. Finally, in order to discover the different replicas the <em>Multicast-Based Replica Discovery</em> pattern is used.</td>
</tr>
<tr>
<td>Why</td>
<td>For an architectural pattern promising scalability is crucial to quantify this scalability. The scalability should measure how many times the peak throughput of the original, non-replicated system is multiplied by the throughput of the replicated system with a particular number of replicas. This quantification is crucial to understand whether the scalability requirements of a particular application can be met by the chosen database replication pattern.</td>
</tr>
<tr>
<td>Architecture Component(s) affected</td>
<td>The main architectural components affected by this PoC are databases. They are the architectural components that contain the critical data that must be replicated in order to provide high availability and scalability to the applications that use these data. Database replication is controlled by a replication middleware in charge of communicating the changes among the different replicas deployed in a LAN and extracting/injecting the required data from the database through the extended database replication interfaces. The middleware is accessed by an application server.</td>
</tr>
</tbody>
</table>
Alternatives

The different alternatives to replicate database systems are described in depth in the *White-Box and Black-Box Database Replication* patterns.

In addition to the solution used in this PoC to extract/inject data from a database, there are other different alternatives described in depth in the patterns *Trigger Writeset Extraction* and *Log Mining Writeset Extraction*.

Relationship with the NEXOF-RA

High availability and scalability are two requirements of today’s SOI. NEXOF-RA addresses them at different levels. In particular, one of the most common forms of the service runtime component is the Multi-Tier Transactional Service Runtime pattern. In this pattern, a database component is used to provide persistency and data coherence guarantees so called ACID properties. The database replication patterns are used to provide high availability and scalability to service runtimes architected via the multi-tier pattern.
### Legend of the pattern language for HA and Scalability

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution Categorization</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Pattern</td>
<td>Multi-Tier Transactional Service Runtime</td>
<td>Specializes/extends requirements</td>
</tr>
<tr>
<td>Replication of Transactional Multi-Tier Service Runtime</td>
<td>can be solved</td>
<td></td>
</tr>
<tr>
<td>DB Replication</td>
<td>can be solved</td>
<td></td>
</tr>
<tr>
<td>Middleware-Based DB Replication</td>
<td>can be solved</td>
<td></td>
</tr>
<tr>
<td>White-Box DB Replication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gray-Box DB Replication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black-Box DB Replication</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### How

<table>
<thead>
<tr>
<th>Scenarios for validation</th>
<th>The scenario for validation is an e-commerce scenario which provides the services of an online bookstore. This scenario is similar to the one that is described in D10.1 (e-Commerce information sharing, S12).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suggested Architecture</td>
<td>In order to evaluate the Gray-Box database replication a cluster is required. The online bookstore application is a web application based on a three-tier architecture. So, each node/replica of the cluster includes a web/application server that includes the presentation and the business logic of the application and a replicated database system containing a full replica of the persistent data used by the application. A client node runs emulated clients that inject request in the replicated infrastructure. The components are interconnected through a LAN.</td>
</tr>
</tbody>
</table>
### Environment

The employed hardware environment consists of nodes with AMD Athlon dual CPUs (2GHz), with 1 GB of RAM and two 320 GB hard disks. The running operating system is a Fedora Linux distribution.

The validation is going to be performed using the industrial benchmark TPC-W for web-based applications that provides an e-commerce application for selling books through the web.

The particular choice of software components is the following:

- Tomcat web server/sevlet container/application server. Contains the TPC-W application services.
- PostgreSQL database server enhanced with binary writeset extraction/injection. Contains the data of the TPC-W application.
- Middle-R replication middleware for replicating PostgreSQL data.
- Ensemble group communication system in order to communicate the Middle-R replicas.

### Estimated Effort

1-2 persons-month. Main efforts: setup the cluster (install all server software, TPC-W application, configure it at all sites), running the evaluation and assessing it.

### Methods

The method used to evaluate the scalability of the **Gray-Box DB Replication** pattern lies in comparing the results obtained in the performance evaluation of a non-replicated system with the performance results obtained in a cluster configured with gray-box data replication approach, for an increasing number of
replicas (nodes).

The assessment criteria are partially based on the ATAM methodology that has been adapted to our specific needs that is to evaluate, quantify and compare the tradeoffs between architectural choices by means of ATAM’s utility trees. The ATAM methodology was originally developed to assist architectural decisions by taking into account early in the design process the quality attributes. Moreover, in this PoC, the business requirements of the utility tree have been obtained from the online bookstore application provided by a benchmark. The utility tree for the Gray-Box DB Replication pattern applied in this PoC is depicted in the following picture:

So, in here, we use the ATAM utility trees to derive from the scenario, business requirements and metrics (on the right side of the figure), technical requirements and quality attributes (on the left side).

Then, we use an evaluation methodology based on industrial practice. We will use a standard benchmark (TCP-W) that allows to evaluate the performance of databases in terms of throughput and response times. We extend the evaluation to quantify the scale-out by running the benchmark with different configurations. We will inject increasing loads to the system till reaching saturation for each configuration. The scale-out shows how many times a particular number of replicas multiply the peak throughput of a non-replicated setup. Also as part of the scalability evaluation the average response time for client requests will be obtained. Availability is measured as the number of single points of failure of the
solution. The applicability is measured as the number and strength of assumptions and the maintainability as the number of components we require to modify and maintain.

The generic quality attributes for the Gray-Box DB Replication pattern are (See the document for the pattern):

- Scalability \( \rightarrow + \)
- Applicability \( \rightarrow + \)
- Applicability \( \rightarrow - \)
- Maintainability \( \rightarrow - \)

Further Information

### 3.1.3 Gray-Box Database Replication Architectural Pattern in WANs

<table>
<thead>
<tr>
<th>POC Name</th>
<th>Gray-Box Database Replication Architectural Pattern in WANs</th>
</tr>
</thead>
<tbody>
<tr>
<td>What</td>
<td></td>
</tr>
<tr>
<td>Owners</td>
<td>Universidad Politécnica de Madrid (UPM)</td>
</tr>
<tr>
<td>Description of the PoC</td>
<td>High availability is a crucial requirement in current service oriented infrastructures. The services provided by the companies must be accessible despite node failures, disconnections from the Internet of full data centres or even catastrophic failures destroying a full data centre. Scalability is another important requirement to cope increasing loads due to an increase in the request volume. Another requirement for applications, typically web applications, with a combination of users from different and distant geographical areas is low latency. Centralized servers result in large latencies for distant clients. Edge computing allows diminishing the latency of the access to static contents, but access to dynamic contents is still centralized, with high latencies. An edge server basically is a server placed close to the clients of a particular region that holds a replica of the data used by the applications (e.g. an edge server can serve the client requests of a particular country). Data replication in Wide Area Networks (WANs) enables to provide low latency to both static and dynamic contents to all clients of such services. This PoC aims at quantifying the improved latency that this pattern can obtained in comparison with current approaches to edge computing based on central servers for dynamic contents.</td>
</tr>
<tr>
<td>ACPs involved</td>
<td>This PoC applies the Gray-Box Database Replication pattern.</td>
</tr>
<tr>
<td>Objective of the PoC</td>
<td>The main objective of this PoC is to evaluate the application of the Gray-Box Database Replication pattern to edge computing to avoid a central sever for static and dynamic contents of applications. That is, storing static and dynamic contents of applications at edge servers and using the database replication pattern over a WAN to keep all edge servers up-to-date and consistent. The emphasis of this PoC is on evaluating an additional quality attribute of the Gray-Box Database Replication pattern: the latency experienced by the clients in a WAN.</td>
</tr>
<tr>
<td>Functionalities</td>
<td>The PoC focuses on the data replication functionality across WANs and its integration with a multi-tier system.</td>
</tr>
<tr>
<td>Dependencies</td>
<td>The solution described in this PoC is an instantiation of the Gray-Box Database Replication pattern in the context of WANs. The architectural pattern used in this</td>
</tr>
</tbody>
</table>
PoC depends also on the writeset extraction interfaces described in the *Writeset Extraction based on Extended DB Interfaces* pattern. Moreover, for failover purposes, this pattern may use the *Transparent Replication Proxy* in order to provide replication transparency to the clients. Finally, in order to discover the different replicas the *Multicast-Based Replica Discovery* pattern is used.

### Why

**Rationale**
The Gray-Box Database Replication pattern can be used to attain low latency for dynamic contents in edge computing. This PoC aims at quantifying the real latency that can be obtained by this pattern to determine its range of applicability.

**Architecture Component(s) affected**
The main architectural components affected by this PoC are the database component and its replication support. The pattern is instantiated in the context of edge computing within edge servers using a multi-tier service run-time (See *Multi-Tier Transactional Service Runtime* pattern), in which they used a WAN replicated database.

### Alternatives
The other alternatives for offering replicas of the services in WANs consist of accessing a centralized database to access dynamic contents. This alternative has as shortcoming the high latency observed by clients in the accesses to dynamic contents.

### Relationship
High availability and scalability are two key requirements of current SOI and
addressed by NEXOF-RA. In some application scenarios clients are geographically distributed. In these scenarios latency becomes the main non-functional aspect. The service run-time is typically architected in a multi-tier manner (See Multi-Tier Transactional Service Runtime pattern). This PoC applies the Gray-Box Database Replication pattern to address all these requirements with emphasis in minimizing the latency.

Legend of the pattern language for HA and Scalability

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution Categorization</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Tier Transactional Service Runtime</td>
<td>specializes/extends requirements</td>
<td></td>
</tr>
<tr>
<td>Replication of Transactional Multi-Tier Service Runtime</td>
<td>can be solved</td>
<td></td>
</tr>
<tr>
<td>DB Replication</td>
<td>can be solved</td>
<td></td>
</tr>
<tr>
<td>Middleware-Based DB Replication</td>
<td>can be solved</td>
<td></td>
</tr>
<tr>
<td>White-Box DB Replication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gray-Box DB Replication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black-Box DB Replication</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How

Scenarios for validation: The scenario chosen for this PoC is an edge computing environment for an e-commerce application in which clients are naturally geographically distributed: an online bookstore. This scenario is similar to the one that is described in D10.1 (e-Commerce information sharing, S12).

Suggested Architecture: The online bookstore application is a web application based on a three-tier architecture that is deployed over a WAN. In this case, each node (replica) of the cluster includes a web server (acting as an application server) that contains the presentation and the business logic of the application and a database system containing the full data used by the application in order to provide high availability. A client node runs emulated clients that inject request in each location of the replicated infrastructure.
Environment

The validation is going to be performed using the industrial benchmark for web applications, TPC-W that provides an online bookstore application for selling books through the Internet.

The particular choice of software components is the following:

- Tomcat web server/servlet container/application server. Contains the TPC-W application services.
- PostgreSQL database server. Contains the data of the TPC-W application.
- Middle-R replication middleware for replicating data across the WAN to provide low latency access to dynamic web contents.
- Ensemble group communication system in order to communicate the Middle-R replicas.

The operating system is an Ubuntu Linux distribution with kernel 2.6.24. The instances of the middleware for replication + Postgres will run in Intel Xeon 1.8 GHz servers with 3.5 GB of RAM. The Tomcat instances and the emulated clients will run in an Intel Pentium Core Duo 2.8 GHz with 2 GB of RAM.

The WAN is emulated by software by introducing a delay equivalent to one for the distance among nodes.

Estimated Effort

1-2 PMs. Main efforts: running the evaluation and assessing it, setup the cluster and the software (install all server software, TPC-W application, configure it at all sites, infrastructure for running the experiments, scripting, etc.).
### Methods

The PoC compares the observed latency of edge computing approach based on WAN database replication consists in comparing the results obtained with existing architectural choices for edge computing, namely, a central database server for the dynamic contents.

### Assessment criteria & metrics & way to document

The assessment criteria for this PoC are also partially based on the use of the utility trees of the ATAM methodology. Moreover, in this PoC, the business requirements of the utility tree have been obtained from the online bookstore application provided by the TPC-W benchmark. So, an utility tree is generated for the quality attributes of the PoC and a benchmark is used to measure the most important features. The utility tree for the Gray-Box DB Replication pattern applied in WANs implemented in this PoC is depicted in the following picture:

![Utility Tree for Pattern](image)

In this PoC we mainly aim to quantify the latency in a WAN of the Gray DB Replication pattern. So, apart from ATAM, we will use an evaluation methodology based on the same standard benchmark (TCP-W) used in the Database Replication Architectural Patterns For Highly Available Stateful Services PoC. We will inject increasing loads to a system with a replicated database deployed in a WAN with different configurations for dynamic contents till reaching saturation. We will obtain the average latency perceived by clients when performing requests. The latency is indicative of the QoS perceived by the clients. So, the latency of the edge-based approach for the dynamic contents of applications will be compared to the latency obtained in infrastructure that have centralized those contents. Since the benchmark itself does not express the requirements of availability, applicability and maintainability, they have been evaluated with regard to the requirements of the architectural pattern. Availability is measured as the number of single points of failure of the solution. The applicability is measured as the number and strength of assumptions and finally, the maintainability is
measured as the number of components we require to modify and maintain.

The generic quality attributes for the *Gray-Box DB Replication* pattern applied to WANs are the same as the ones described in the *Gray-Box DB Replication* pattern (See the document for the pattern).

### 3.1.4 Vertical Replication Architectural Pattern For Scalable and Highly Available Stateful Services

<table>
<thead>
<tr>
<th>POC Name</th>
<th>Vertical Replication Architectural Pattern For Scalable and Highly Available Stateful Services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What</strong></td>
<td></td>
</tr>
<tr>
<td>Owners</td>
<td>Universidad Politécnica de Madrid (UPM)</td>
</tr>
<tr>
<td>Description of the PoC</td>
<td>Current state-of-the-art for availability and scalability of services architected in multiple tiers only addresses the replication of session state at the application server tier (e.g. using the <em>Session Replication</em> pattern) or replication at the data tier (e.g. White-Box, <em>Gray-Box</em> and <em>Black-Box Database Replication</em> patterns), but not of both tiers at the same time. Thus, the non-replicated tier may become a single point of failure and a scalability bottleneck for the infrastructure. The <em>Vertical Replication</em> pattern evaluated in this PoC goes beyond current state-of-the-art by enabling to provide scalability and availability of services architected in multiple tiers whilst guaranteeing full consistency of the solution. This PoC aims to demonstrate the validity of architectural choices and architectural patterns described in NEXOF-RA with regard to vertical data replication in multi-tier architectures.</td>
</tr>
<tr>
<td>ACPs involved</td>
<td>The <em>Vertical Replication</em> pattern.</td>
</tr>
<tr>
<td>Objective of the PoC</td>
<td>The main objective of this PoC is to evaluate the trade-offs in non-functional attributes (scalability, availability, applicability and maintainability) between the vertical replication approach and other architectural choices such as non-replicated solutions and single-tier replication.</td>
</tr>
<tr>
<td>Functionalities</td>
<td>The PoC focus on functionalities to attain a holistic replication of multi-tier architectures.</td>
</tr>
<tr>
<td>Dependencies</td>
<td>The pattern used in this PoC is the <em>Vertical Replication</em> pattern. Other architectural patterns such as <em>Multicast-Based Replica Discovery</em>, <em>Session Replication</em> and <em>Transparent Replication Proxy</em> are also used by the <em>Vertical Replication</em> pattern for discovery of replicas and transparent failover and session state replication.</td>
</tr>
<tr>
<td>Why</td>
<td>High availability and scalability are two key features for applications and current SOI that is typically multi-tiered. The <em>Vertical Replication</em> pattern provides a holistic solution to satisfy these requirements. However, it is necessary to quantify the degree of scalability to understand in which scenarios it can be used. This PoC aims at performing this quantification to identify the scalability limits of the pattern and therefore, its applicability.</td>
</tr>
<tr>
<td>Rationale</td>
<td>The main architectural components affected by this PoC is the service runtime, more specifically its multi-tier specialization that consists of an application server and a database components (See <em>Multi-Tier Transactional Service Runtime</em>).</td>
</tr>
<tr>
<td>Architecture Component(s) affected</td>
<td></td>
</tr>
</tbody>
</table>
Alternatives

Other alternatives to the Vertical Replication pattern perform the replication of a single tier, either the middle-tier (See the Session Replication pattern) or the data-tier (see White-Box, Gray-Box and Black-Box Database Replication patterns). However, these solutions have single points of failure and bottlenecks in the non-replicated tier. An alternative pattern addressing the replication of both tiers is the Horizontal Replication pattern.

Relationship with the NEXOF-RA

High availability and scalability are two key requirements of current SOI and addressed by NEXOF-RA. The service run-time is typically architected in a multi-tier manner (See Multi-Tier Transactional Service Runtime pattern). The vertical replication pattern addresses precisely these requirements for this architectural pattern of the service run-time.
How Scenarios for validation

The scenario is a supply-chain management application architected with a multi-tier service run-time (See Multi-Tier Transactional Service Runtime pattern). This application allows to maintain an inventory of products by retailer and to sell/order manufactured goods when necessary. This scenario is related to the one that is described in D10.1 (e-Commerce information sharing, S12).

Suggested Architecture

The architecture includes application implemented in a three-tier architecture, with the web server, application server, and database tiers. The vertical replication pattern is used to replicate the whole system. Each node of the cluster includes a web server that includes the UI (presentation tier), an application server that runs the business logic of the application (middle-tier) and a database system containing the data used by the application (data-tier). A client node runs the emulated clients of the benchmark that inject requests in the system at a given rate.
Environment

The available hardware consists of computers with AMD Athlon dual CPUs (2GHz), with 1 GB of RAM and with 320 GB in the hard disk. The running operating system is a Fedora Linux distribution.

The validation is going to be performed using an industrial benchmark for J(2)EE application servers called SPECjAppServer, which includes the supply-chain management application that provides services to car dealers for selling cars through an intranet (LAN) environment.

The particular choice of software components is the following:

- Apache web server to distribute the load to the different replicas of the cluster.
- Tomcat application server. Runs the presentation part of the application.
- JOnAS J(2)EE application server. Contains the business logic of the application.
- Spread group communication system to communicate the application server replicas.
- PostgreSQL database server. Contains the data of the SPECjAppServer application.
Estimated Effort

1-2 PMs. Main efforts: setting up the system (install all server software, SPEC-AppServ, connect all components, configure them), setup the cluster (install the nodes, set up the software to run the nodes such as network booting, private network, NIS/NFS), running the evaluation and assessing it.

Methods

The method used to measure the advantages of the Vertical Replication pattern in multi-tier consists in comparing the results obtained in the performance evaluation of the vertical replication approach against a non-replicated approach and some of the standard replication solutions currently available for multi-tier applications.

Assessment criteria & metrics & way to document

The assessment criteria are partially based on the ATAM methodology that has been adapted to our specific needs that is to evaluate, quantify and compare the tradeoffs between architectural choices by means of utility trees. Moreover, in this case, the business requirements of the utility tree have been derived from a particular scenario related to an online bookstore application included in a benchmark. The utility tree for the Vertical Replication pattern applied in this PoC is depicted in the following picture:

![Utility Tree for Pattern](image)

In the figure are shown the business requirements and metrics identified (on the right side of the figure), technical requirements and quality attributes (on the left side).

Apart from ATAM, we will use an evaluation methodology based on industrial practice. We use a standard benchmark (TCP-W) that allows to evaluate the performance of databases in terms of throughput and response times. We will
inject increasing loads to the system till reaching saturation for each configuration. We extend the evaluation to quantify the scale-out by running the benchmark with different configurations. The scale out shows how many times a particular number of replicas multiply the peak throughput of a non-replicated setup. Also as part of the scalability evaluation the average response time for client requests will be obtained. Since the benchmark itself does not express the requirements of availability, applicability and maintainability, they have been evaluated with regard to the requirements of the architectural pattern. Availability is measured as the number of single points of failure of the solution. The applicability is measured as the number and strength of assumptions and finally, the maintainability is measured as the number of components we require to modify and maintain.

The generic quality attributes for the Vertical Replication pattern are (See the document for the pattern):

- Scalability \( \rightarrow + \)
- Applicability \( \rightarrow + \)
- Maintainability \( \rightarrow 0 \)

### Further Information

#### 3.2 PoC Results Related to High Availability and Scalability

#### 3.2.1 Database Replication Interfaces for Highly Available Stateful Services

In order to provide highly available stateful services, it is necessary to provide at the service infrastructure level high availability for persistent data. Data replication is the main technique for providing highly available and scalable data services. Different architectural patterns have been proposed for attaining data replication outside the DB. In order to extract the data to replicate them to the set of replicas, these database replication patterns require either the use of standard DB interfaces or the implementation of some minimal interface within the DB.

Scalability and Availability are two non-functional aspects covered by WP4.
3.2.1.1 **Scope of the PoC with respect the Reference Architecture**

![Diagram](Image)

**Figure 112: Scope of the PoC – Data Base Replication Interfaces**

In this PoC are involved patterns to extract data from databases. These patterns are:

- Trigger Writeset Extraction pattern,
- Log Minning Writeset Extraction pattern
- Writeset Extraction based on Extended Interfaces pattern

With regard to the standards catalogue used, the main standard is the Structured Query Language (SQL). SQL is a language that allows to retrieve and to manage data in relational database management systems and is standardized by ISO.

The patterns are related to the abstract database component, in particular relational databases. In this case, the patterns have been implemented for a mix of commercial and open source databases:

- PostgreSQL database server.
- Microsoft SQLServer.
- MySQL.
3.2.1.2 Evaluation methodology

The pattern is evaluated according to quality attributes it claims. These quality attributes are stated in the filled-in template description of the architectural pattern (according to a shared and agreed template provided by WP7). The stated quality attributes in this architectural pattern description are evaluated using the ATAM approach. Especially the quality attributes utility tree is used to show and analyze the results of the implementation of each quality attribute.

3.2.1.3 Quality attributes provided to WP7 for this pattern

The Trigger Writeset Extraction, Log Mining Writeset Extraction and the Writeset Extraction Based on Extended DB Interfaces patterns are specified with the work done on WP4. In the context of specification, this pattern has been identified to support certain defined quality attributes. These quality attributes affect the quality of the architecture where each pattern is integrated. Evaluating these quality attributes permits to know if the resulting architecture where the pattern has been applied meets the expected requirements.

[The significations of plus (+), minus (-) or neutral (0) are given in the specification of this pattern for D7.5]

<table>
<thead>
<tr>
<th>Triggers Writeset Extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Log Mining Writeset Extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Writeset Extraction based on Extended DB Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
</tr>
</tbody>
</table>
3.2.1.4 Evaluating quality attributes by using Pattern Quality Attributes Tree

The assessment criteria are partially based on the ATAM methodology, which has been adapted to our specific need that is to evaluate, quantify and compare the tradeoffs between architectural choices. The ATAM methodology was originally developed to assist architectural decisions by taking into account early in the design process the quality attributes.

We use the ATAM utility tree for matching the quality attributes (performance, applicability and maintainability) with metrics to be used in the evaluation. The utility tree for each pattern is presented in [D8.1 pXX Figure 2: Utility tree to support validation in PoC1] and a detailed description may be found in the same deliverable [D8.1 pXX PoC 1: Database Replication Interfaces for Highly Available Stateful Services] of.

In order to evaluate some of the quality attributes addressed by the patterns, we have performed several experiments with the different writeset extraction patterns. The setup of every experiment consists in a node containing a client request emulator launching requests to a database node implementing a particular writeset extraction pattern.

3.2.1.5 Evaluation results

The evaluation of the performance quality attribute identified in the ATAM’s utility tree (See D8.1) for the three patterns has been performed using synthetic workloads to measure the performance (in terms of throughput and response times metrics) of the different interfaces.

In Figure 113 and Figure 114 are shown the throughput and the response time of Microsoft SQL Server DBMS for capturing the writesets using log mining and triggers. The baseline has been obtained measuring the standard execution of transactions.

The throughput of the baseline is around 45 requests per second and the response times for the baseline (Figure 114) are around 20 ms for low loads and 200 ms for high loads of the system.

With regard to the throughput exhibited by capturing the writesets by performing log mining, it can be seen that the throughput is significantly reduced (around a 85 %). Of course, the response times confirm also the performance degradation.

Finally, the throughput obtained when using triggers also shows a drop in the throughput obtained. It is not so noticeable as the one observed with the log mining but is around a 55-60 %. However, for the response time, the transaction latency is not affected dramatically compared to the normal execution.

So, the data showed in this figures shows that there is a significant degradation of the performance when using the current state-of-the art interfaces for extracting the writesets from the database.
Figure 113: Microsoft SQL Server Throughput

Figure 114: Microsoft SQL Server Response Time

The next two figures (Figure 115 and Figure 116) show the throughput and response times when the binary (black-box) writeset extraction is done in the MySQL DBMS kernel.
A loss of throughput around the 35% can be observed in the maximum throughput obtained in the two configurations evaluated (335 req/sec in the normal execution without capturing the writesets vs. 220 req/sec when the SQL binary writeset is captured). When the throughput starts to stabilize, the MySQL execution without capturing the writeset stabilizes around 170 req/sec. On the other hand, the configuration capturing the writeset stabilizes at 120 req/sec,
which represents a loss of throughput around a 30%, which is quite affordable. In saturation, the performance of the configuration drops to 65 req/sec.

With regard to the response time, for low loads until medium loads, both configurations perform very similar. For higher loads, the response time of the configuration extracting the writesets grows faster than the standard configuration.

Figure 117 and Figure 118 show the throughput and the response time for the binary writeset extraction in PostgreSQL. The throughput for both configurations is more or less the same (around 11 requests per second). With regard to the response time, it grows linearly with the number of clients in both configurations, what means that the cost of the writeset extraction is almost negligible. This is due to the fact that the writeset extraction can be done as a local task for each connection and does not require synchronization with other tasks performed by the database management system.

Figure 117: Postgres Throughput: WS Extraction
Finally, the next two figures (Figure 119 and Figure 120) show the throughput and the response time of the writeset application in PostgreSQL. Two configurations have been analyzed. The first one applies the SQL writesets obtained (textual form), and the second one applies binary writesets. Both configurations are compared against the normal execution of transactions in Postgres.

Figure 118: Postgres Response Time: WS Extraction

Figure 119: Postgres Throughput: WS Application
In the two configurations applying the writesets, the throughput achieved is higher. Applying the binary writesets is around a 10-20% more performant than applying SQL writesets. For medium and high loads, the throughput achieved is between 100 and 150 requests per second. Compared to the baseline that just applies update transactions, the gain in performance of both configurations applying the writesets is quite significant. The analysis of the response time shows a similar behaviour. Whilst the normal execution of transactions in PostgreSQL grows very fast, the configurations applying the writesets are much more performant, showing response times between 10 and 100 ms.

The metrics and the evaluation for the other quality attributes identified in the ATAM’s utility tree for the patterns (See D8.1) are commented below.

With regard to applicability, neither the Trigger Writset Extraction nor the Log Mining Writset Extraction patterns require to access the code of the database component in order to implement writset extraction. This means that maintainability is also avoided for these solutions. The main difference between them is the fraction of implementations providing these mechanisms. The trigger mechanism is present in almost all database management system (both commercial and open source) whilst the tools for performing log mining are not so common in database products. However, the purpose of these two mechanisms is not writset extraction, so the performance of the solutions that implement them is affected as the previous analysis has shown. To the best of our knowledge, there are no database systems that offer the specific interfaces for writset extraction described in the Writset Extraction Based on Extended DB Interfaces pattern. In order to apply this pattern, it is required to access the internals of the database. The modifications that need to be done are moderate.
3.2.1.6 Conclusion

Database replication has as one of the main sensitivity points the cost of extracting and applying the updates (writesets) performed by transactions.

The applicability and maintainability of database replication has also as main sensitivity point, how the writesets are extracted and applied since they affect the assumptions on the target database system.

The trade-off that has been identified is that having high applicability and maintainability when using standard database mechanism to extract the writesets (triggers and log mining), results in a high overhead, what hampers the scalability for update workloads that is directly dependant on the cost of writeset extraction and application.

Reducing the cost of writeset extraction is possible by using an extended database interface, what increases the scalability. However, this is attained at the cost of applicability and maintainability that are decreased due to the requirement of having access to the database code in order to extend its interface.

Therefore, the trade-off is between applicability and maintainability, and performance (that includes indirectly scalability, attainable when these interfaces are applied to database replication).
3.2.2 Gray-Box Database Replication Architectural Pattern For Highly Available Stateful Services

In order to provide highly available stateful services in SOI, it is necessary to provide high availability for persistent data. Databases are the components in charge of storing data of services and applications. The lack of availability of databases derives on the lack of availability of the services and applications that persist their data in them. This PoCs aims to demonstrate how high availability and scalability are provided to databases.

3.2.2.1 Scope of the PoC with respect the Reference Architecture

This PoC is based on the Gray-Box Database Replication pattern.

With regard to the standards catalogue used, the main standard is the Structured Query Language (SQL). SQL is a language that allows to retrieve and to manage data in relational database management systems and is standardized by ISO.

The pattern is related to the abstract database component of the NEXOF-RA, in particular relational databases. In this case, the gray-box approach has been implemented on top of the PostgreSQL database server.

The gray-box database replication approach also uses the Database Replication Middleware abstract component of the NEXOF-RA in order to...
encapsulate the replication logic. In this PoC, the Middle-R replication software has been used.

3.2.2.2 Evaluation methodology

The pattern is evaluated according to quality attributes it claims. These quality attributes are stated in the filled-in template description of the architectural pattern (according to a shared and agreed template provided by WP7). The stated quality attributes in this architectural pattern description are evaluated using the ATAM approach. Especially the quality attributes utility tree is used to show and analyze the results of the implementation of each quality attribute.

3.2.2.3 Quality attributes provided to WP7 for this pattern

The Gray-Box Database Replication pattern is specified with the work done on WP4. In the context of specification, this pattern has been identified to support certain defined quality attributes. These quality attributes affect the quality of the architecture where each pattern is integrated. Evaluating these quality attributes permits to know if the resulting architecture where the pattern has been applied meets the expected requirements.

[The significations of plus (+), minus (-) or neutral (0) are given in the specification of this pattern for D7.5]

<table>
<thead>
<tr>
<th>Gray-Box Database Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability                   +</td>
</tr>
<tr>
<td>Availability                  +</td>
</tr>
<tr>
<td>Applicability                 0</td>
</tr>
<tr>
<td>Maintainability               0</td>
</tr>
</tbody>
</table>

3.2.2.4 Evaluating quality attributes by using Pattern Quality Attributes Tree

As for the previous PoC, the assessment criteria are partially based on the ATAM methodology adapted to our specific needs (mainly the adoption of utility trees). However, in this case the business requirements have been derived from a particular scenario related to an online bookstore application included in a benchmark.

The utility tree for the Gray-Box DB Replication pattern applied in this PoC is depicted in D8.1 [D8.1 pXX Figure 4: Utility tree to support validation in PoC2]. A detailed description may be found in the same deliverable [D8.2 pxx PoC 2: Gray-Box Database Replication Architectural Pattern For Highly Available Stateful Services].
In order to evaluate some of the quality attributes of the pattern, we have performed several experiments. The validation has been done using the industrial benchmark TPC-W for web-based applications. The setup for the experiment consists in a node containing a client request emulator launching requests to an increasing set of database replicas each one deployed in a different node. The particular choice of software components for the replicas is the following:

- Tomcat web server/sevlet container/application server. Contains the TPC-W application services.
- PostgreSQL database server enhanced with binary writeset extraction/injection. Contains the data of the TPC-W application.
- Middle-R replication middleware for replicating PostgreSQL data.
- Ensemble group communication system in order to communicate the Middle-R replicas.

3.2.2.5 Evaluation results

The evaluation of the scalability and performance quality attributes identified in the ATAM’s utility tree for the *Gray-Box DB Replication* pattern has been performed using an industrial benchmark for web applications called TPC-W, which provides an e-commerce site for selling books through the Internet. The emulated clients of the TPC-W benchmark inject a workload mix that consists of 75% read transactions and 25% update transactions. The environment is based on a typical SOA multi-tier stack consisting of a web server (omitted from the figure for the sake of clarity), a servlet container/application server and a database server. The data of the TPC-W application’s database is replicated completely (full replication) into a set of replicas that conform a cluster. A DB replication middleware performs this task. This guarantees high availability for the data of the TPC-W application.

The metrics to show the scalability of the approach is the scale-out of the solution. This means, how many times a particular number of replicas multiply the peak throughput of a non-replicated setup. Also as part the scalability the response time has been measured.

The scale-out is shown in Figure 122. The results are shown for an increasing number of nodes (replicas) in the cluster. We can see that when new nodes are added the scale-out of the cluster of replicas increases. We can observe that when using two nodes in the system, the scale-out is almost 2, meaning that the system can almost double the performance of a non-replicated system. Of course, the scale-out does not increase linearly with the number of replicas due to the overhead introduced by the replication of all the modified data to all the replicas.
In Figure 123 are shown the response times for increasing loads measured in transactions per second (tps). With regard to the response times obtained, they increase until the system reaches saturation, and it stabilizes around 1300 ms. As the TPC-W clients are synchronous, the response time does not grow exponentially when the system reaches saturation.

Figure 123: Scale-out

The metrics and evaluation for the other quality attributes identified in the ATAM’s utility tree for the Gray-Box DB Replication pattern application (See
D8.1) are commented in the following text. In order to apply the pattern and maintain the implemented solution (applicability and maintainability attributes) it is necessary to modify the database code in order to implement the writerset extraction interfaces (See *Writerset Extraction Based on Extended DB Interfaces* pattern). The modification on the database source code is moderate, and mainly involves the transaction management process. It is also necessary to provide a middleware replication component on top of the database in order to implement the replication protocol. Finally, with regard to the availability of the solution proposed, it is possible to support N-1 replica failures. If any of the components of a replica fails (application server, database replication middleware or database), the replica is considered as failed.

### 3.2.2.6 Conclusion

We have identified multiple sensitivity points for the architecture of the database component: availability, scalability, performance, applicability and maintainability.

A centralized database provides excellent applicability and maintainability since databases are off-the-self components available for multiple vendors. Regarding performance, it can provide good performance for low loads manageable by a single database instance. However, a centralized database provides low availability since it does not tolerate neither crash failures of the underlying node, nor the software infrastructure. Furthermore, a centralized database can at most scale-up by buying more expensive hardware but does not scale-out since it does not have any mechanism to share the load across nodes.

The architectural alternative to a centralized database is a replicated database. As it has been proven in the PoC evaluation it provides a medium scalability, attaining scale-outs of 3.5x with 14 node replicas. Additionally, it tolerates as many failures as replicas minus one, providing high availability. The performance is also kept for very high loads.

However, these improvements in availability, scalability and performance with respect to the centralized approach, imply trading-off applicability and maintainability, since the gray-box database replication pattern requires an extended database interface that typically requires access to the database code.

Therefore, there is a trade-off between applicability and maintainability, and scalability, availability and performance.
3.2.3 Gray-Box Database Replication Architectural Pattern in WANs

Nowadays, many services are provided by the companies through the web. The users of these services are many times scattered around large geographical areas and even around the globe. Centralized servers result in large latencies for distant clients. Edge computing allows diminishing the latency of static contents, but access to dynamic contents is still centralized. The *Gray-Box DB Replication* pattern applied to WANs described in this PoC enables to provide low latency to both static and dynamic contents to all clients. Therefore, this pattern allows providing scalable and high-available systems.

Scalability and Availability are two non-functional aspects covered by WP4.

### 3.2.3.1 Scope of the PoC with respect the Reference Architecture

![Diagram of the PoC](image)

**Figure 124: Scope of the PoC – Gray-Box Database Replication applied to WANs**

This PoC is based on the *Gray-Box Database Replication* pattern applied in WANs.

With regard to the standards catalogue used, the main standard is the Structured Query Language (SQL). SQL is a language that allows to retrieve and to manage data in relational database management systems and is standardized by ISO.
The pattern is related to the abstract database component of the NEXOF-RA, in particular relational databases. In this case, the gray-box approach has been implemented on top of the PostgreSQL database server.

The gray-box database replication approach uses the Database Replication Middleware abstract component of the NEXFOF-RA in order to encapsulate the replication logic. In this PoC, the Middle-R replication software has been used.

3.2.3.2 Evaluation methodology
The pattern is evaluated according to quality attributes it claims. These quality attributes are stated in the filled-in template description of the architectural pattern (according to a shared and agreed template provided by WP7). The stated quality attributes in this architectural pattern description are evaluated using the ATAM approach. Especially the quality attributes utility tree is used to show and analyze the results of the implementation of each quality attribute.

3.2.3.3 Quality attributes provided to WP7 for this pattern
This PoC is related to the Gray-Box Database Replication pattern in the context of WANs. The Gray-Box Database Replication pattern is specified with the work done on WP4. In the context of specification, this pattern has been identified to support certain defined quality attributes. These quality attributes affect the quality of the architecture where each pattern is integrated. Evaluating these quality attributes permits to know if the resulting architecture where the pattern has been applied meets the expected requirements.

[The significations of plus (+), minus (-) or neutral (0) are given in the specification of this pattern for D7.5]

<table>
<thead>
<tr>
<th>Gray-Box Database Replication</th>
</tr>
</thead>
</table>
| Scalability                  | +  
| Availability                 | +  
| Applicability                | 0  
| Maintainability              | 0  
| Latency in WANs              | +  

3.2.3.4 Evaluating quality attributes by using Pattern Quality Attributes Tree
As in the previous PoCs, a utility tree is generated for the quality attributes of the PoC and a benchmark is used to measure the most important features. The utility tree for the Gray-Box DB Replication pattern applied in WANs is depicted in D8.1 [D8.1 p XX Figure 8: Utility tree to support validation in PoC4].
detailed description can be found also in D8.1 [D8.1 pXX PoC 4: Gray-Box Database Replication Architectural Pattern in WANs].

In order to evaluate some of the quality attributes of the pattern, we have performed several experiments. The validation has been done using the industrial benchmark TPC-W for web-based applications. The setup for the experiment consists in a node containing a client request emulator launching requests to an increasing set of database replicas each one deployed in a different node. The particular choice of software components for the replicas is the following:

- **Tomcat** web server/sevlet container/application server. Contains the TPC-W application services.
- **PostgreSQL** database server enhanced with binary writeset extraction/injection. Contains the data of the TPC-W application.
- **Middle-R** replication middleware for replicating PostgreSQL data.
- **Ensemble** group communication system in order to communicate the Middle-R replicas.

### 3.2.3.5 Evaluation results

This PoC aims at evaluating the real benefits of the architectural pattern and its impact in the latency perceived in WANs as has been identified in the performance attribute of the ATAM’s utility tree for the PoC. The metric used has been the quantification of the reduction of the latency perceived by the clients using an implementation of the pattern compared to the one of a centralized web server.

The validation has been performed also using the industrial benchmark for web applications called TPC-W. It provides an e-commerce site for selling books through the Internet. The emulated clients of the TPC-W benchmark inject a workload mix that consists of 75% read transactions and 25% update transactions. The system has 4 edge servers and looks similar to the architecture described in Figure 125. Each edge server consists on a typical SOA multi-tier stack that includes 2 nodes. In the first node, the web server/servlet container/application server executes the business logic of the TPC-W application. The second node holds the middleware-based replicated database. The JDBC driver makes transparent to the application server the different database replicas. The two nodes that configure an edge are connected through a LAN. The WAN that connects the edge sites has been emulated and includes a latency of 40 ms.
Figure 126 shows the response time for the evaluated configurations. In all configurations, the data of the TPC-W application’s database is replicated completely (full replication) into the set of edge servers. First of all, we have analyzed two baselines. The baselines follow a primary-backup scheme for the transactions that have updates. Whilst read-only transactions can be executed in any edge server, update transactions are executed in one of the edge servers (the primary) and the changes are lazily replicated to the other replicas (the backups). Lazily means that the updates are replicated to the backups after the transaction has been committed in the primary. In the first of the baselines (labelled as LPN), the web/servlet container that receives the client requests, executes the update transactions and the replication middleware sends one message per database access. That is, if a database transaction includes several operations, one message per operation will be sent through the WAN.

In the other baseline (labelled as LP1) the replication middleware only sends one message at the end of the transaction including all the possible operations performed in the transactional context. These two baselines have two main drawbacks. The first one is that clients may not perceive their own updates when executing two serial transactions, the first one updating A and the second reading it again in another edge. This is because maybe the server where the read operation is done has not received yet the previous update. The other disadvantage may occur when the primary edge server crashes and has not propagated the changes to the backups. This scenario may lead loss data.
Finally, in the configuration labelled as SEQ, both types of operations -read and write- can be performed on every edge server (called update everywhere). Therefore, the web/servlet container does not need to be aware of the replicated infrastructure because the transactions are executed locally. Before committing the transactions, a certification protocol based on snapshot isolation is executed in order to guarantee the consistency in the whole set of edge servers. This requires to send through the WAN only one message round (per transaction) containing the changes produced in the transaction (eager replication).

![Graph showing response time](image-url)

**Figure 126: Response Time**

As we can see in the figure, all the curves offer an acceptable response times (around 600 ms) until 500 clients. From that point on, the primary-backup replication approaches (LP1 and LPN) start to grow exponentially. This can be explained because only one server is in charge of processing all the transactions that include updates. Moreover, the response time of the LPN curve grows faster than the LP1 because it requires to send more messages through the WAN (one per update operation in LPN). However, the response time of the update everywhere approach exhibits very good behaviour. It is able to serve 1200 clients with an average response time of 350 ms and also offers much more performant response times than the baselines when the system is less overloaded (e.g. from 100 till 700 clients). This means a better customer experience.

The metrics and evaluation for the other quality attributes identified in the ATAM’s utility tree for the **Gray-Box DB Replication** pattern applied to WANs (See D8.1) are the same as in the PoC 3.7 **Database Replication Architectural Pattern For Highly Available Stateful Services.**
3.2.3.6 Conclusion

The main sensitivity point for edge computing is the observed performance, especially response time, by end clients. Response time is highly impacted by the architecture of the edge computing system.

In this PoC, we have compared a traditional edge computing architecture based on a centralized database for dynamic contents with a WAN replicated database architectural choice.

The trade-off is between observed performance by end clients (response time) and the applicability and maintainability of the architectural choice. The traditional centralized database approach results in poor performance for dynamic contents, since it has high latencies to access the geographically distant centralized database in a per-interaction with dynamic contents basis. On the other hand, it has good applicability and maintainability due to it is based on off-the-shelf component.

Performance (in terms of response time) can be dramatically improved by adopting the Gray-Box Database Replication pattern for storing the dynamic contents. This enables to perform local reads and writes plus a single WAN interaction per transaction. However, this improvement is attained by trading-off applicability and maintainability due to the requirement of extended database interfaces for writeset extraction.

3.2.4 Vertical Replication Architectural Pattern For Scalable and Highly Available Stateful Services

In order to provide highly available and scalable stateful services in SOI, it is necessary to take into account also the multi-tier architectures supporting those services. Vertical replication allows providing scalability and high availability for stateful services deployed in multi-tier architectures. This approach consists in replicating pairs of application and database servers, encapsulating the replication logic in the application server.

Scalability and Availability are two non-functional aspects covered by WP4.
3.2.4.1 **Scope of the PoC with respect the Reference Architecture**

![Diagram](image)

**Figure 127: Scope of the PoC – Vertical Replication**

This PoC is based on the *Vertical Replication* pattern. Additionally the *Session Replication* pattern has been used in the evaluation of the PoC.

With regard to the standards catalogue used, the Structured Query Language (SQL) is required for the underlying relational databases required by the pattern. SQL is a language that allows retrieving and managing data in relational database management systems. It is standardized by ISO. J(2)EE is a specification lead by Sun Microsystems that specifies how to build compatible application servers using Java technology.

The pattern is related to the application server and database abstract components of the NEXOF-RA. The application server includes the replication logic for the solution proposed. In this case, the *Vertical Replication* pattern has been implemented in this PoC using an open-source J(2)EE compatible application server (JOnAS), and an open-source relational database (Postgresql).

3.2.4.2 **Evaluation methodology**

The pattern is evaluated according to quality attributes it claims. These quality attributes are stated in the filled-in template description of the architectural pattern (according to a shared and agreed template provided by WP7). The
stated quality attributes in this architectural pattern description are evaluated using the ATAM approach. Especially the quality attributes utility tree is used to show and analyze the results of the implementation of each quality attribute.

3.2.4.3 Quality attributes provided to WP7 for this pattern

The Vertical Replication pattern is specified with the work done on WP4. In the context of specification, this pattern has been identified to support certain defined quality attributes. These quality attributes affect the quality of the architecture where each pattern is integrated. Evaluating these quality attributes permits to know if the resulting architecture where the pattern has been applied meets the expected requirements.

[The significations of plus (+), minus (-) or neutral (0) are given in the specification of this pattern for D7.5]

<table>
<thead>
<tr>
<th>Vertical Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Availability</td>
</tr>
<tr>
<td>Applicability</td>
</tr>
<tr>
<td>Maintainability</td>
</tr>
</tbody>
</table>

3.2.4.4 Evaluating quality attributes by using Pattern Quality Attributes Tree

The method is the same as the one used for the previous pattern(s). A utility tree is generated for the quality attributes of the PoC and benchmark is used to measure the most important features. In this PoC the business requirements have been derived from a scenario related to a supply-chain management application included in the benchmark. The utility tree for the Vertical Replication pattern applied in this PoC is depicted in the [Figure 6 - Utility Tree to support validation in PoC3] of D8.1. The description of details can be found in [PoC 3: Vertical Replication Architectural Pattern for Scalable and Highly Available Stateful Services] of this same deliverable.

In order to evaluate some of the quality attributes of the pattern, we have performed several experiments. The validation has been done using the industrial benchmark SPECjAppServer for web-based applications. The setup for the experiment consists in a node containing a client request emulator launching requests to an increasing set of vertical replicas each one deployed in a different node. The particular choice of software components for the vertical replicas is the following:
- Tomcat application server. Runs the presentation part of the application.
- JOnAS J(2)EE application server. Contains the business logic of the application.
- Spread group communication system to communicate the application server replicas.
- PostgreSQL database server. Contains the data of the SPECjAppServer application.

The Apache web server has been used to distribute the load to the different replicas.

3.2.4.5 Evaluation results

One of the main goals of this PoC is to show an evaluation of the scalability and performance quality attributes identified in the ATAM’s utility tree for the Vertical Replication pattern implemented in this PoC (See D8.1). To achieve this objective, we have evaluated it using the SPECjAppServer industrial benchmark.

The baseline to compare the scale-out of the Vertical Replication pattern was the non-replicated system. We also evaluated the current state-of-the-art solution that is Session Replication pattern applied with an update-everywhere approach. The configuration of this solution uses two application servers accessing a single database. In both configurations there was another node that emulated the clients injecting the load to the corresponding system configuration.

We have implemented the Vertical Replication pattern combined with an updated everywhere approach. For this pattern we used up to ten replicas. The implementation includes a replicated cache that is kept consistent in all the replicas of the cluster by means of a certification protocol. The assessment criteria has been the scalability of the approach measured as scale-out, that is, how many times a particular number of replicas multiply the peak throughput of a non-replicated setup. Additionally, the throughput and the response time have been measured for an increasing number of replicas. In all configurations, the clients in are executed in a separated machine and each application server and its corresponding database are collocated in the same server.

Figure 128 shows the overall throughput of the baselines in Tx/sec for increasing loads measured in injection rates (IR).
Figure 128: Throughput of the baselines

The first one evaluates the non-replicated configuration (the yellow bar). The second one has been obtained with the session replication pattern (the green bars). This configuration is the one commonly found in commercial implementations and represents the state of the art. Finally, the red line shows the expected throughput by the SPECjAppServer benchmark.

The first noticeable fact is that the non-replicated configuration and session replication can only handle a load up to an IR of 3.

Finally, the throughput of the vertical replication pattern has been evaluated using up to 10 replicas. The throughput results are shown in Figure 129.
Figure 129: Vertical replication throughput results

In contrast, the vertical replication results outperform the two baseline implementations by a factor of 2, even if there is only one replica. The reason is that the multi-version cache is able to avoid many database reads compared to regular caching. Session replication did not help because the shared database was already saturated with two application server replicas.

The vertical replication approach is able to handle a load up to 14 achieving the required throughput with 9 and 10 replicas. That is, by adding new replicas to the cluster a higher number of client requests can be served.

Even when the vertical replication configurations saturate (that is, when the throughput is lower than the injected load), configurations with a higher number of replicas exhibit a more graceful degradation. For instance, for IR = 13, both the 5-replica and 8-replica configuration are saturated. However, the throughput achieved with 8 replicas is higher than with 5 replicas, providing a better service to the clients. This is very important, since it will help the system to cope with short-lived high peak loads without collapsing.

So, to summarize, we have achieved scalability for stateful applications, increasing the throughput when new replicas are added to a cluster. Moreover, high availability is guaranteed because if one of the replicas fails, the other replicas can still serve the client requests.

Figure 130 shows the response times for read-only transactions. It can be observed that the curves of our protocol are almost flat independently of the number of replicas even at high loads when the system reaches saturation. This contrast with the behaviour of the two baselines, that grows exponentially. The
reason is that for read-only queries our application server caching used in the vertical replication pattern is very effective avoiding expensive database access in many cases. Moreover, read-only transactions don’t require communication among the replicas.

**Figure 130: Response time for read only transactions**

Update transactions are quite different. They are shown in Figure 131.
Figure 131: Response time for update transactions

The response times for the non-replicated configuration and session replication are worse than the ones for the multi-version approach even for low loads. This means that our caching strategy saves expensive accesses to the database. Moreover, the more replicas the system has, the more graceful is the degradation of the response time at the saturation point. As we have mentioned before, this is important since acceptable response times can be provided in case of short-lived peak loads.

The metrics and the evaluation for the other quality attributes identified in the ATAM’s utility tree for the Vertical Replication pattern application (See D8.1) are commented in the following text. With regard to the availability of the solution proposed, it is possible to support N-1 replica failures. As a replica is considered the set of application server and database, if any of them fails, the replica is considered as failed. In order to apply the pattern and maintain the implemented solution (applicability and maintainability attributes) it is only necessary to modify the application server code in order to implement the replication protocol. The modification on the source code is moderate, and mainly involves the transaction management process and persistence at the application server. The database code does not need to be modified because it is accessed by means of standard interfaces, in this case JDBC.

3.2.4.6 Conclusion

The main sensitivity points in multi-tier service runtimes lie in the attained availability, performance, scalability, applicability and maintainability.
The architecture of the runtime highly affects these quality attributes. A traditional approach, with a centralized application server and database, has a good applicability and maintainability due to the availability of application servers and databases as off-the-self components. The performance is reasonable for low loads that do not saturate the application server nor the database.

However, the availability is quite poor since a failure in either the node holding the application server instance and database or a failure in either of the software infrastructures, results in the lack of service availability. That is, all components are single points of failure, resulting in the unavailability of any service deployed in the multi-tier service runtime. Additionally, they do not provide scalability and the performance is absolutely degraded for medium and high loads.

The approach based on the Session Replication pattern enables to improve modestly the scalability for loads that are data intensive and increases slightly the availability of the application server component by tolerating failures of its replicas.

The Vertical Replication pattern attains a good scalability till around ten replicas and high availability tolerating as many failures as replicas minus one of any of the components.

Both, the Session Replication pattern and the Vertical Replication pattern, trade-off applicability and maintainability due to they require the modification of the application server in order to introduce the replication logic. Therefore, the trade-off lies between applicability and maintainability, and scalability, performance and availability.

3.3 PoC Descriptions Related to Security

The following PoC has been selected and setup based on a proposal coming from WP4 and took over by Thales based on work performed and achieved by WP4 in close cooperation with WP7. The Trusted Timestamping and Non-Repudiation patterns are part of the Security In ESOA pattern contributed to Enterprise SOA pattern. [This was reported in D8.1.]

### 3.3.1 Security PoC

<table>
<thead>
<tr>
<th>POC Name</th>
<th>Security PoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>What</td>
<td></td>
</tr>
<tr>
<td>Owners</td>
<td>Thales</td>
</tr>
<tr>
<td>Description of the PoC</td>
<td>The security PoC demonstrates the validity of promoted architectural choices (WS-Security and WS-Policy) and architectural patterns:</td>
</tr>
<tr>
<td></td>
<td>- Non-repudiation architectural pattern</td>
</tr>
<tr>
<td></td>
<td>- Trusted Time stamping architectural pattern</td>
</tr>
<tr>
<td>ACPs</td>
<td>Architectural choices are used the following standard requirements of the</td>
</tr>
</tbody>
</table>
involved

web services (WS):

**WS-Security** (Web Services Security) is a communications protocol providing a means for applying security to Web services.

**WS-Policy** is a specification that allows web services to use XML to advertise their policies (on security, Quality of Service, etc.) and for web service consumers to specify their policy requirements.

**Two architectural patterns involved in the security PoC are described hereafter:** (these descriptions serve to understand the security application context)

**Non-repudiation** is the concept of ensuring that a party in a dispute cannot repudiate, or refute the validity of a statement or contract. Although this concept can be applied to any transmission, including television and radio, by far the most common application is in the verification and trust of signatures.

According to traditional legal practice, a signature on a paper contract or memorandum may always be repudiated by the signatory. Such repudiation may take one of two forms: The signatory may claim fraud or forgery, such as "I did not sign that." Alternately, he/she may accept the signature as authentic but dispute its validity due to coercion, as in the scenario of blackmail or confessions given under torture.

The legal burden of proof differs depending upon the repudiation reason. In the former scenario the burden of proof typically rests on the party claiming validity, while in the latter it shifts to the signatory claiming lack thereof.

**Trusted timestamping** is the process of securely keeping track of the creation and modification time of a document. Security here means that no one, not even the owner of the document, should be able to change it once it has been recorded provided that the timestamper's integrity is never compromised. By consequence, this pattern trusts the timestamp of a document.

The administrative aspect involves setting up a publicly available, trusted timestamp management infrastructure to collect process and renew timestamps.

Objective of the PoC

Firstly, the objective of the PoC serves to demonstrate the results coming from WP4 (Non-Functional research package where Thales leads in security domain) based on work performed and achieved by WP4 in close cooperation with WP7.

Secondly, Thales extracted two patterns from the set of security patterns in order to prove the feasibility of the research work package. This work is based on the solid business requirements defined by the WP10 (especially the D10.1).

This PoC has come from with two patterns. These patterns have already contributed to the WP7 (Specification Reference Architecture).

By this way, they aims to prove the following quality attributes:

For Non-Repudiation Pattern:
- Integrity
- Authenticity

For the Trusted Timestamping Pattern:
- Integrity
Accountability

The implementation of these patterns will improve the:

Integrity: the document is digitally signed. Only the owner can modify the document.

Authenticity: the document is signed with a certificate. This certificate is based on the trusted third party. And the certificate contains the information concerning the owner. When a document is used with a certificate, we know who is the author of this document.

Accountability: to make a track on a document. The trusted third party as TSA (Trusted Timestamping Authority) stamp a time on the document. This authority signs also the document + timestamp. The timestamp put in the document is a formal prove. This timestamp cannot be modified by others thanks to the signature.

Functionalities

The PoC focus on a business scenario which helps the citizens to declare their tax securely and on the corresponding choice of the patterns. These patterns provide the functionalities in order to fulfil the objective of the requirements of the scenario:

Integrity/ Non-Repudiation/ Traceability.

It means that we have some security requirements. How can use the security components to obtain these functionalities.

By consequence, we define requirements from business process. Coming up with this work, the security requirements help to find the functionalities needed. These functionalities will be translated to the choice of the patterns.

Dependencies

Anticipated functional dependencies on architectural patterns related to (at least):

- Public Key Infrastructure
- Secure IP Networking
- Services Composition
- Services Runtime

The dependencies of Non-Repudiation pattern and Trusted Timestamping with other security patterns:

![Diagram](image)

Figure 132: dependencies of Non-Repudiation and Trusted Timestamping with Security In ESOA pattern

For more details about the relation [isPartOf], it is defined in the template
pattern which is specified by WP7.

<table>
<thead>
<tr>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rationale</td>
</tr>
</tbody>
</table>
| To come up with trustworthy service composition and runtime platform.  
To first demonstrate the potential of security architectural patterns to cope with security requirements and second to pave the way towards effective usage of these security patterns in conjunction with other patterns from other areas or concerns to solve additional security needs.  
The essential rationale is to build the PoC in the way to help us to understand each block of the Reference Architecture Structure described in the DoW. And at the end, we tried to map our work to this RA structure. |

<table>
<thead>
<tr>
<th>Architecture Component(s) affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>The components are affected by this PoC:</td>
</tr>
</tbody>
</table>

**Figure 133: components of Trusted Timestamping pattern**

**Figure 134: components of Non-Repudiation pattern**

These components were described in the Trusted Timestamping and Non-Repudiation patterns which are part of Security In ESOA pattern.. They have already described and contributed in the ESOA pattern (Top-Level Pattern) specified by the WP7.

Forecasting the next step, these components could also participate to the Internet of Services.

<table>
<thead>
<tr>
<th>Alternatives</th>
</tr>
</thead>
</table>
| There are some alternatives to guarantee the integrity by using the Authorization Pattern. This pattern defines the access control to the data. Only the authorized access can modify the data.  
We can also encrypt the data. Only the owner can decrypt the data and modify it.  
Concerning the authenticity, other manner to achieve it is to use the authentication. But the authentication is more difficult to achieve and is not
practical to put in place in the context of Tax Declaration. And the Accountability can be achieved by logging all security information. But the traceability used by this scenario impose us to use only the Trusted Timestamping Pattern

| Relationship with the NEXOF-RA | The two patterns extracted from the set of the security patterns defined on WP4 and specified in two documents for WP7 can be used as a service. It means that these patterns are compliance with SaaS\(^{25}\). Moreover, these patterns can be used in the IoS\(^{26}\).

And the scenario used for the Tax Declaration described in the PoC is the use case on the Internet.

Our work is driven in the direction prescribed by the Nexof-RA (the future of internet). |
| --- | --- |

### How

| Scenarios for validation | Firstly, one of the main scenarios (among the ones described in D10.1) we selected to validate those architectural choices and patterns is the e-government online fee visualization and payment service (S16) of D10.1 and more specifically in the context of this Security PoC, a fiscal portal where citizens declare their income tax. Through it, we demonstrate the validity of two architectural choices and two patterns w.r.t. to important requirements for Secured and trustworthy SOA environments which are Non-repudiation and Trusted Timestamping. These requirements are also stated in D10.1 (e.g. Non-repudiation is R39 (non-repudiability of data transfer) and the Trusted Timestamping is R33 (trust and confidence)). The choice of these patterns in the e-government context takes place in the real need of modernization of the interface between the citizen and the government by making it more efficient and cost-saving. These patterns permit not only to validate the reference architecture of Nexof but also to be useful for the million of citizens.

To come up with trustworthy service composition and runtime platform, we’d like first demonstrating the potential of security architectural patterns to cope with security requirements and second to pave the way towards effective usage of these security patterns in conjunction with other patterns from other areas or concerns to solve additional security needs (e.g. secure service composition and/or secure service discovery).

We describe hereafter a functional scenario for validation. This scenario is the architecture of the Security PoC we set-up. This architecture choice was designed and shared to demonstrate and validate the two architectural patterns of the Security PoC (i.e. Trusted Timestamping and Non-Repudiation architectural patterns) |

---

\(^{25}\) SaaS : Software as a Service

\(^{26}\) IoS : Internet of Service
SBs = SignatureBroker service
TTSs = Trusted Timestamping service
TDs = Tax Declaration service

0. The user makes available user’s tax declaration as a XML Form to the SBs.
1. The SBs requests the TTSs to provide a time-stamp.
2. The TTSs generates a trusted timestamp, signs and encrypts it and sends back to the SBs.
3. The SBs decrypts the message received from the TTSs, verifies the signature, concatenates the XML Form and the received time-stamp and finally signs and encrypts the resulting content and submits the message to the TDs.

The TDs decrypts the message received from the SBs, verifies the signature and de-concatenates the XML Form and the time-stamp. Then, it verifies that the time-stamp is within the range accepted for a valid tax declaration. It creates a signed and encrypted response for the SBs.

Suggested Architecture

After achieving the security PoC, defining a scenario for the Tax Declaration, we firstly defined the phase of design pattern and secondly the implementation pattern. The following report contains these two phases:

The design of the PoC combines two designs already quoted in two patterns.

For this reason, the following abstract design patterns picture is a composition of “Trusted Timestaping” and “Non-Repudiation” patterns.

Abstract Design Patterns

The abstract design patterns should help an architect to design a solution. This
abstract design pattern is implementable. Because these patterns are distilled from the experiences of experts. They enable you to repeat a successful design done by someone else. By doing so you can stand on the shoulders of the experts and do not have to re-invent the wheel. However, since patterns enable many implementation variations you still have to keep the brain turned on. Finally, since patterns provide you with names for design building blocks they provide you with a vocabulary to describe and discuss a particular design.

Description of each element

Each element will be described in the D8.2a (Report on the proof of concept) and will be also depicted hereafter:

The input is “document + hash".

This hashed document is timestamped by the TSA.

The “hashed document + timestamp” is signed by TSA and then returned to the consumer. In this way, the original document if hashed obtains the same result with “document + hash”. As the hashed document is signed, it proves the signature. And the timestamp is signed; it proves the document is stamped with the indicated time. Only the TSA can encode the timestamp! Only the TSA has the secret key.

In this design pattern, the role played by the TSA is very important. The weakness could be TSA. If the key is stolen, the timestamp could be modified.

The input is a “signed document”.

This signed document is received by the consumer.

He checks in the certificates list found in the Certificate Store.

The certificate can be revoked by the CRL (Certificate Revocation List) or OCSP server (Online Certificate Status Protocol).

He reads the document with the public key found in the certificate.

One difference with respect to the previous session is that it is now possible to verify the signature of the (Document + Timestamp) thanks to "Certificate Store".

The following implementation is one of the possible solutions. The implementation needs the concrete components. Although these components are exposed by open source software. It is possible to use other concrete components. In this case, another list of concrete components would be obtained. The unchanged thing remains the concept viewed in this PoC.
### Environment
The particular choice of components will be:
- Axis2 Web Services Engine
- Rampart (WS-Security and Ws-Policy)
- PKI (simulated in the context of this PoC phase 1)

### Estimated Effort
3 person-month (developing patterns, running the evaluation and assessing it, setup the first draft of the demonstrator: install servers, install software, develop web services; configuration).

### Methods
The two patterns described in the PoC provide the following quality attributes: integrity, authenticity, accountability.

To guarantee the quality of the patterns is met.

It should use a method to measure this quality. By measuring the quality attributes of an architecture, the ATAM is the adequate choice.

In software engineering, ATAM is a risk-mitigation process used early in the software development life cycle. ATAM is most beneficial when done early in the software development life-cycle, when the cost of changing architectures is minimal.

The ATAM process consists of gathering stakeholders together to analyze business drivers and from these drivers extract quality attributes that are used to create scenarios. These scenarios are then used in conjunction with architectural approaches and architectural decisions to create an analysis of trade-offs, sensitivity points, and risks (or non-risks). This analysis can be converted to risk themes and their impacts whereupon the process can be repeated.

In this case, the quality attributes will be measured with the architectural choices. These measures will lighten if the identified quality attributes will be met with the technical solutions.

### Assessment criteria & metrics & way to document
To build a system or an architectural pattern, we need to define three fields: Non-Functional Requirements, Functional Requirements and the Constraints (report to the Rationale section of template pattern specified by WP7).

For the complete way to assess the PoC, the assessment will be done in these three fields.

---

**Figure 137: Implementation design patterns of the PoC**

![Diagram of Implementation Design Patterns]

27 **ATAM**: Architecture Trade-offs Analysis Method. ATAM was developed by the Software Engineering Institute at the Carnegie Mellon University. Its purpose is to help choose a suitable architecture for a software system by discovering trade-offs and sensitivity points.
Firstly, the functional requirements are to be reported with the business requirements of WP10 (especially the D10.1)

Secondly, the Non-functional Requirements (quality attributes) will be measured with the ATAM method.

Thirdly, the Rationale will be assessed with a technical evaluation criteria (these evaluation criteria will be reported in the annexe 3 of D8.2)

Further Information

After implementing these patterns, WP8 understand more the relationship with the Reference Architecture Structure described in the DoW. For this reason, the mapping to this structure can be done and schematised in these following pictures. Each element of the RA structure can be mapped with this PoC

- Concrete components: Rampart
- Abstract component: thanks to Abstract Design Pattern
- Abstract Design Pattern: see suggested architecture
- Implementation Design Pattern: see suggested architecture
- Reference model: reference to the WP4
- Guidelines and principles: best practices of XML signature prescribed by W3C
- Standard catalogue: XML Signature

The mapping can be used in the return of experiences. Of course, this mapping is still imperfect, but there are the feedbacks of the implementation of the PoC in order to go further in the building of the RA structure.

**Figure 138: Non-Repudiation Pattern mapped to RA of D7.4**
3.4 PoC Results Related to Security

3.4.1 Pattern: Trusted Timestamping Pattern

Trusted timestamping is the process to track a document in the time, since its creation, and follow its changes. It is the means guaranteeing that nobody, even the owner, is capable of modifying it, once it has been timestamped, provided that the timestamper's integrity is never compromised. Consequently, this pattern strengthens the trust in the timestamp of a document.

The administrative aspect involves setting up a publicly available, trusted timestamp management infrastructure to collect process and renew timestamps.

3.4.1.1 Scope of the PoC with respect the Reference Architecture

The objectives of the Security PoC are to support the pattern which is specified in the WP7 and to validate the research results of WP4. Starting to populate the NEXOF-RA according to the structure of the reference architecture proposed. As such the figure hereafter gives a global view of Security PoC components to each element/constituent of the NEXOF RA structure.
Figure 140: Scope of the PoC – Trusted Timestamping

Before describing all elements, here is the relationship between abstract components and concrete components.

<table>
<thead>
<tr>
<th>Abstract component</th>
<th>Concrete component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>XML encryption</td>
</tr>
<tr>
<td>TSA</td>
<td>TSA</td>
</tr>
<tr>
<td>Timestamp</td>
<td>Timestamp server</td>
</tr>
<tr>
<td>Digital Signature</td>
<td>XML Signature</td>
</tr>
<tr>
<td>Key Management</td>
<td>Key Managers</td>
</tr>
<tr>
<td>SOA Framework</td>
<td>N/A</td>
</tr>
<tr>
<td>N/A</td>
<td>AXIS 2</td>
</tr>
</tbody>
</table>

The achievement of the PoC has used other components which are necessary in the context but not necessary for the pattern.

The Framework SOA helps the development of this pattern on the environment SOA.

AXIS 2 is used to simulate the consumer.
For the convenience of deployment, the TSA is simulated by a script which is written specifically for the demonstration, and the timestamp server is used through a call of a Java function.

**Description of each element**

WS-Security (Web Services Security) is a communication protocol providing a means for applying security to Web services. On April 19 2004 the WS-Security 1.0 standard was released by Oasis-Open. On February 17 2006 they released version 1.1. In addition with WS-Security, the WS-Security Policy which defines the security policy on the server side is also suggested to be used.

Rampart (see Figure 141 in the simplified architecture) proposes a complete set of concrete components, but some of them are identified in order to achieve the implementation. Axis 2 is a concrete component which plays the role of consumer.

![Image: Complete set of concrete components proposed by RAMPART](image)

**Figure 141: Complete set of concrete components proposed by RAMPART**

In a very simple view, Rampart consists of a core module and packages related to WS-Security and WS-Security Policy. For XML-Encryption and Signature, Rampart/C uses OMXMLSecurity. Both Rampart and OMXMLSecurity use Apache AXIOM\(^2^8\) and Axis2-Util libraries. OpenSSL is used as the crypto library in OMXMLSecurity.

The interface between Rampart and the Apache Axis2 engine is the Rampart module called mod_rampart. The module has two handlers, one for the inflow

---

\(^2^8\) [http://ws.apache.org/axis2/c/docs/om_tutorial.html](http://ws.apache.org/axis2/c/docs/om_tutorial.html)
and the other for the outflow of the Axis2 engine. Rampart directs messages to its other components for further security related processing using these handlers.

Handlers are a way of extending capabilities of the core engine. Once the Axis2 engine calls the invoke() method of the handler, the module can do the necessary processing over the SOAP message. Rampart use this mechanism to build/process security related SOAP headers.

The abstract components are materialized together with the concrete components which participate to their implementation.

The abstract design patterns are represented firstly by the high level of security pattern and followed directly by the pattern. This level to which is attached the two security architectural patterns is secure SOA.

The implementation design patterns are represented with these three main activities (to build a complete chain, we need more details of flow exchange. And then we must integrate more actors on the chain):

- Hash of the document (from the document, a dedicated software calculates a hash)
- The TSA added the timestamp to the hashed document
- TSA signed the document with the secrete key.

The following sequence diagram which is already specified in the WP7 [Trusted Timestamping Pattern] explains how a trusted timestamp is implemented. This diagram show the use of the set of abstract components which are described the previous paragraph.

---

**Diagram: Sequence Diagram**

1. External Consumer Agent
2. Hash
3. Timestamping Authority
4. Timestamping
5. Digital Signature
6. Key Management

---

The abstract components are materialized together with the concrete components which participate to their implementation. The abstract design patterns are represented firstly by the high level of security pattern and followed directly by the pattern. This level to which is attached the two security architectural patterns is secure SOA. The implementation design patterns are represented with these three main activities (to build a complete chain, we need more details of flow exchange. And then we must integrate more actors on the chain):

- Hash of the document (from the document, a dedicated software calculates a hash)
- The TSA added the timestamp to the hashed document
- TSA signed the document with the secrete key.

The following sequence diagram which is already specified in the WP7 [Trusted Timestamping Pattern] explains how a trusted timestamp is implemented. This diagram show the use of the set of abstract components which are described the previous paragraph.
3.4.1.2 **Evaluation methodology**

The pattern is evaluated according to quality attributes it claims. These quality attributes are stated in the filled-in template description of the architectural pattern (according to a shared and agreed template provided by WP7). The stated quality attributes in this architectural pattern description are evaluated using the ATAM approach. Especially the quality attributes utility tree is used to show and analyze the results of the implementation of each quality attribute.

3.4.1.3 **Quality attributes provided to WP7 for this pattern**

The trusted timestamping pattern is specified with the work done on WP4. In the context of specification, this pattern has been identified to support certain defined quality attributes. These quality attributes affect the quality of the architecture where each pattern is integrated. Evaluating these quality attributes permits to know if the resulting architecture where the pattern has been applied meets the expected requirements.

[The significations of plus (+), minus (-) or neutral (0) are given in the specification of this pattern for D7.5]

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity</td>
<td>+</td>
</tr>
<tr>
<td>Accountability</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability</td>
<td>+</td>
</tr>
<tr>
<td>Performance</td>
<td>-</td>
</tr>
</tbody>
</table>

3.4.1.4 **Evaluating quality attributes by using Pattern Quality Attributes Tree**

After identifying the quality attributes, evaluating these quality attributes with the ATAM\(^{29}\) methodology is depicted in the following picture.

---

\(^{29}\) More information on ATAM methodology can be found in [Errore. L'origine riferimento non è stata trovata.](#)
Description of each element

Please report to the ATAM methodology for the details of using ATAM approach. The following description is just a report of results of the analysis.

The aim of integrity is to guarantee that a data is not modified either accidentally or maliciously. To guarantee the integrity, a protocol as Digital Signature is used. The X509 with keys of 1024 bits are possible technical solutions.

In cryptography, X.509 is an ITU-T standard for a public key infrastructure (PKI). X.509 specifies, amongst other things, standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm. Today, the recommended length for keys used by the format X509 certificates is 1024 bits. This key length meets the needs of applications of e-commerce. Its robustness guarantees the integrity of the Trusted Timestamping.

The Accountability enforces imputability, guarantying the transparency of the operations. Time stamping contributes to accountability since it can be used to add information about the time of digital signature. Time stamping relies on the same technology as digital signature. The Time Stamping infrastructure is based on a TSA Trusted third party Timestamp Authority and its X.509
certificates. The timestamp signed by the TSA guarantee the traceability of the time of an action and can be use as a legal proof.

The overall security architecture is based on the SOA. All security components are seen as a service. By this way, they inherit of the SOA benefits: interoperability, loose-coupling. It affects directly positively the maintainability.

In the general security context and particularly for the Digital Signature, the performance is affected negatively. Intrinsically, the Digital Signature uses hash and encryption mechanisms, which are consumers of resources. The encryption will increase the volume of the data which will be transmitted to the network. Furthermore, the data transmission on the network will be large all the more as the volume of data to be transmitted will be sizable.

3.4.1.5 Evaluation results

The Evaluation results are subjective. Quantitative indicators are not applicable in this context/field. We must ask ourselves: how to measure the trust, which itself is subjective and is often related to personal experience or that of a third party in whom we trust.

We can see it with the Digital Signature which guarantees the integrity. Digital Signature uses the certificate which is delivered by the Certification Authority. This signature proves who the author of the data is. This proof is based on the best practices recommended by the RFC 3161 for the trusted timestamping. This process of Trusted Timestamping is supported by two main security activities:

- Cryptography
- Trusted Third Party

The cryptography uses the private key to encrypt the data. This encryption using this key is considered as trust as the key has not been broken.

The quality of a key is its ability to resist to an attack or any practice of codebreaking or cracking the code. While a key is not broken, it may be considered of sufficient quality.

To verify that a message has been signed by a user with his/her secret key and has not been modified by the receiver, we must only know the corresponding
public key. The proof of the non-modification of a message guarantees the message **integrity**.

In cryptography, a Trusted Third Party (TTP) is an entity which facilitates interactions between two parties who have trust in the third party. In TTP models', the relying parties use this trust to secure their own interactions. In the Trusted Timestamping process, a certificate authority (CA) generates a digital identity certificate used by the TSA (Timestamping Authority). The CA then becomes the Trusted-Third-Party in this certificates issuance. To this end, the TSA is also a TTP.

*We can have more or less confidence. This perception is subjective, proven or not proven by facts, but there is no unit of measurement to evaluate it. We can just position it on a scale of values that we ourselves graduated according to the idea that we do trust.*

The proof of this identity permits to guarantee the traceability that supports the **Accountability** quality attribute.

**Computational cost and best practices (Performance)**

The asymmetric algorithms known thus far are relatively big resources consumers compared with most symmetric key algorithms. Therefore they are mainly used to protect secret keys or encryption of small amounts of data, such as a hash.

Although symmetric encryption is faster than asymmetric encryption, the last is more secure. The programmer chooses what to use and must consider both speed and security. Large amounts of data are difficult to encrypt by using asymmetric key encryption due to the performance overhead. One other major problem with this type of encryption is key management. In many organizations, a public key infrastructure is implemented and used for revoking, distributing, and managing certificates.

This study can be done with the real need of performance.

In this case, the balance with different parameters must be trade-offed:

- number of bits to use to encrypt the data,
- cryptographic algorithm,
- type of processor (math support),
- asymmetric, symmetric, key session?

In the security PoC, this study is not done because the costs of it are important according to the objectives.

Concerning the **Maintainability**, all parties are communicated within the SOA environment. In fact, the standard interfaces defined for this purpose permits to
be interoperable with other services. It means that we can change a service if it doesn’t correspond to our needs without changing the interface with other services.

**Determination of Sensitivity Points**

After evaluating subjectively the results, we ought to determine the sensitivity point of TTP (Trusted Third Party) or of Cryptographic Key.

![Diagram of Sensitivity Point](image)

**Figure 144: Example of sensitivity point of the confidence of TTP or of a cryptographic key**

The above picture shows that the level of confidence on the TTP or on the cryptographic key. Once, we lost the trust on the TTP or on the cryptographic key for many reasons.

For example, a cryptographic key could be breakable according to the means deployed and time spent to break it. To this moment, we can trust a cryptographic key for a given time-to-live period after which the key must be replaced. The moment of lost of the confidence should determine the moment of the stop of use. The above picture is valuable for the both quality attribute: integrity and accountability [Figure 143].

**Determination of trade-off points of TTP or Cryptographic key**

The trade-off point of TTP is very difficult to express with a measurable rule. The decision to take a TTP or another TTP depends of many criteria such as security policy of the enterprise, the law, the trust, the number of customers of the TTP…

The trade-off point a cryptographic key depends of the following parameters (we choose a cryptographic key indeed of another):
Asymmetric
Symmetric
Computation time
Strength
Confidence
Etc…

We can take an example of comparison between two different algorithms. One time, we are confident on these keys. This example shows only how we can balance between these algorithms (asymmetric and symmetric). According to the security expertise, the Asymmetric algorithm is slower with the big volume. This example is simplified and reduced to two algorithms. In certain cases, it’s more complex to find the trade-off. Ease of key distribution should be taken into account. All the time, the choice of a security algorithm is defined according to the security needs.

![Diagram](image)

Figure 145: Example of trade-off point of a cryptographic key

This example is a concrete determination of the trade-off of the encryption algorithm.

3.4.1.6 Conclusion
The ATAM approach by using the quality attributes pattern tree highlights clearly how the quality attributes have reached their target. It means that this approach supports the specification of the quality attributes inside the Trusted Timestamping pattern.

The subjective approach permits at least to determine the sensitivity and trade-off points. As consequence of the evaluation based on the subjectivity, these points are only the approximate data.
The achievement of this pattern brings a high added value for the trust and confidence requirement [R33] of D10.1. There isn’t any major risk of the implementation to state here.
3.4.2 Pattern: Non-Repudiation Pattern

Non-repudiation is the concept of ensuring that a party in a dispute cannot repudiate, or refute the validity of an information, statement or contract. Although this concept can be applied to any transmission, including television and radio, by far the most common application is in the verification and trust of signatures.

According to traditional legal practice, a signature on a paper contract or memorandum is sometimes challenged by the signatory. Such contesting may take one of two forms: The signatory may claim fraud or forgery, such as “I did not sign that.” Alternately, he/she may accept the signature as authentic but dispute its validity due to coercion, as in the scenario of blackmail or confessions given under torture. In the latter case the digital signature does nothing more.

The legal burden of proof differs depending upon the repudiation reason. In the former scenario the burden of proof typically rests on the party claiming validity, while in the latter it shifts to the signatory claiming lack thereof.

3.4.2.1 Scope of the PoC with respect the Reference Architecture

The main objective of the PoC is to achieve quality attributes which are specified in WP7. By achieving our quality attributes of the trusted timestamping pattern, we can map this pattern to Reference Architecture Structure.

![Diagram](image.png)

**Figure 146: Scope of the PoC – Non-Repudiation**
Before describing all elements, here is the relationship between abstract components and concrete components.

<table>
<thead>
<tr>
<th>Abstract component</th>
<th>Concrete component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>XML encryption</td>
</tr>
<tr>
<td>Digital Signature</td>
<td>XML Signature</td>
</tr>
<tr>
<td>Key Management</td>
<td>Key Managers</td>
</tr>
<tr>
<td>SOA Framework</td>
<td>N/A</td>
</tr>
<tr>
<td>N/A</td>
<td>AXIS 2</td>
</tr>
<tr>
<td>Certificate Store</td>
<td>Contains in the Web Browser (IE, Firefox…)</td>
</tr>
<tr>
<td>CRL(^{30})</td>
<td>CRL</td>
</tr>
<tr>
<td>OCSP(^{31})</td>
<td>OCSP</td>
</tr>
</tbody>
</table>

The achievement of the PoC has used other components which are necessary in the context but not necessary for the pattern.

The Framework SOA helps the development of this pattern on the environment SOA.

AXIS 2 is used to simulate the consumer.

For the convenient of deployment and the context of the demonstration, the CRL or OSCP are simulated by a script.

**Description of each element**

XML Signature (also called XMLDsig, XML-DSig, XML-Sig) is a W3C recommendation that defines an XML syntax for digital signatures. Functionally, it has much in common with PKCS#7\(^{32}\) but is more extensible and geared towards signing XML documents. It is used by various Web technologies such as SOAP, SAML, and others.

XML signatures can be used to sign data—a resource—of any type, typically XML documents, but anything that is accessible via a URL can be signed. An XML signature used to sign a resource outside its containing XML document is called a detached signature; if it is used to sign some part of its containing document, it is called an enveloped signature; if it contains the signed data within itself it is called an enveloping signature.

---

\(^{30}\) CRL: Certificate Revocation List

\(^{31}\) OCSP: Online Certificate Status Protocol

\(^{32}\) Public Key Cryptographic Standards (#7, it means version 1.5),
This pattern is primarily aimed to be applied to SoS using SOA.

Rampart proposed other concrete components, but these components are identified in order to achieve the implementation. Axis 2 plays a role of consumer in the use case.

The abstract components are identified in accordance with the concrete components. The abstract components participate to build abstract design patterns.

The abstract design patterns are represented firstly by the high level of the security pattern and followed directly by the pattern. This level indicates that we want to secure the SOA environment. To build this objective, this pattern furnishes a service to secure this environment. We will describe more details in the next session.

The implementation design patterns are represented with these three main activities (to build a complete chain, we need more details of flow exchange. And then the integration of the main actors on the chain is presented below:

- The document signed with the secret key
- With the public key found in the certificate, the receiver can read the document

He can also verify the signature.

The following sequence diagram which is already specified in the WP7 [Non-Repudiation Pattern] explains how a Digital Signature is implemented. This diagram shows the use of the set of abstract components which are described the previous paragraph.
Figure 147: Verification of a signature sequence diagram – use of the set of components

3.4.2.2 Evaluation methodology

The pattern is evaluated according to quality attributes it claims. These quality attributes are stated in the filled-in template description of the architectural pattern (according to a shared and agreed template provided by WP7). The stated quality attributes in this architectural pattern description are evaluated using the ATAM approach. Especially the quality attributes utility tree is used to show and analyze the results of the implementation of each quality attribute.

3.4.2.3 Quality attributes provided to WP7 for this pattern

The non-repudiation pattern is specified with the work done on WP4. In the context of specification, this pattern has been identified to support certain defined quality attributes. These quality attributes affect the quality of the architecture where each pattern is integrated. Evaluating these quality attributes permits to know if the resulting architecture where the pattern has been applied meets the expected requirements.

[The significations of plus (+), minus (-) or neutral (0) are given in the specification of this pattern for D7.5]

<table>
<thead>
<tr>
<th>Integrity</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticity</td>
<td>+</td>
</tr>
</tbody>
</table>
3.4.2.4 Evaluating quality attributes by using Quality Attributes Pattern Tree

After identifying the quality attributes, evaluating these quality attributes with the ATAM methodology is depicted in the following picture.

![Quality Attributes Pattern Tree](image)

**Figure 148: Evaluating quality attributes of Non-Repudiation pattern with ATAM Approach**

**Description of each element**

Please report to the ATAM methodology for the details of using ATAM approach. The following description is just a report of results of the analysis.

The aim of integrity is to guarantee that a data is not modified either accidentally or maliciously. To guarantee the integrity, a protocol as Digital Signature is used. The X509 and 1024 bits are possible technical solutions.

In cryptography, X.509 is an ITU-T standard for a public key infrastructure (PKI). X.509 specifies, amongst other things, standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm. Today, the recommended length for keys used by the format X509 certificates is 1024 bits. This key length meets the needs of...
applications of e-commerce. Its robustness guarantees the integrity of the Non-repudiation pattern.

A data is considered as authentic when it can be used as a legal proof. To guarantee the authenticity, the X.509 certificate is signed and revocable by the Authority Certificate (the trusted third party). The quality of authenticity leans directly on the Authority Certificate.

The use of Digital Signature can bring a proof of the timestamp. The Digital Signature is used for the authenticity. It assures that nobody can modify the data or the service.

The overall security architecture is based on the SOA. All security components are seen as a service. By this way, they inherit the benefits of SOA: interoperability, loose-coupling. It affects directly positively the maintainability.

Although symmetric encryption is faster than asymmetric encryption, the last is more secure. The programmer chooses what to use and must consider both speed and security. Large amounts of data are difficult to encrypt by using asymmetric key encryption due to the performance overhead. Another major problem with this type of encryption is key management. In many organizations, a public key infrastructure is implemented and used for revoking, distributing, and managing certificates.

### 3.4.2.5 Evaluation results

The same assessment for the **Integrity**, **Performance** and **Maintainability** of the trusted timestamping pattern [3.4.2.5 Evaluation results]

**Authenticity.** A subject is considered as authentic when it can be used as a legal proof. This might involve confirming the identity of a person, tracing the origins of an artefact, ensuring that a product is what its packaging and labelling claims to be, or assuring that a computer program is a trusted one. In the case of the PoC, the certificate is used to guarantee the identity of the message sender. Of course, the trust of this identity depends on the trust that we have on the TTP.

In fact, the trust on the identity doesn’t depend only of the trade-off but also of the context (use case). Actually, in authentication context, fingerprint spoof detection using blood-flow analysis is one of the best to prove the identity. In other words, the context of the TSA should enable us to use the recommendation of the best practices which are described in the RFC 3161.

### 3.4.2.6 Conclusion

Same conclusion at the previous section [3.4.1.6]
4 Open Process

This section describes the open process to collect patterns on different topics and the results of the second call of this open process.

4.1 Proposed Topics.

4.1.1 Topic 4.5 Multilevel security for SOA.

4.1.1.1 Rational

Security is a significant challenge for Service-Oriented-Architectures (SOA) in a multi-domain environment. Security incorporates the concept of Multi-Level Security (MLS). MLS has been until recently a niche market with only a few government agencies needing it. In recent years, there has been an emphasized need for multiple government agencies to share information on a “need-to-know” basis. Supporting MLS in large scale distributed enterprise systems becomes a critical requirement for enterprise collaborations (Intranet, Extranet and others).

This topic contributes to resolve confidentiality and trust issues and complements the consortium expertise on this thematic. Criteria to decide when to issue the call refers to progress state in architecture reference model (as soon as possible to introduce them in the NEXOF-RA model). Any patterns will contribute directly to NEXOF-RA models, others will be used in the architecture implementations security.

4.1.1.2 ItC Text

<table>
<thead>
<tr>
<th>Contact</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal Bisson (Thales) – <a href="mailto:pascal.bisson@thalesgroup.com">pascal.bisson@thalesgroup.com</a></td>
<td></td>
</tr>
<tr>
<td>Daniel Gidoin (Thales) – <a href="mailto:daniel.gidoin@thalesgroup.com">daniel.gidoin@thalesgroup.com</a></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>This call addresses the area of the multilevel security for SOA. Service-oriented architectures are dynamic, flexible and compositional in nature. Security is a significant challenge for Service-Oriented-Architectures (SOA) in a multi-domain environment. Security incorporates the concept of Multi-Level Security (MLS). MLS has been until recently a niche market with only a few government agencies needing it. However, in recent years, In recent years, there has been an emphasized need for multiple government agencies to share information on a need-to-know basis, hence there is a government push to migrate the existing isolated MLS infrastructures to a single integrated MLS infrastructure. Therefore, supporting MLS in large scale distributed enterprise systems becomes an urgent and critical requirement for intra- and inter-enterprise collaborations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>The MLS concept was originally described in the DoD Orange Book on the</td>
</tr>
</tbody>
</table>
needs of common evaluation criteria. At that time, there was almost no concept of distributed computing, web services, policy management or metadata technologies.

An MLS system is supposed to operate as follows: All resources are assigned a security label denoting the sensitivity of the resource; Users are issued security clearances denoting their trustworthiness and the types of information they need to know; Mandatory access control compares each user’s clearance with each resource’s label before access is granted.

With the advancement of technologies such as web services, SOA, ontology and the deployment of networks, to achieve MLS in a distributed computing environment today, in reality, it must do the following: Provide mechanisms at the hosts and network nodes to enable security services at each specified classification level; Provide the ability to enforce accountability by logging an audit trail of all events; Guarantee impenetrable barrier between treatments, services and information of different levels of sensitivity, according to security classification.

**Scope**

The scope is multilevel security for SOA.

The aims of this call are to provide solutions for making implementable and deployable improvements to the usability of multilevel security.

The main control functions expected of an MLS system includes:

- **Access control.** This is accomplished through the use of access control lists that identify the users that can access a given resource (service, data.. and their level of authority with regard to that resource;

- **Auditing:** Audit records associate security-related events (such as file access) with the user that caused the event;

- **Name-hiding:** The names of files, data sets and directories are only displayed to users with access authority. Users without a need-to-know will not see the file or object listed or displayed;

- **Write-down prevention:** To prevent users from declassifying data, in order to grant access to users without a need-to-know and or of lower level of classification.

With the availability of the SOA and the capabilities of applications in today’s distributed computing environment, there are other low level functions required in order to ensure a true single integrated MLS system environment. An MLS system design should achieve the following goals:

- **To establish controls that prevent users from accessing information at a higher classification than their authorization permits;**

- **To ensure that the controls prevent unauthorized users from declassifying information.**

- **To enable information on an as-needed basis among multiple
administrative domains.

Recommendations: To propose MLS having realistic implementation plans and budgets.

Contributions

MLS can be one of the services provided in a SOA environment. The MLS service requires the deployment of security mechanisms at different layers. So, because of the maturity of technologies, we think to expand to support MLS in a SOA environment is very feasible, in particular for the following reasons:

- Availability of standard web-services interfaces, languages and protocols;
- Almost all security mechanisms are standard based implementations;
- Availability of MLS functions. In recent years open-source operating systems are ideal platforms for implementing MLS.

The key components of the MLS architecture and expected contributions concerns:

- Integration of diverse MLS services and tools into the architecture;
- Establishment Service. It interfaces with a policy manager to determine if the user is authorized to request the requested classification level. If yes, it then gets the security resources requirements and determines what actions is permitted for each object state given the user's security classification and need-to-know;
- Security Configuration Service: It matches the security resources requirements and the configuration of security infrastructure that is involved.

Baseline

The baseline is composed of web services standards (OASIS..), J2EE, technologies in the areas of dynamic configuration management, object metadata model, rules execution tools.

For further information

Visit web page.

4.1.2 Topic 4.6 Dynamic security in SOA.

4.1.2.1 Rational

The concepts of system and application as we know them today will disappear, evolving from static architectures with well-defined pieces of hardware, software, communication links, limits and owners, to architectures that will be sensitive, adaptive, context-aware and responsive to users' needs and habits. These Am1 ecosystems will offer highly distributed dynamic services in environments that will be heterogeneous, large scale and nomadic, where computing nodes will be omnipresent and communications infrastructures will be dynamically assembled. The combination of heterogeneity, mobility, dynamism, sheer number of devices, along with the growing demands for security, is going to make the security provision for these ecosystems
increasingly difficult to achieve with existing security solutions, engineering approaches and tools (Reference: Serenity project)

This topic contributes to resolve security issues and complements the consortium expertise on this thematic. Criteria to decide when to issue the call refers to progress state in architecture reference model (as soon as possible to introduce them in the NEXOF-RA model). Any patterns will contribute directly to NEXOF-RA models, others will be used in the architecture iomplementations security.

4.1.2.2 ItC Text

<table>
<thead>
<tr>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal Bisson (Thales) – <a href="mailto:pascal.bisson@thalesgroup.com">pascal.bisson@thalesgroup.com</a></td>
</tr>
<tr>
<td>Daniel Gidoin (Thales) – <a href="mailto:daniel.gidoin@thalesgroup.com">daniel.gidoin@thalesgroup.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>This call addresses the area of dynamic security for SOA.</td>
</tr>
<tr>
<td>The evolution of dynamic execution environments increasingly requires security policies that are also dynamic in nature to address such events as process migration, changes in personnel, shifts in alliances, and detected intrusion that cannot be well anticipated or addressed by static policies.</td>
</tr>
<tr>
<td>In addition, Web Services (WS) will play a significant role in the next generation Web. However, the attractive features of WS such as platform independence, XML and SOAP reliance, and simplicity to use, make them vulnerable to many security threats including new unexplored and inherited old problems.</td>
</tr>
<tr>
<td>As well, dynamic separation of duties, delegation and other dynamic security constraints requires the state of the security system to be managed explicitly at run-time. So, dynamic segregation of duties constraints are a form of history-based access control. The permission for an actor to take certain actions in some context will depend on him/her not having performed related actions in that same context already. For example, a clerk may be authorised to sign or countersign a given cheque, but a single clerk is not authorised to carry out both actions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>The evolution of dynamic execution environments and dynamic security adaptability requires architectures that adapt to changing security policies during runtime with minimal loss of functionality and with little or no manual assistance.</td>
</tr>
<tr>
<td>In this context, we must take into account the ability to reconfigure the global security policy at any time to address events such as shifts in alliances, changes in personnel, execution environment (e.g. transition from trusted execution environment to untrustworthiness execution environment), crisis situations…</td>
</tr>
</tbody>
</table>
Security agility is a software flexibility technique to address security properties and their dynamic evolution. An agile software component is cognizant of the security environment in which it executes, is aware of its responsibilities for enforcing “its part” of a more global policy, and contains internal mechanisms that adapt its functionality in coordination with authorized external policy changes. The heterogeneous nature of a dynamic execution environment presents some significant obstacles to developing dynamic security. The first such impediment is the wide range of possible security semantics. A variety of access control policies might be employed, for example, including information disclosure policies, role based policies.

The variety of architecture components employed in a heterogeneous and or ubiquitous environment, including operating systems, system software, and mission-specific software require the development of flexible techniques that are not bound to a single environment.

Scope
To help overcome the heterogeneity obstacles, dynamic security techniques can employ various strategies to address security properties and their dynamic evolution. For example, they can embed components with pre-formulated security policy models (security patterns) and mechanisms to provide policy awareness in support of security reconfiguration. They can also provide a flexible component architecture that allows dynamic code extensions for adding new security semantics or changing security-relevant behaviour to maintain compatibility with new security rules or execution environment.

Contributions
The contributions can take different shapes: 1).security policies and dynamic models. Security policy models allow a component to be aware of the security policy governing their operation. 2). Policy components awareness. One important consideration in dynamic execution environments is the likely fluctuation in resource availability resulting from security policy reconfiguration. When processes find themselves unable to access resources they expect will always be available, undesirable events may occur, including termination of critical processes. Process dependencies, such as client/server relationships, are often subtle, or even unknown, and may be overlooked when dynamic policy changes are implemented, particularly in time-critical situations such as in response to an intrusion detection event. 3) Dynamic security architecture (model, mechanisms, patterns, components, toolkits). Once components have been made aware of the security policy changes that could affect their execution, they can be extended with adaptive functionality that reacts positively to these changes, rather than failing in some manner. Positive responses might include terminating connections invalidated by policy changes, temporarily suspending or reducing normal operations until lost resources become available once again, reacquiring lost resources, or switching to alternate algorithms to produce equivalent results. The dynamic security toolkit architecture facilitates the coordination of policy awareness and adaptive behavior functionality.
Baseline
The baseline is composed of web services standards (W3C, OASIS), J2EE, and the standards from the service privacy forum.

For further information
Visit web page

4.1.3 Topic 4.7 Service Level Agreements (SLAs) and Quality of Service (QoS).

4.1.3.1 Rational
The motivation of this topic is basically to have full coverage of all non-functional aspects in the scope of WP4. The ItC aimed at receiving all kinds of contributions related to SLA management and QoS monitoring. The motivation for calling this topic in the second call was to allow the main expected contributor, SLA@SOI, an NSP to progress sufficiently so it could contribute substantially.

4.1.3.2 ItC Text

<table>
<thead>
<tr>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricardo Jiménez-Peris - <a href="mailto:rjimenez@fi.upm.es">rjimenez@fi.upm.es</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic covers two issues. The first one is service level agreements (SLAs) in all its aspects including SLA description, SLA translation, SLA monitoring, and SLA negotiation. The second issue covered by the topic is how to guarantee and/or enforce quality of service (QoS) to satisfy requirements coming from SLAs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ItC focuses on SLAs and how to enforce QoS requirement derived from the SLAs. On the SLA side expected contributions include languages for describing SLAs, approaches to translate SLAs, architectures to monitor SLAs, interfaces, protocols, and standards to negotiate SLAs, etc. On the QoS side contributions are expected to concentrate on how to enforce QoS requirements set by SLAs on regular and large scale systems (e.g. in cloud computing).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contributions on all aspects of SLAs and on how to enforce QoS requirements derived by SLAs are welcomed. QoS contributions without a link to SLAs are not in scope.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural contributions for both SLAs and QoS are expected to take the shape of architectural patterns. Contributions regarding SLA description are</td>
</tr>
</tbody>
</table>
expected to be in the form of languages for describing SLAs. Also descriptions of interfaces are welcomed for SLA negotiation and SLA monitoring.

**Baseline**
The baseline is SOI in any shape either traditional (multi-tier) or more innovative (SaaS and cloud computing).

**For further information**
Visit web page

### 4.1.4 Topic 4.8 Federated and Autonomic Management in SOA

#### 4.1.4.1 Rational
The motivation of this topic is basically to have full coverage of all non-functional aspects in the scope of WP4. The ItC aimed at receiving all kinds of contributions related to federated and autonomic management. The motivation for calling this topic in the second call was to allow running FP7 projects to progress and have higher opportunities to contribute to the topic.

#### 4.1.4.2 ItC Text

**Contact**
Ricardo Jiménez-Peris rjimenez@fi.upm.es

**Overview**
This topic focuses on two main issues. First, how the management of different service infrastructures can be federated to obtain a holistic management of the whole service infrastructure (including the federation of the management virtualization infrastructures). Second, how to enrich SOA management with autonomic capabilities to obtain self-management (self-healing, self-provisioning, self-optimization, self-configuration).

**Problem Statement**
The first issue lies in how given different service infrastructures used in combination to support a particular set of services how can be managed as a single logical entity by federating the individual management of the different infrastructures. For instance, in a multi-tier architecture with web, application and database tiers how the management of the three tiers can be federated to obtain a holistic management.

The second issue consists in how to incorporate autonomic capabilities into the management of service infrastructures. More concretely how to obtain self-* properties for SOI such as self-healing, self-provisioning, self-configuration, and self-optimization.

**Scope**
The scope is on federated and autonomic management. Regular management
such as Life Cycle Management is out of scope since it will be covered by a specific ItC.

**Contributions**
The contributions on federated management are expected to take the form of architectural patterns and/or management interfaces to enable federation. The contributions on self-management are expected to take the form of architectural patterns for generic and concrete SOI.

**Baseline**
The baseline lies in current approaches to SOI, either traditional ones such as multi-tier architectures or newer ones such as SaaS and cloud computing.

**For further information**
Visit web page

### 4.2 Kick off of Call2

The following figure presents the participants in the different topics proposed for Call 2:

#### Outcome Call 2
- SLA and QoS.
- Federated and Autonomic Management in SOA.
- Dynamic Security in SOA and SOI.
- Multi-level Security in Interconnected Systems.

The participation in the topics was quite successful, especially the one on SLAs due to the strong participation from SLA@SOI partners. The IT activity is currently suspended after working for one month to wait to get a first version of the reference architecture materialized with a sufficient set of populated patterns in which to plug in the IT contributions. This strategy will enable to a faster integration of IT contributions since they will be adapted to the already existing top level patterns. This approach is also in line with the project extension to enable the integration of call 2 IT contributions.
5 Appendix

5.1 Contribution on Terms to the Glossary

WP4 has contributed a set of terms to the glossary on the following areas: general terms, availability and scalability and security.

5.1.1 General Terms (UPM)

- **Applicability**: In the context of software architectural patterns refers to the number and strength of assumptions that must be taken into account in order to apply the pattern. For example, if it is required the source code of one or several components in order to implement an architectural solution using the pattern. This is an important quality attribute for non-functional patterns related to high availability and scalability. The system architects can use it in order to decide among different architectural patterns (solutions), analyzing how this attribute affects the system requirements with regard to the trade-offs identified in the different patterns under consideration.

- **Application Server**: It refers to a middleware component that runs applications and services. This is an important architectural component that arises in the Multi-Tier Transactional Service Runtime pattern. This pattern separates the data storage functionality (performed by the Database component) from the business logic of applications and services that uses those data and which runs in the application server.

- **Database**: It refers to a data storage component that uses transactional features to guarantee data integrity. It can be a relational database, an object-oriented database, a data warehouse, a data mart, etc. This is an important architectural component that arises in the Multi-Tier Transactional Service Runtime pattern. This pattern separates the data storage functionality (performed by the Database component) from the business logic of applications and services that uses those data and which runs in the Application Server.

- **Maintainability**: In the context of software architectural patterns refers to the number of components that is required to modify and maintain in order to apply the pattern. For example, when talking about database replication, whether it is required to modify only the database kernel source code or if additionally, it is necessary to introduce new components in order to implement an architectural solution using the pattern. This is an important quality attribute for non-functional patterns related to high availability and scalability. The system architects can use it in order to decide among different architectural patterns (solutions), analyzing how this attribute affects the system requirements with regard to the trade-offs identified in the different patterns under consideration.
5.1.2 Terms on Availability and Scalability (UPM)

1. **Database Replication Middleware:** It is a middleware component - interposed between the database and its clients - that contains the Replication Logic that allows achieving high availability and/or scalability of data. This is an important component for non-functional patterns related to database replication. This component is placed out-of-the-box of the database management system and allows implementing replication solutions with null or negligible impact on the database kernel. This term has been adopted from database terminology.

2. **Failover:** Is the automatic process of a system that allows to redirect the control/requests to a redundant replica when a failure occurs in the current one. This is a well-known term that has been adopted from distributed systems terminology. Failover is an important process taken into consideration in all the patterns that involve data/state replication to achieve high-available solutions.

3. **Replication Logic:** A term used to refer to the algorithms/protocols related to replication. This is a concept adopted from distributed systems terminology. The replication logic is present in all the patterns that involve data/state replication to achieve high-available and/or scalable solutions.

4. **Replication Transparency:** A term used to refer to the ability of a replication solution to hide the clients the underlying replication process and possible Failovers. This is a well-known term that has been adopted from distributed systems terminology. Replication transparency is a desirable feature in any replicated system. It is taken into consideration in all the patterns that involve data/state replication to achieve high-available and/or scalable solutions.

5. **Writeset:** A term used to refer to changes on data performed the context of a particular transaction. It is a common term when talking about data replication and has been adopted from database terminology. The writeset it is necessary to identify the changes that each transaction produces in a particular replica in those patterns related data replication. Writesets are extracted from the original replica that processes the transaction and injected in the other of the replicas after being replicated.

5.1.3 Terms on Security (Thales)

1. **Administrative Domain:** An environment or context that is defined by some combination of one or more administrative policies, Internet Domain Name registrations, civil legal entities (for example, individuals, corporations, or other formally organized entities), plus a collection of hosts, network devices and the interconnecting networks (and possibly other traits), plus (often various) network services and applications running upon them. An administrative domain may contain or define one or more security domains. An administrative domain may encompass a single site or multiple sites. The traits defining an administrative domain may, and in many cases will, evolve over time. Administrative domains
may interact and enter into agreements for providing and/or consuming services across administrative domain boundaries.

2. **Anonymity:** Anonymity of a subject means that the subject is not identifiable within a set of subjects, the anonymity set.

3. **Applicable Policies:** The set of policies and policy sets that governs access for a specific decision request.

4. **Assertion:** A piece of data produced by a Security Assertion Markup Language (SAML) authority regarding either an act of authentication performed on a subject, attribute information about the subject, or authorization permissions applying to the subject with respect to a specified resource.

5. **Asserting Party:** Formally, the administrative domain that hosts one or more Security SAML authorities. Informally, an instance of a SAML authority.

6. **Asymmetric Keys:** Two related keys, a public key and a private key that are used to perform complementary operations, such as encryption and decryption or signature generation and signature verification.

7. **Attacker:** Any individual who is attempting to subvert the operation of the biometric system. The intention may be either to subsequently gain illegal entry to the portal or to deny entry to legitimate users.

8. **Attribute:** A single, specific piece of information. An example of an identity attribute is a name or date of birth. A distinct characteristic of an object (in SAML, of a subject). An object’s attributes are said to describe it. Attributes are often specified in terms of physical traits, such as size, shape, weight, and color, etc., for real-world objects. Objects in cyberspace might have attributes describing size, type of encoding, network address, and so on. Which attributes of an object are salient is decided by the beholder.

9. **Attribute Assertion:** An assertion that conveys information about attributes of a subject.

10. **Attribute Authority:** A system entity that produces attribute assertions.

11. **Authentication Assertion:** An assertion that conveys information about a successful act of authentication that took place for a subject.

12. **Authentication Authority:** A system entity that produces authentication assertions.

13. **Authentication Protocol:** A well specified message exchange process that verifies possession of a token to remotely authenticate a claimant. Some authentication protocols also generate cryptographic keys that are used to protect an entire session, so that the data transferred in the session is cryptographically protected.

14. **Authentication Service Component:** A federated architecture that leverages credentials from multiple domains through certifications,
guidelines, standards adoption and policies. The Authentication Service Component (ASC) accommodates assertion-based authentication (i.e., authentication of PINs and Passwords) and certificate-based authentication (i.e., public key certificates) within the same environment. Over time, the architecture will leverage multiple emerging schemes and will not be built around a single scheme or commercial product. In this light, the ASC is more precisely defined as an architectural framework.

15. Authentication Session: Period of time that an end user remains trusted after authentication. Typically a credential service does not require an end user to re-authenticate for every page requested.

16. Authentication Decision: The result of evaluating applicable policy, returned by the PDP (Policy Decision Point) to the PEP (Policy Enforcement Point). A function that evaluates to “Permit”, “Deny”, “Indeterminate” or “NotApplicable”, and (optionally) a set of obligations.

17. Biometric: A measurable biological or behavioral characteristic, which reliably distinguishes one person from another, used to recognize the identity, or verify the claimed identity, of an enrollee.

18. Biometric Data: The extracted information taken from the biometric sample and used either to build a reference template or to compare against a previously created reference template.


20. Biometric Sample: Captured data that represents a biometric characteristic of a user of a biometric system.

21. Canonical Form: The complete, unambiguous and unique encoding of an abstract value obtained by the application of encoding rules that allow one and only one way to encode the abstract value.

22. Certification Authority: A Certification Authority (CA) is the body in a network that issues and manages security credentials and public keys for message encryption. As part of a public key infrastructure (PKI), a CA checks with a Registration Authority (RA) to verify information provided by the requestor of a digital certificate. If the RA verifies the requestor's information, the CA can then issue a certificate. Depending on the public key infrastructure implementation, the certificate includes the owner's public key, the expiration date of the certificate, the owner's name, and other information about the public key owner.

23. Certificate Policy: X.509 Certificate Policy profiles the format and semantics of certificates and certificate revocation lists (CRLs) for the Internet PKI.

24. Certificate Validation: Whenever a certificate is to be trusted, a check is conducted to ensure it is not revoked, expired, or otherwise invalid.
25. **Claim:** A claim is a declaration made by an entity (e.g. name, identity, key, group, privilege, capability, etc).

26. **Claimant:** A party whose identity is to be verified using an authentication protocol.

27. **Credential Assessment Profile:** A list of related criteria used to assess the Assurance Level of a Credential Service.

28. **Credential Service:** A service from Cryptographic Service Provider (CSP) that provides electronic credentials to subscribers for use in electronic transactions. If a CSP offers more than one type of credential, then each one is considered a separate Credential Service (CS).

29. **Credential Service Provider:** A trusted entity that registers, creates, issues and administers identity tokens and electronic credentials to subscribers. The CSP may encompass Registration Authorities and verifiers that it operates. A CSP may be an independent third party such as a bank, educational institution or insurance company or a Government agency. The entity may issue credentials for its own use and extend for use on government online services.

30. **Cryptographic Key:** A value used to control cryptographic operations, such as decryption, encryption, signature generation or signature verification.

31. **Cryptographic Token:** A token where the secret is a cryptographic key.

32. **Cryptography:** The discipline which embodies principles, means and methods for the transformation of data to hide its information content, prevent its undetected modification; prevent its unauthorized use or a combination thereof. Cryptography deals with the transformation of ordinary text (plaintext) into coded form (ciphertext) by encryption and transformation of ciphertext into plaintext by decryption.

33. **Decision (Rule, Policy):** The result of evaluating a rule, policy or policy set.

34. **Decision Request (Authorization):** The request by a PEP (Policy Enforcement Point) to a PDP (Policy Decision Point) to render an authorization decision.

35. **Digital Identity:** Digital identity denotes attribution of attributes to an individual person, which are immediately operationally accessible by technical means. More to the point, the identifier of a digital partial identity can be a simple e-mail address in a news group or a mailing list. Its owner will attain a certain reputation.

36. **Direct Brokered Trust:** Direct Brokered Trust is when one party trusts a second party who, in turn, trusts or vouches for, a third party.

37. **Direct Trust:** Direct trust is when a relying party accepts as true all (or some subset of) the claims in the token sent by the requestor.
38. **Effect (Rule):** The intended consequence of a satisfied rule (either "Permit" or "Deny").

39. **End-to-end Message Level Security:** End-to-end message level security is established when a message that traverses multiple applications (one or more SOAP intermediaries) within and between business entities, e.g. companies, divisions and business units, is secure over its full route through and between those business entities. This includes not only messages that are initiated within the entity but also those messages that originate outside the entity, whether they are Web Services or the more traditional messages.

40. **Enrollee:** A person who has a biometric reference template stored in a biometric system.

41. **Environment Authorization Decision:** The set of attributes that are relevant to an authorization decision and are independent of a particular subject, resource or action.

42. **Hash:** A mathematical function which evenly and randomly distributes values from a large domain into a smaller range.

43. **HMAC:** A mechanism for message authentication using a cryptographic hash function and a specific key.

44. **Identifiability:** Identifiability of a subject from an attacker’s perspective means that the attacker can sufficiently identify the subject within a set of subjects, the identifiability set.

45. **Identifier:** A representation (for example, a string) mapped to a system entity that uniquely refers to it.

46. **Identity:** A unique name of an individual person. Since the legal names of persons are not necessarily unique, the identity of a person must include sufficient additional information (for example an address, or some unique identifier such as an employee or account number) to make the complete name unique. An identity is any subset of attributes of an individual person which sufficiently identifies this individual person within any set of persons.

47. **Identity Management:** Identity management means managing various partial identities (usually denoted by pseudonyms) of an individual person, i.e., administration of identity attributes including the development and choice of the partial identity and pseudonym to be (re-)used in a specific context or role.

48. **Identity Management System:** An identity management system in its broadest sense refers to technology-based administration of identity attributes including the development and choice of the partial identity and pseudonym to be (re-) used in a specific context or role.

49. **Identity Mapping:** Identity Mapping is a method of creating relationships between digital identities or attributes associated with an individual principal by different Identity or Service Providers.
Identity Proofing: The process by which a Cryptographic Service Provider (CSP) and an RA validate sufficient information to uniquely identify a person.

Identity Provider: An Identity Provider is an entity that acts as an authentication service to end requestors and data origin authentication service to service providers (this is typically an extension of a Security Token Service). Identity Providers (IP) are trusted (logical) 3rd parties which need to be trusted both by the requestor (to maintain the requestor's identity information as the loss of this information can result in the compromise of the requestors identity) and the service provider which may grant access to valuable resources and information based upon the integrity of the identity information provided by the IP.

Indirect Brokered Trust: Indirect Brokered Trust is a variation on direct brokered trust where the second party negotiates with the third party, or additional parties, to assess the trust of the third party.

Information Security: Preservation of confidentiality, integrity and availability of information; in addition, other properties, such as authenticity, accountability, non-repudiation, and reliability can also be involved.

Information Security Event: An information security event is an identified occurrence of a system, service or network state indicating a possible breach of information security policy or failure of safeguards, or a previously unknown situation that may be security relevant.

Information Security Incident: An information security incident is indicated by a single or a series of unwanted or unexpected information security events that have a significant probability of compromising business operations and threatening information security.

Login: The process whereby a user presents credentials to an authentication authority, establishes a simple session, and optionally establishes a rich session.

Logout: The process whereby a user signifies desire to terminate a simple session or rich session.

MAC: A cryptographic value resulting from passing a message through the message authentication algorithm using a specific key.

Message Confidentiality: Message Confidentiality is a property of the message and encryption is the mechanism by which this property of the message is provided.

Message Freshness: Message freshness is the process of verifying that the message has not been replayed and is currently valid.

Message Integrity: Message Integrity is a property of the message and digital signature is a mechanism by which this property of the message is provided.
62. **Named Attribute**: A specific instance of an attribute, determined by the attribute name and type, the identity of the attribute holder (which may be of type: subject, resource, action or environment) and (optionally) the identity of the issuing authority.

63. **Obligation (Authorization Decision)**: An operation specified in a policy or policy set that should be performed by the PEP (Policy Enforcement Point) in conjunction with the enforcement of an authorization decision.

64. **Partial Identity**: An identity of an individual person may comprise many partial identities of which each represents the person in a specific context or role. A partial identity is a subset of attributes of a complete identity, where a complete identity is the union of all attributes of all identities of this person. On a technical level, these attributes are data. A pseudonym might be an identifier for a partial identity.

65. **Password**: A secret that a claimant memorizes and uses to authenticate his or her identity.

66. **Password Identification Number**: A password consisting only of decimal digits.

67. **Policy Administration Point**: The system entity that creates a policy or policy set Policy-combining algorithm. The procedure for combining the decision and obligations from multiple policies.

68. **Policy Combining Algorithm**: The procedure for combining the decision and obligations from multiple policies.

69. **Policy Decision Point**: The system entity that evaluates applicable policy and renders an authorization decision. This term is defined in a joint effort by the IETF Policy Framework Working Group and the Distributed Management Task Force (DMTF)/Common Information Model (CIM) in [RFC3198].

70. **Policy Enforcement Point**: The system entity that performs access control, by making decision requests and enforcing authorization decisions.

71. **Policy Set**: A set of policies, other policy sets, a policy-combining algorithm and (optionally) a set of obligations. May be a component of another policy set.

72. **Predicate**: A statement about attributes whose truth can be evaluated.

73. **Privacy Enhancing Identity Management**: Given the restrictions of a set of applications, identity management is called privacy-enhancing if it sufficiently preserves unlinkability (as seen by an attacker) between the partial identities of an individual person required by the applications. Identity management is called perfectly privacy-enhancing if it perfectly preserves unlinkability between the partial identities, i.e., by choosing the pseudonyms (and their authorizations) denoting the partial identities carefully, it maintains unlinkability between these partial identities.
towards an attacker to the same degree as giving the attacker the attributes with all pseudonyms omitted.

74. **Private Key:** A key of an entity’s key pair known only to that entity.

75. **Profile:** A set of rules describing how to embed assertions into and extract them from a framework or protocol. Each profile is given a name in the pattern “xxx profile of SAML”.

76. **Proof-of-possession token:** A proof-of-possession token is a security token that contains data that a sending party can use to demonstrate proof-of-possession. Typically, although not exclusively, the proof-of-possession information is encrypted with a key known only to the sender and recipient.

77. **Proxy:** A proxy can be defined like an entity authorized to act for another, an authority or power to act for another and also a document giving such authority.

78. **Public Key:** A key of an entity’s key pair known publicly.

79. **Public Key Certificate:** A digital document issued and digitally signed by the private key of a Certification Authority that binds the name of a subscriber to a public key. The certificate indicates that the subscriber identified in the certificate has sole control and access to the private key.

80. **Public Key Infrastructure:** Public Key Infrastructure (PKI) is the combination of software, encryption technologies, and services that enables entities to protect the security of their communications and business transactions on networks. Using a combination of private (i.e., secret) key and public key cryptography, PKI enables a number of other security services including data confidentiality, data integrity, and non-repudiation. PKI integrates digital certificates, public key cryptography, and certification authorities into a complete network security architecture. A typical PKI infrastructure encompasses the issuance of digital certificates to individual users and servers; end-user enrollment software; integration with certificate directories; tools for managing, renewing, and revoking certificates; and related services and support.

81. **Pseudonymity:** A pseudonym is an identifier of a subject other than one of the subject’s real names. A subject is pseudonymous if a pseudonym is used as identifier instead of one of its real names.

82. **Registration:** The process through which an end user applies to become a subscriber of a Cryptographic Service Provider (CSP) and an Registration Authority (RA) validates the identity of that user on behalf of the CSP.

83. **Registration Authority:** A trusted entity that establishes and vouches for the identity of a subscriber to a Cryptographic Provider (CSP). The Registration Authority (RA) may be an integral part of a CSP, or it may be independent of a CSP, but it has a relationship to the CSP(s).
84. **Request Security Token**: A Request Security Token is a message sent to a security token service to request a security token.

85. **Request Security Token Response**: A request security token response (RSTR) is a response to a request for a security token. In many cases this is a direct response from a security token service to a requestor after receiving an request security token message. However, in multi-exchange scenarios the requestor and security token service may exchange multiple RSTR messages before the security token service issues a final RSTR message. One or more RSTRs are contained within a single RequestSecurityTokenResponseCollection (RSTRC).

86. **Resource**: Data, service or system component.

87. **Risk**: Combination of the probability of an event and its consequence.

88. **Risk Management**: Coordinated activities to direct and control an organization with regard to risk.

89. **Rule**: A target, an effect and a condition. A component of a policy.

90. **Rule Combining Algorithm**: The procedure for combining decisions from multiple rules.

91. **SAMLAssertion Authority**: An abstract system entity that issues assertions.

92. **Secure Multi-Purpose Internet Mail Extensions**: A standard that extends the MIME to support the signing and encryption of e-mail transmitted across the Internet.

93. **Secure Sockets Layer**: Protocol for transmitting private documents via the Internet by using a private key to encrypt data transferred over the SSL connection.

94. **Security Architecture**: A plan and set of principles for an administrative domain and its security domains that describe the security services that a system is required to provide to meet the needs of its users, the system elements required to implement the services, and the performance levels required in the elements to deal with the threat environment. Complete security architecture for a system addresses administrative security, communication security, computer security, emanations security, personnel security, and physical security, and prescribes security policies for each. Complete security architecture needs to deal with both intentional, intelligent threats and accidental threats. Security architecture should explicitly evolve over time as an integral part of its administrative domain’s evolution.

95. **Security Assertion Markup Language**: An XML-based framework for communicating user authentication, entitlement and attribute information. SAML allows entities to make assertions regarding the identity, attributes, and entitlements of a subject (an entity that is often a human user) to other entities, such as a partner company or an application. SAML addresses web single sign-on, web services authentication,
attribute exchange, authorization, non repudiation, and secure communications. SAML defines assertion message formats that are referenced in Liberty Alliance, Shibboleth, WS-Security, and other specifications. SAML has become the standard web SSO identity management solution.

96. **Security Policy**: A set of rules and practices that specify or regulate how a system or organization provides security services to protect resources. Security policies are components of security architectures. Significant portions of security policies are implemented via security services, using security policy expressions.

97. **Security Service**: A processing or communication service that is provided by a system to give a specific kind of protection to resources, where said resources may reside with said system or reside with other systems, for example, an authentication service or a PKI-based document attribution and authentication service. Security services typically implement portions of security policies and are implemented via security mechanisms.

98. **Security Token**: A security token represents a collection (one or more) of claims.

99. **Security Token Service**: A security token service (STS) is a Web service that issues security tokens. That is, it makes assertions based on evidence that it trusts, to whoever trusts it (or to specific recipients). To communicate trust, a service requires proof, such as a signature to prove knowledge of a security token or set of security tokens. A service itself can generate tokens or it can rely on a separate STS to issue a security token with its own trust statement (note that for some security token formats this can just be a re-issuance or co-signature). This forms the basis of trust brokering.

100. **Sender Authentication**: Sender authentication is corroborated authentication evidence possibly across Web service actors/roles indicating the sender of a Web service message (and its associated data). Note that it is possible that a message may have multiple senders if authenticated intermediaries exist. Also note that it is application-dependent (and out of scope) as to how it is determined who first created the messages as the message originator might be independent of, or hidden behind an authenticated sender.

101. **Shared Secret**: A secret used in authentication that is known to the claimant (user) and the verifier. There are two durations for a shared secret: 1) Session (temporary) secret – duration of the secret is limited to the duration of the user session. That is, the secret is created, used, and expired during a single user authentication session. 2) Long-term secret – duration of the secret persists ongoing, and is used from one user authentication session to another user authentication session.

102. **Shibboleth**: Standards-based, open source middleware software which provides Web Single Sign-on (SSO) across or within
organizational boundaries. The Shibboleth software implements the OASIS SAML v1.1 specification, providing a federated Single Sign-on and attribute exchange framework. Shibboleth also provides extended privacy functionality allowing the browser user and their home site to control the Attribute information being released to each Service Provider.

103. **Signature:** A signature is a value computed with a cryptographic algorithm and bound to data in such a way that intended recipients of the data can use the signature to verify that the data has not been altered and/or has originated from the signer of the message, providing message integrity and authentication. The signature can be computed and verified with symmetric key algorithms, where the same key is used for signing and verifying, or with asymmetric key algorithms, where different keys are used for signing and verifying (a private and public key pair are used).

104. **Signature Validation:** Digital signature validation is the process of verifying that digitally signed data/message has not been altered since it was signed.

105. **Signed Security Token:** A signed security token is a security token that is asserted and cryptographically signed by a specific authority (e.g. an X.509 certificate or a Kerberos ticket).

106. **Simple Object Access Protocol:** Lightweight XML-based messaging protocol used to encode the information in Web service request and response messages before sending them over a network. It consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP messages are independent of any operating system or protocol and may be transported using a variety of Internet protocols, including MIME and HTTP.

107. **Single Sign-On:** Once an end user has authenticated their identity at a CS, he or she may, by their choice, move among RPs compatible with the CS without re-authenticating. In other words, the end user is seamlessly logged into any other RP compatible with the CS. For privacy considerations, end users must take explicit actions to opt-in to SSO. SSO applies to assertion based Federation member systems only. In addition, SSO is in effect only for the duration of the end user's current browser session and authentication session. An end user must opt-in to SSO each time he or she opens a new web browser session.

108. **Subject:** The person whose identity is bound in a particular credential.

109. **Target:** The set of decision requests, identified by definitions for resource, subject and action, that a rule, policy or policy set is intended to evaluate.
110. **Threat:** A potential cause of an unwanted *incident*, which may result in harm to a system or organization.

111. **Token:** Something that the *claimant* possesses or knows (typically a key or *password*) that can be used to remotely *authenticate* the claimant’s *identity*. Technically, the token includes an end user id and password that ensures token uniqueness within a credential domain.

112. **Transaction Identifier:** Mechanism for tracking transactions across various *components* in the architecture. TIDs will be generated by the E-Authentication Portal for SAML 1.0 and will be passed with the end user information.

113. **Transport Layer Security:** An *authentication* and security protocol implemented in current browsers and web servers.

114. **Trust:** Trust is the characteristic whereby one entity is willing to rely upon a second entity to execute a set of actions and/or to make a set of *assertions* about a set of *principals* and/or *digital identities*. In the general sense, trust derives from some relationship (typically a business or organizational relationship) between the entities. With respect to the assertions made by one entity to another, trust is commonly asserted by binding messages containing those assertions to a specific entity through the use of *digital signatures* and/or *encryption*.

115. **Trust Engine:** The trust engine of a *Web service* is a conceptual *component* that evaluates the security-related aspects of a message.

116. **Unlinkability:** Unlinkability of two or more items of interest (IOIs, e.g., *subjects*, messages, actions, ...) from an attacker’s perspective means that within the system (comprising these and possibly other items), the attacker cannot sufficiently distinguish whether these IOIs are related or not.

117. **Undetectability:** Undetectability of an item of interest (IOI) from an attacker’s perspective means that the attacker cannot sufficiently distinguish whether it exists or not.

118. **Unobservability:** Unobservability of an item of interest (IOI) means undetectability of the IOI against all subjects uninvolved in it and *anonymity* of the *subject(s)* involved in the IOI even against the other subject(s) involved in that IOI.

119. **User-Controlled Identity Management System:** A user-controlled identity management system is an *IMS* that makes the flow of this user’s *identity attributes* explicit to the user and gives its user a large degree of control. The guiding *principle* is “notice and choice”.

120. **Validation Service:** A service that validates certificates remotely. The *Validation Service* is an end-to-end solution that spans server-side (i.e., the validation service provider’s hosted service) and client-side (i.e., software integrated into the Agency application).
121. Virtual Identity: Virtual identity is sometimes used in the same meaning as digital identity or digital partial identity, but because of the connotation with “unreal, non-existent, seeming” the term is mainly applied to avatars.
PUBLICATIONS


