NEXOF-RA

NESSI Open Framework – Reference Architecture

IST- FP7-216446

Deliverable D14.1

Methodology to Write Instantiation Guidelines for the NEXOF Reference Architecture

Due date of deliverable: 30/06/2010

Actual submission date: 13/07/2010

Francisco Pérez-Sorrosal
Ricardo Jiménez-Péris
Marta Patiño-Martínez

Date: 14/07/2010

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.

creative commons

Change History

Version	Date	Status	Author (Partner)	Description
0.1	17/06/2010		Francisco Pérez Sorrosal	First Version
0.2	28/06/2010		Francisco Pérez Sorrosal, Ricardo Jiménez Peris, Marta Patiño Martínez	Evelyn and Angelo's comments addressed Integration of the Guidelines for HA and Scalability in E-SOA as example
1.0	13/07/2010		Ricardo Jiménez Peris, Marta Patiño Martínez	Final version

cc creative commons

EXECUTIVE SUMMARY

This deliverable describes a methodology to write instantiation guidelines for architectural domains of the NEXOF Reference Architecture (NEXOF-RA). The methodology can be used for both documenting the instantiation process of current domains addressed by the reference architecture as well as for new domains that might be added in the future to the reference architecture. The deliverable also provides a full example of instantiation guidelines developed for the High Availability and Scalability domain of the ESOA top-level pattern.

creative commons

Document Information

IST Project Number	FP7 – 216446	**Acronym**	NEXOF-RA
Full title	NESSI Open Framework – Reference Architecture		
Project URL	http://www.nexof-ra.eu		
EU Project officer	Arian Zwegers		

Deliverable	**Number**	D14.1	**Title**	Methodology to Write Instantiation Guidelines for the NEXOF Reference Architecture
Work package	**Number**	WP-4	**Title**	Methodology to Write Instantiation Guidelines for the NEXOF Reference Architecture

Date of delivery	**Contractual**	**30/06/2010**	**Actual**	**30/06/2010**
Status	Version 1.0		Final ☑	
Nature	Report ☑ Demonstrator ☐ Other ☐			
Abstract (for dissemination)				
Keywords	Instantiation guidelines			

Internal reviewers	Angelo Gaeta (MOMA)		
	Evelyn Pfeuffer (Siemens)		
Authors (Partner)	Francisco Pérez-Sorrosal (UPM), Ricardo Jiménez-Peris (UPM), Marta Patiño-Martínez (UPM)		
Responsible Author	Ricardo Jiménez-Peris	**Email**	rjimenez@fi.upm.es
	Partner UPM	**Phone**	+34 656 68 29 48

creative commons

TABLE OF CONTENTS

EXECUTIVE SUMMARY 3
TABLE OF CONTENTS 5
1 INTRODUCTION 6
1.1 This Methodology is Not… 7
2 METHODOLOGY 8
2.1 Description of the Intention 9
2.2 Identification of Functional Aspects/Requirements Addressed 9
2.3 Description of the Functional Patterns 10
2.4 Provision of an Instantiation Process for Obtaining a Functional Architecture/s 11
2.5 Identification of the Non-Functional Aspects/Requirements Addressed 12
2.6 Description of the Non-Functional Patterns Addressed 12
2.7 Provision of a Process for Selecting the Non-Functional Crosscutting patterns 14
2.7.1 Techniques to Evaluate Non-Functional Quantitative Attributes 16
3 CONCLUSION 18
APPENDIX A: INSTANTIATION GUIDELINES FOR HIGH AVAILABILITY AND SCALABILITY IN E-SOA 19
Context and Intent 19
Pattern Categorization 20
Generic Replication Patterns Domain 23
Multi-Tier Replication Patterns Domain 24
Database Replication Patterns Domain 25
Helper/Low Level Replication Patterns Domain 26
Instantiation Process 28
Phase 1: Confront Pattern Assumptions with Initial Architecture 30
Phase 2: Pattern Selection Through Trade-Off Analysis 34
Phase 3: Evaluation of Quantitative Requirements Fulfilment 42
The Role of ATAM in the Instantiation Process for HA and Scalability Properties 45
ACRONYMS 47
REFERENCES 48

creative commons

1 INTRODUCTION

Defining the architecture of a system or defining architectural extensions to existing systems are the core tasks performed by software architects. When performing these tasks, the architects have to take some important decisions that will have some long lasting effects on the resulting systems. The decisions taken (or not taken) by the architects in this phase will be very difficult to catch up later in the subsequent phases of the software development process.

The decisions to take have to do mainly with two different elements: functional and non-functional requirements. Both kinds of requirements address as a whole, the needs and demands to be fulfilled in each specific context (e.g. E-SOA, IoS, etc.).

Functional requirements are related specifically to the required system functionality that must be provided at the end of the design process of any system architecture or architectural extension. In order to fulfil these requirements, the architect has to combine different architectural blocks (i.e. functional patterns) in order to provide as outcome, one or more architectural configurations addressing the required functionality. This process has to be done taking into account and respecting the relationships that can be set up between those patterns. This/these architecture/s are termed *functional architecture/s*.

However, how architects achieve security, scalability, maintainability, high availability, etc. in the system architecture is one of the most difficult key points to materialize for guaranteeing the success of a project. So, with regard non-functional requirements (a.k.a. quality attributes), the architect has to be very focused in how to accommodate the cross-cutting patterns that allow to fulfil them in the functional system architecture/s.

The process of instantiating architectures is complex, especially in the case of reference architectures such as the NEXOF Reference Architecture (NEXOF-RA). The evolving nature of this kind of architectures requires adding extensions to the core reference architecture, for example when a new application context/domain appears. This characteristic is called “Extendability” in NEXOF-RA (See Section 2.2.2 in D6.3 [NRM]).

The methodology presented in this document, establishes the steps that should be performed in order to write instantiation guidelines for different domains of the reference architecture either existing ones or future extensions of the NEXOF-RA. The methodology tackles on how to instantiate a given architecture taking into account both, functional and non-functional requirements. In this way, the instantiation guidelines resulting from applying this methodology can be document how NEXOF-RA can be used by system architects as an entry point to understand particular contexts and architectural extensions.

creative commons

1.1 This Methodology is Not…

It is important to note that the goal of this methodology is NOT to describe the NEXOF Reference Architecture NOR describe the instantiation of any particular NEXOF-RA compliant platforms/infrastructures (NCIs/NCPs). Additional information about these two topics can be found in the document that describes the architectural framework and principles D7.2c [AFP] and in the deliverable D6.3 that explains the reference model [NRM].

creative commons

2 METHODOLOGY

When a new application context arises (e.g. E-SOA, Internet of Services), the current elements and mechanisms (standards, abstract and concrete components, pattern catalogue etc.) provided by the NEXOF-RA, may be not sufficient to address the new requirements (both functional and non-functional) introduced by the new context. When this occurs, it is necessary to extend the NEXOF-RA. In order to not to turn this process into chaos, a set of guidelines for extending the NEXOF-RA architecture must be provided.

In order to write the guidelines that describe a new context or extension to NEXOF-RA in terms of architecture (called *architectural extension*) and how can NEXOF-compliant solutions be instantiated from them, the following key points are advised to be addressed:

1. **Describe the intention of the application context or architectural extension provided (Section 2.1)**.
2. **Identify the functional aspects addressed by the new context/extension (Section 2.2)**.
3. **Introduce the functional patterns that allow fulfilling the functional aspects addressed by the new context/extension (Section 2.3)**.
4. **Provide an instantiation process to help the architect in selecting and combining the functional patterns provided in order to derive one or more functional architectures (Section 2.4).**
5. **Identify the non-functional aspects addressed (Section 2.5)**.
6. **Introduce the non-functional crosscutting patterns that allow fulfilling the non-functional aspects (Section 2.6).**
7. **Provide an instantiation process to help/guide the architect in selecting the non-functional cross-cutting patterns for the new context taking into account their applicability on the functional architecture/s obtained in point 4 (Section 2.7).**

It is recommended to provide the guidelines for the new context or architectural extension in a separate document addressing the required points from the seven described above. The first one is mandatory, and its goal is to act as an entry/link point to the reader in order to understand the intention of the context or architectural extension that is going to be provided. The rest of the points can be divided in two *parts*; points from 2 to 4 are related to the functionality provided by the context/extension, whilst points from 5 to 7 are related to non-functional aspects.

The guidelines for a particular context or architectural extension may not need to address all the points provided above. For example, if a new context or extension is related exclusively to non-functional aspects (e.g. security), the guidelines will include only points 1, 5, 6 and 7. Moreover, the order of the two parts is not mandatory either. If an architect considers that the non-functional aspects guide better the description of a particular context, is free to re-organize the functional and non-functional parts.

creative commons

The next sections include the description of the contents that each one of the previous points should address.

2.1 Description of the Intention

The goal and intent of the application context or architectural extension that is going to be introduced must be described and motivated here. As the new context or extension is going to provide new additional elements for the NEXOF-RA (e.g. new patterns, standards etc.), these additions needs to be justified in terms of business and/or technical requirements.

An example of the kind of content/description that should be provided in this section can be found in the description of the context for Enterprise Service Oriented Architectures described in the E-SOA pattern [E-SO] (Section 2):

> "*The current market context is characterized by continuous growth, rapid changes and product and service innovations that require enterprises to respond rapidly to adapt their business processes. The success of an enterprise is, then, tied to its ability to suddenly embrace new business requirements. This ability is mostly related to IT. Agility and adaptability of IT systems are the most pressing issues of contemporary IT.*
>
> *…*
>
> *As a result, enterprises need to factor the system in reusable functionality and make it easier to compose them to meet business requirements. This requires to have self-contained functionality that are as much independent as possible from other functionality. When functionality grows, it becomes a fundamental issue to well design, organize and share them to help their effective reuse.*
>
> *For these types of "in-flux" operations, a loosely coupled architecture is required because it helps to reduce the overall complexity and dependencies. Such an architectural style makes the application landscape more agile, enables quicker change, and reduces risk. The concept of service-oriented architectures aims at providing exactly these types of features."*

Descriptions of scenarios such as the ones found in the deliverable D10.1 [RR] relative to requirements may also be useful in order to strengthen the introduction of the new application context or extension.

2.2 Identification of Functional Aspects/Requirements Addressed

The new context extension may introduce additional functionalities (a.k.a. functional requirements/aspects) to the current ones identified in the NEXOF-RA. In principle, the functionalities must be specified by referring to the concerns and functionalities captured by the *NEXOF-RA Model* [AFP, NRM], i.e. Services, Messaging, Discovery, Composition, Analysis, Presentation, Management, Security, Resources.

creative commons

When describing the functionalities addressed, it is possible to reference the identified business requirements (See Section 2.1). This fact reinforces the need for the new context or extension that is being introduced.

If the functionalities provided in the new application context are radically different and do not fit in any of the current concerns provided by the NEXOF-RA model, the architect might decide to introduce a new concern in the model if she considers that is strictly necessary.

2.3 Description of the Functional Patterns

In this point, the functional patterns that are going to be included in the new context or extension are introduced and described.

On one side, this implies the description of the functional patterns in separate documents as it is described in D7.2c, "Definition of the Architectural Framework and Principles" [AFP]. Each one of these documents contextualizes a particular pattern, providing information about the problem that addresses, the assumptions it requires, relationships with the functional requirements identified in the Section 2.2, related standards, the architectural solution it provides, etc.

On the other hand, it is interesting to provide a high level view of how all the patterns provided fit together. This can be done through one or more diagrams that include the patterns (joint with the relationships between them), the functionalities they are addressing and how the patterns and the relationships are related. The following diagram (Figure 1) is an example of this kind of diagrams for the Service concern in the Enterprise SOA context:

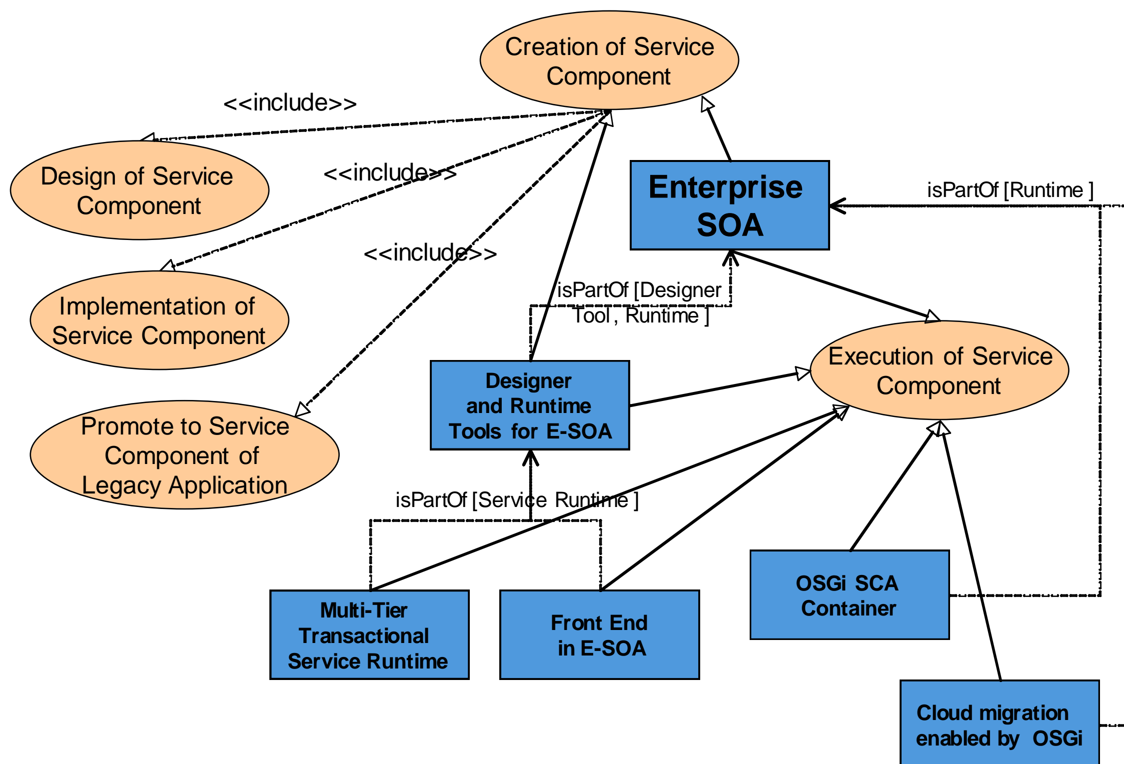


Figure 1 Patterns, functionalities and their relationships in the E-SOA context

creative commons

In this diagram, the patterns related to the Services concern are depicted as blue boxes and the relationships between them as dotted arrows categorized with the relationships for functional patterns described in D7.2c. The functionalities addressed are depicted as ovals. Finally, the black arrows relate the patterns with the functionalities.

At least one top-level pattern should be introduced in order to provide an entry point to the new context or extension of the architecture. In the previous figure, the top-level pattern *Enterprise SOA* provides the entry point. The rest of the functional patterns (abstract and implementation) can be related to the top-level pattern/s (and among themselves), following the relationships in D7.2c. In the previous figure, the *Designer and Runtime Tools for E-SOA*, *OSGi SCA Container* and *Cloud Migration Enabled by OSGi* patterns are related to the Enterprise SOA pattern through the *isPartOf[component]* relationship. As it is described in D7.2c, this implies that the solution provided by these patterns has to satisfy all the requirements that the set of components specified between brackets must meet in the *Enterprise SOA* pattern. The same applies for the *Multi-Tier Transactional Service Runtime* and the *Front End in E-SOA* patterns with regard to the *Designer and Runtime Tools for E-SOA* pattern.

2.4 Provision of an Instantiation Process for Obtaining a Functional Architecture/s

Once the functional patterns for the application context have been categorized, a process for deriving architectures that fulfil the functional requirements of the new context is necessary. The goal of the process is to help architects in instantiating functional architectures for the application context using the elements provided by the existing current NEXOF-RA and the new patterns provided in the previous point.

The exact steps provided by the instantiation process depend on the concrete context that is being described. However, some steps can be highlighted for all the contexts, as is described in the following paragraphs.

First of all, it is useful to find the driving principle that will guide the process. The process can be driven by the requirements specified for the final system or the quality attributes to be instantiated or by any other criteria decided by the architect that became of interest/utility for understanding the instantiation process.

The process has to take also into account that some of the functional patterns introduced may refine some already existing components or patterns of the current NEXOF-RA or can be related with each other in order to form new building blocks that jointly address in a better way some of the functionalities (collection of patterns). In order to better understand the possible architectural choices, their description can be done with diagrams based on the pattern categorization done in the previous point and other categorizations already existing in the NEXOF-RA that can be used by this context or extension being described.

creative commons

Moreover, the existing trade-offs of the architectural choices must be described and analyzed, exhibiting the pros and cons of each one of them.

Finally, a method for the quantitative evaluation of the resulting architecture/s should be provided or suggested. For example, in NEXOF-RA assessment and validation activities for architectural decisions have been based on a common foundation that is the Architectural Trade-off Analysis Method (ATAM) [KKC00].

The outcome after this point is a functional architecture or architectures that address/es the functional requirements of the new context. The next points will take into consideration the impact of non-functional aspects on this functional architecture/s.

2.5 Identification of the Non-Functional Aspects/Requirements Addressed

After taking into account the functional properties of the application context or architectural extension, then the non-functional aspects, if they are required to be addressed by the extension, must be presented. As it has been done with the functional aspects in Section 2.2, here there are described the non-functional aspects/attributes that the architectural extension promotes.

When possible, it is desirable to refer to the business requirements or scenarios identified in Section 2.1 that the non-functional aspects are going to address. In this way, it is reinforced the need for the new context or extension that is being introduced.

For example, Section "Context and Intent" of Appendix A describes the instantiation guidelines for two particular non-functional aspects required by current E-SOA infrastructures, High Availability and Scalability.

If the non-functional aspects addressed in the new application context or extension are radically different and do not fit in any of the current concerns provided by the NEXOF-RA model, the architect might decide to introduce a new concern in the model if she considers that is strictly necessary.

2.6 Description of the Non-Functional Patterns Addressed

At this point the non-functional patterns are introduced and described. This pattern description must be done reflecting the existing mechanisms provided by the NEXOF-RA (See D7.2c [AFP]) with regard crosscutting patterns.

The way of describing the non-functional crosscutting patterns depends on the concrete context that is being described. However, some key points are going to be described and illustrated in the following paragraphs.

First of all, it should be identified at least one functional pattern where the non-functional crosscutting patterns are applicable to. This pattern will provide the hook to where the non-functional crosscutting patterns in this context can be applied.

Then, it is also interesting to introduce a pattern categorization with regard the non-functional aspects identified and the problems that they aim to solve. This

creative commons

categorization will help the architects in understanding the set of crosscutting patterns as a whole.

The following figure (Figure 2) presents a fraction of the diagram that categorizes the non-functional cross-cutting patterns related to high availability and scalability in the context of Enterprise SOA (See Section "Pattern Categorization" in Appendix A. More specifically, it depicts the problems and patterns in the Multi-Tier Replication Domain. Instead of presenting directly all the patterns related to high availability and scalability, the document has identified different sub-domains in order to better categorize all the patterns and present them to the user in a clear way. Other domains in which the patterns have been categorized in this context are the Generic Replication Domain, the Database Replication Domain and the Helper Patterns Domain.

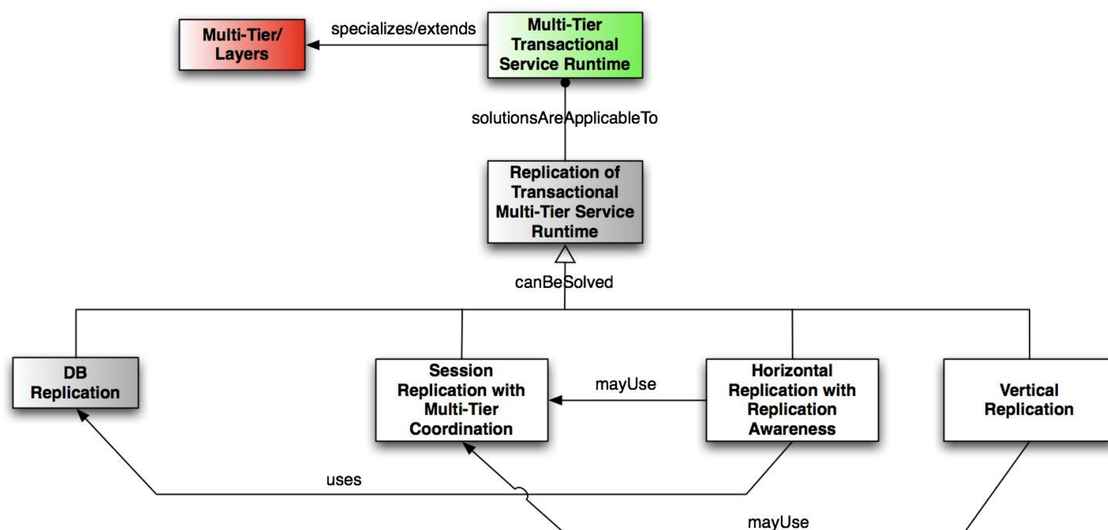


Figure 2 HA and Scalability patterns in E-SOA

In this case, the *Multi-Tier Transactional Service Runtime* pattern (depicted in green) is the functional pattern –offering a concrete architectural approach- where the non-functional crosscutting patterns in this domain (in white) can be applied. From this point on, the diagram can be interpreted with the help of the instructions found in D7.2c as follows[1].

The *Multi-Tier Transactional Service Runtime* pattern *specializes/extends* an architectural approach described as a pattern the literature [GHJV95, BMRS+96], in this case the *Multi-Tier/Layers* pattern (in red). The gray-box under the green box represents the main *problem* that the non-functional crosscutting patterns presented aim to solve and can be seen as a domain for categorizing different solutions; in this case, the replication of transactional multi-tier runtimes. The *solutionsAreApplicableTo* relationship –expressed through the line ended with a black dot- that links the problem with the functional pattern, describes that the solutions to this problem are applicable to the *Multi-Tier Transactional Service Runtime* pattern in order to extend it with

[1] The concrete semantics of the relationships and boxes depicted in the figure are the tools defined in D7.2 to describe non-functional crosscutting pattern descriptions.

creative commons

the additional non-functional features provided by the crosscutting patterns. The complete explanation of the pattern categorization in the sub-domain depicted in this diagram can be found in [IGHAS].

2.7 Provision of a Process for Selecting the Non-Functional Crosscutting patterns

After point 4 described in Section 2.4, the architects are able to build architectures that address the functional aspects of the new application context, the so-called functional architectures. At this point, the crosscutting patterns introduced in Section 2.6 can be applied in order to enhance the architectures with non-functional aspects.

As it happens with functional aspects, the way of describing the process for instantiating the non-functional crosscutting patterns depend on the concrete context that is being described. However, some steps can be highlighted for all the contexts, as is described in the following paragraphs.

First of all, some of the initial functional architectures that are the outcome of Section 2.4 can be discarded taking into account some of the non-functional requirements of the desired system. For example, if one of the non-functional requirements for the system is maintainability and we have two functional architectures as result of Section 2.4, being one of them monolithic and the other based on a microkernel approach, the architect can automatically discard the monolithic solution. If this step can be introduced in the process, many of the resulting functional architectures can be filtered.

Moreover, the assumptions of the non-functional crosscutting patterns must be taken into account when they are going to be applied to the functional architecture/s, in order to properly accommodate them. In this case, this step filters those non-functional patterns that cannot be applied.

As occurs with functional patterns, it must be considered that non-functional cross-cutting patterns may refine some already existing components or patterns of the current NEXOF-RA or can be related to each other to address collectively a set of non-functional requirements. Therefore, the existing trade-offs of the different alternatives that have been identified must be described and analyzed, exhibiting the pros and cons that may arise on the resulting functional architecture/s enhanced with the non-functional patterns.

It is also important to take into account the priority of the non-functional requirements to fulfil when selecting the non-functional patterns or building blocks, because the selection of a pattern for achieving a particular non-functional requirement may affect further selection of other patterns.

Finally, as multiple alternative architectures can be obtained at the end of this process, it will be necessary to apply one or more evaluation methods for evaluating the most critical/relevant quantitative attributes of the resulting architectures. This will enable the architect to take an informed decision regarding the architectural choices and justify the final architecture chosen. There are three main options/techniques to evaluate the quantitative requirements of a system:

creative commons

1. Analytically
2. Through simulations
3. Implementing prototypes and/or Proofs of Concept (PoC)

Each one of them is described in the next subsection.

As example, in Appendix A is described the process designed for the instantiation guidelines for high availability and scalability in the context of Enterprise SOA (See Section "Instantiation Process"). The next figure (Figure 3) offers an illustrative schema of the instantiation process.

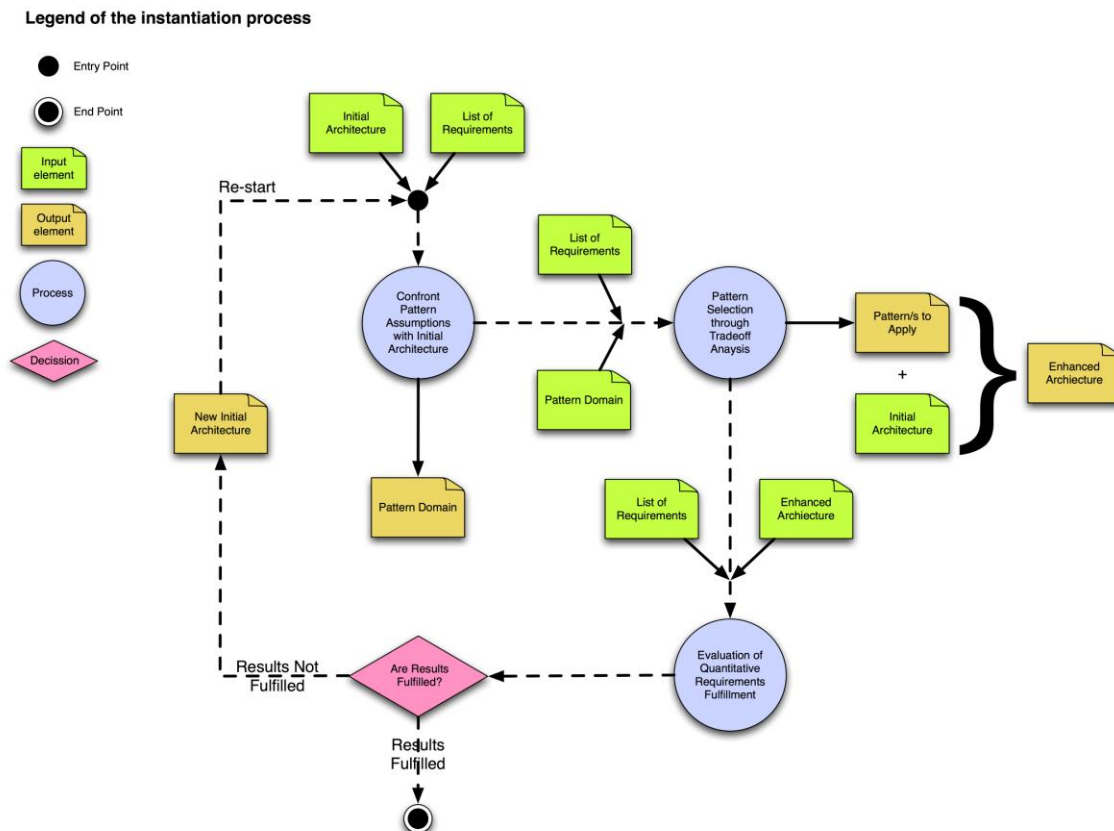


Figure 3 Instantiation process of the HA and Scalability Instantiation Guidelines

The phases described aim at guiding architects in the process of selecting the most appropriate HA and scalability patterns for the desired Enterprise SOA system obtaining in the end, an enhanced functional architecture. If the high level architecture is too complex, is recommended first to split it in several subsystems in order to derive the right high available and/or scalable architecture for each one of them. The phases of the instantiation process can be sum up as follows:

1. Confront the assumptions made by the different patterns in terms of architecture (e.g. a multi-tier architecture or the use of a database component) against the initial functional architecture to be enriched with high availability and/or scalability. This will help the architect in selecting the most appropriate pattern domain/s described in the document.

cc creative commons

2. Given a pattern domain and the prioritization of the non-functional requirements for the architecture to be instantiated in this phase, a trade-off analysis is performed to select the most appropriate pattern from the domain.
3. For some applications, some of the non-functional requirements are expressed quantitatively (e.g. response time below 10 ms). After improving the architecture to deal with qualitative non-functional requirements, it is needed to evaluate whether the architecture will fulfil as well the quantitative non-functional requirements.

At the end of phase 3, if all the input requirements can be fulfilled, the process is completed. Otherwise, it will be necessary for the architect to re-think and re-structure the initial architecture passed as input and return to phase 1.

2.7.1 Techniques to Evaluate Non-Functional Quantitative Attributes

These are the three main techniques used in the evaluation of architectures with regard to quantitative attributes.

2.7.1.1 *Analytical Evaluation*

The goal of this step is to compare analytically alternative system architectures in terms of the non-functional attributes. In order to do so, it is required the construction of analytical (i.e. mathematical) models that provide an analytical quantification of the different evaluated non-functional attributes for each system architecture being evaluated. An analytical model has a number of parameters and then yields a quantitative value of the non-functional attributes it evaluates such as scale-out, etc.

For instance, in [JPAK03] provides an analytical model for estimating the scale-out for database replication protocols that takes three parameters, number of replicas, workload (fraction of read-only queries) and ratio between the cost of fully executing and update transaction and simply installing the resulting updates, and yields the scale-out of the replicated database.

2.7.1.2 *Evaluation Through Simulations*

A simulation is another technique that can be used for evaluating quantitatively system architectures. It consists in simulating the environment partially or totally and evaluate the architecture in the simulated environment. A simulation represents key features of the architecture and environment and provides an approximation of the quantitative evaluation of some non-functional attributes.

This technique can be used when one or more of the following situations are presented to the architect:

- If is too expensive in terms of time and/or money to fully implement a prototype in order to evaluate it.

creative commons

- If the environment of the prototype is too complex, too expensive too lengthy (i.e. the simulation covers months or years of execution) or unfeasible to reproduce.
- If is too risky to test the prototype in the real environment (e.g. safety or economical reasons).

An example of simulation can be found in [BJPQ+05, BJPQ+08] where a simulation is used to evaluate different quorum systems in the context of P2P Networks based on distributed hash tables (DHTs). In this case is the difficulty that was overcome by means of the simulation was the lack of availability of a large scale environment (1000s of nodes) to evaluate the quorum protocols. The simulator was built in order to enable the evaluation of the protocols in large virtual P2P networks with 1000s of nodes. The simulator is in charge of emulating each quorum algorithm (i.e. the quorum messages, etc.), routing the messages among simulated nodes, simulating joins and leaves, failures and keeping track of the different metrics necessary to perform a performance comparison of each algorithm.

2.7.1.3 Prototypes and Proofs of Concept (PoC) Evaluation

The last method proposed to evaluate the architectures is by building prototypes. This process requires more work and resources than the others because it is necessary to implement and evaluate the key components of the architecture. Prototypes are a key concept for Proof of Concepts (PoCs) of NEXOF-RA. The goal of a PoC is “on the validation of patterns’ claim about quality attributes” and is defined as “a (set of) software artefact(s) used to validate some patterns of the NEXOF-RA.” [PPSC]. By means of a prototype/PoC it becomes possible to run one or more evaluation campaigns that measure quantitatively the value of different quality attributes under different configurations, enabling the comparison of the quality attributes across different architectural alternatives and also to measure the quality attributes for a single architecture to validate whether the architectural approach will be able to satisfy a particular set of non-functional requirements, for instance, attain a particular response time for a service or a particular scale-out in a distributed architecture.

creative commons

3 Conclusion

This document has presented a methodology that provides the steps to write instantiation guidelines for architectural domains in the NEXOF Reference Architecture (NEXOF-RA). The methodology is flexible enough to address different cases of instantiation. Appendix A provides a full developed example of instantiation guidelines that have been produced using the proposed methodology for two non-functional properties, High Availability and Scalability, for Enterprise Service-Oriented Architectures.

creative commons

APPENDIX A: INSTANTIATION GUIDELINES FOR HIGH AVAILABILITY AND SCALABILITY IN E-SOA

This appendix shows a complete example of Instantiation Guidelines document generated by the methodology described earlier in this document. The instantiation guidelines address two non-functional aspects -High Availability and Scalability- in the E-SOA context. These guidelines follow the steps 1, 5, 6 and 7 found in the methodology described in Section 2 of this document.

Context and Intent

Defining the architecture of a system is the main task performed by software architects. When performing this task, the architect has to take some important decisions that will have some long lasting effects on the resulting systems. The decisions taken (or not taken) by the architects in this phase will be very difficult to catch up later in the subsequent phases of the software development process. The most part of difficulties that arise when defining architectures are related to quality attributes rather than functional requirements. So, the architect has to be very focused in how to accommodate the expected non-functional requirements (a.k.a. quality attributes) in the final system architecture. It's how do architects achieve security, how do they achieve scalability, maintainability, high availability, etc. in the system what are the most difficult key points to achieve for guaranteeing the success of many projects. At those points is where there are raised a lot of difficult decisions to be taken.

High availability (HA) and scalability are two of the most important requirements to take into account when designing architectures for modern information systems such as Enterprise SOA-based systems or Internet of Services applications. High availability implies the ability to tolerate failures of individual parts of a system (or the whole system itself) and perform recovery while, at the same time, continue to provide service. Scalability means that the system is able to react to increasing loads by incrementally adding system resources without increasing the response time of individual requests.

Currently, there is a large set of applications in any professional -e.g. banking, customer relationship management (CRM), supply chain management, etc.- or entertainment area -e.g. social applications such as Twitter or Facebook, online games etc.- deployed in Service-Oriented or Cloud Architectures, requiring HA and scalability. High availability is required because these applications must be available 24/7 in order to provide the required services to their clients. On the other hand, scalability is needed because of the continuous growth and decay of the client base of these applications, what requires the underlying runtime to be scalable and elastic.

Both requirements, scalability and availability, can be addressed by replication. Replication is a well-known technique that consists in introducing redundancy in the critical parts of a system. In current architectures, it is commonly implemented by running a critical system (e.g. a database management system) on multiple nodes. When replication is implemented in this way, it is said that each node contains a ***replica*** of the system. In this way, it becomes

creative commons

possible to tolerate failures of individual replicas as the replicas in other nodes can take over, and allows for splitting the work triggered by client requests among the different replicas. Many replication solutions are either designed for availability or for scalability, but some can fulfil both purposes.

However, replication has the challenging task of replica control to maintain consistency. Moreover, most high-throughput information systems have strong consistency requirements that demand that replicas are always consistent (replicated data should have the same state at all replicas) in order to achieve replication transparency.

The guidelines described in this document help software architects in deciding the HA and scalability patterns that best fit with the requirements of the Enterprise SOA-based architectures that need to design, taking into account the different trade-offs that are present or may arise. In this way, once the most appropriate pattern/s has/have been selected, each pattern template document will guide the architect in applying its contents to the final architecture. Finally, the guidelines also offer an overview of the different existing methods to evaluate quantitatively the enhanced architectures with HA and scalability.

Pattern Categorization

The following set of patterns addresses high availability and scalability requirements in current system architectures. The patterns provided are defined at such a level of abstraction that allows them to be adapted to a very wide range of system configurations, which in the end allows fulfilling the non-functional requirements specified by the architects.

Figure 4 presents the set of patterns and problems related to high availability and scalability and how they are interrelated taking into account the relationships described in D7.2c [AFP].

creative commons

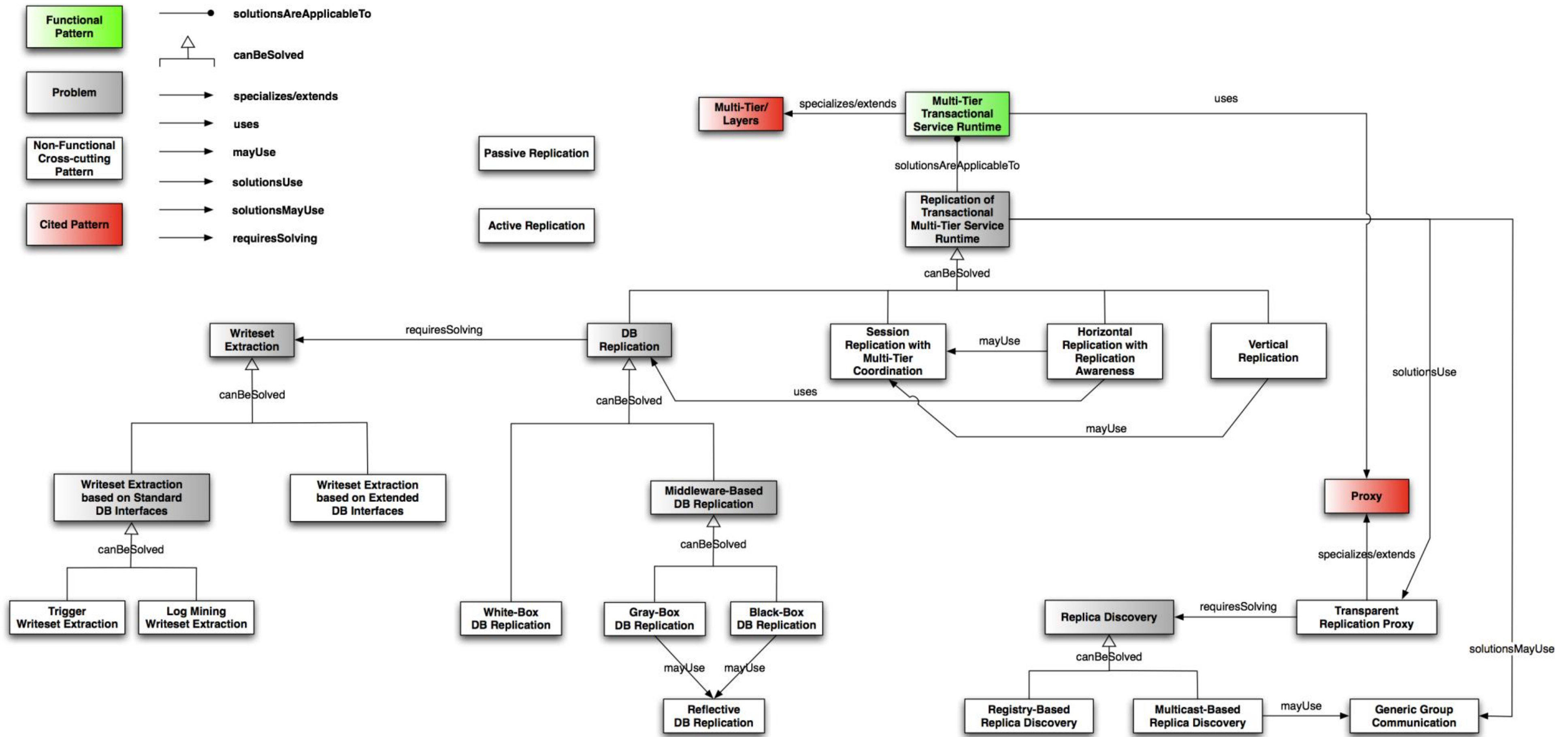


Figure 4 Set of patterns for high availability and scalability

creative commons

The *Multi-Tier Transactional Service Runtime* pattern (depicted in green) can be seen as the root of the diagram. It is a functional pattern described in the NEXOF-RA that represents the architectural choice where the HA and scalability non-functional crosscutting patterns described can be applied. These patterns are depicted as white boxes and form the core of these guidelines. In order to help in the pattern categorization, we have added to the figure several grey boxes that represent those problems that the patterns aim to solve (See D7.2c). Finally, the patterns depicted in red represent cited patterns from the literature that have been related to NEXOF-RA patterns, but not described within the RA [GHJV95, BMRS+96].

For the sake of clarity, we are going to classify the HA and scalability non-functional crosscutting patterns (white boxes) in four different domains. The following are the domains identified for the set of patterns shown above:

1. **Generic replication patterns**. This domain of patterns describes well-known generic replication techniques applicable to many components requiring mainly high availability.
2. **Patterns related to multi-tier replication**. In this domain are included those patterns related to replication of multi-tier architectures.
3. **Patterns related to database replication**. In this domain are included the different alternatives to implement database replication.
4. **Helper/low level patterns supporting replication**. Finally, these patterns are used by the other patterns to implement/complement certain features.

Each one of the first three domains contains alternative patterns that solve a specific problem (e.g. Database Replication). Depending on the pattern, it can sometimes be combined with other patterns in other domains (e.g. *Vertical Replication* pattern can be combined with *Passive Replication* [PVPJ06]). Finally, the fourth domain includes helper patterns that can be used by the other domains of patterns. The following subsections describe all the pattern sub-domains and their relationships, but first of all we are going to describe a general overview of the pattern domains in Figure 4 with the help of the relationships described in D7.2c.

The *Multi-Tier Transactional Service Runtime* pattern (green) *specializes/ extends* an architectural approach described as a pattern the literature, in this case the *Multi-Tier/Layers* pattern (in red). The grey box connected to this pattern represents the main *problem* that the non-functional crosscutting patterns presented aim to solve, i.e., the replication of transactional multi-tier runtimes. The *solutionsAreApplicableTo* relationship states that the solutions to this problem are applicable to the *Multi-Tier Transactional Service Runtime* pattern in order to extend it with the additional non-functional features provided by the crosscutting patterns.

The *Replication of Transactional Multi-Tier Service Runtime* problem is linked to the "**Helper/low level patterns supporting replication**" domain through the

creative commons

solutionsUse and *solutionsMayUse* relationships established to two pattern of this domain, i.e. the *Transparent Replication Proxy* and the *Generic Group Communication* patterns respectively. These relationships express that the pattern the arrow is pointing must be or may be used by any of the solutions specified to the problem respectively.

We have categorized the solutions to this first problem –expressed through the *canBeSolved* relationship- under the domain "**Patterns related to multi-tier replication**." In this domain there is a grey box expressing a sub-problem termed as *DBReplication*. This problem contextualizes the domain "**Patterns related to database replication**."

Finally, the two patterns that form the "**Generic replication patterns**" domain are not linked to any problem or pattern because they can be applied independently or in combination to other patterns in other domains to those components requiring replication.

Generic Replication Patterns Domain

The following figure presents the patterns in this domain.

The patterns in this domain are not linked to any problem or pattern because they can be applied independently or in combination to other patterns in other domains to those components requiring replication.

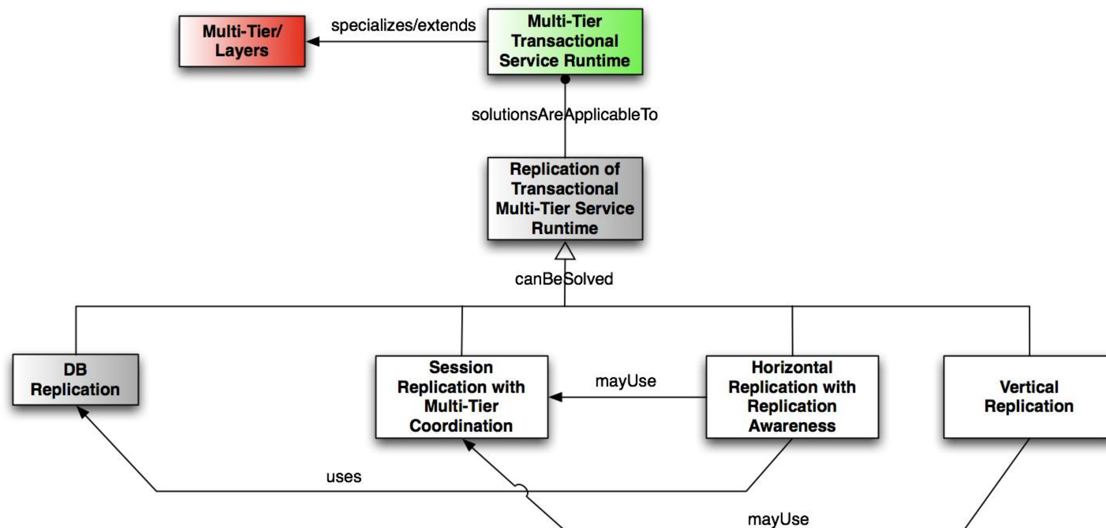
The patterns include:

- **Active Replication**: This pattern describes a technique based on redundancy used for masking errors and achieving high availability of critical components using a group or replicas. The pattern requires that all the (deterministic) requests be delivered to all the component replicas in order to be processed. In the end, taking into account the outputs received from the replicas, a consensus algorithm is used in order to decide on the output.
- **Passive Replication**: This pattern describes a technique based on redundancy used for masking errors and achieving high availability of critical components using a group or replicas. The pattern requires that one of the replicas, called primary, handles the input requests and the rest of them act as a backups in case that primary fails. That's because this pattern is also known as primary-backup.

creative commons

Multi-Tier Replication Patterns Domain

The following figure presents the patterns in this domain and how they are interrelated.



In the patterns above, the following two patterns represent the context where the rest of the patterns are applicable:

- **Multi-Tier Transactional Service Runtime**: This pattern describes the architecture of a multi-tier service runtime based on the Layers pattern. As it is shown in Figure 4, this pattern represents the starting point in which to apply the high availability and scalability patterns described in these guidelines.
- **Layers**: This is a well-know architectural pattern that “helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction” [BMRS+96].

The rest of the patterns show the different alternatives to perform replication in multi-tier architectures (*canBeSolved* relationships). Two different domains can be distinguished. The replication of a single tier is represented by the sub-domain composed by the *Session Replication with Multi-Tier Coordination pattern* –that represents the replication of the middle/business tier- and the *DB Replication Box*, which represents the different alternatives to solve the problem of performing the replication of the data tier (**DB Replication Domain**).

- **Session Replication with Multi-Tier Coordination**: The Session Replication pattern is commonly used to achieve availability and scalability in the application server tier. The Multi-Tier Coordination pattern is useful to track executions that cross tier boundaries in a multi-tier architecture.
- **DB Replication:** This box represents the DB Replication patterns presented in Section “Database Replication Patterns Domain” (See Figure 4).

creative commons

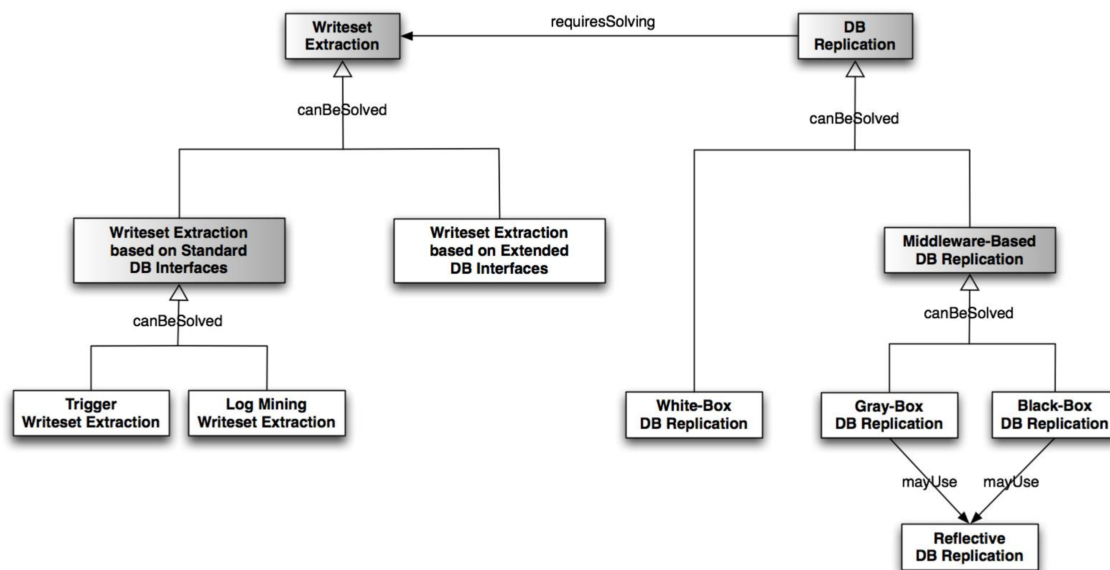
The other sub-domain contains the patterns that perform replication of several tiers:

- **Horizontal Replication with Replication Awareness**: The Horizontal Replication pattern provides high availability and scalability for applications deployed on multi-tier architectures by replicating each tier independently. The Replication Awareness pattern helps in introducing awareness of replication in the different tiers when the Horizontal Replication pattern is used.
- **Vertical Replication**: The Vertical Replication pattern aims at providing high availability and scalability for applications deployed on multi-tier architectures using only one replication protocol at the application server tier.

With regard to the other relationships described in D7.2c, two of them are used. Both, V*ertical Replication* and *Horizontal Replication* patterns *mayUse* the *Session Replication with Multi-Tier Coordination* pattern in order to provide session replication for the clients of the resulting architecture. Moreover, the *Horizontal Replication* pattern must use (*uses* relationship) a replication solution provided in the **DB Replication** domain.

Database Replication Patterns Domain

The following figure presents the patterns in this domain and how they are interrelated.



The *requiresSolving* relationship that departs from the *DB Replication* problem to the *Writeset Extraction* problem states that this problem must be addressed by the solutions of the *DB Replication* problem. It also marks the distinction of two different sub-domains in the figure above. The first sub-domain covers the patterns related to writeset extraction in databases. A writeset represents the data accessed and updated in the context of a transaction. When performing database replication, the replicas where these changes were produced must extract these data and send them to the rest of the replicas. The patterns in this

creative commons

sub-domain present different alternatives for extracting that information and have been categorized using the *canBeSolved* relationship and an additional problem box (*Writeset Extraction based on Standard DB Interfaces*):

- **Trigger Writeset Extraction**: This pattern describes a solution for writeset extraction based on the trigger mechanism, a standard method in database management systems that can be used in database replication.
- **Log Mining Writeset Extraction**: This pattern describes a solution for writeset extraction based on the log mining mechanism, a standard method in database management systems that can be used in database replication.
- **Writeset Extraction Based on Extended DB Interfaces**: This pattern describes a solution for writeset extraction based on the implementation of an extended interface that can be used in database replication.

The second sub-domain covers the different alternatives to perform database replication that have been also categorized through *canBeSolved* relationships and an additional problem box (*Middleware-Based DB Replication*). These are basically the following:

- **Black-Box Database Replication**: This pattern describes a replication mechanism for databases outside the database kernel that does not need to access the database code.
- **Gray-Box Database Replication**: This pattern describes an efficient replication mechanism for databases outside the database kernel that requires access to the source code.
- **White-Box Database Replication**: This pattern describes an efficient replication mechanism for databases implemented in the database kernel that requires access to the source code.

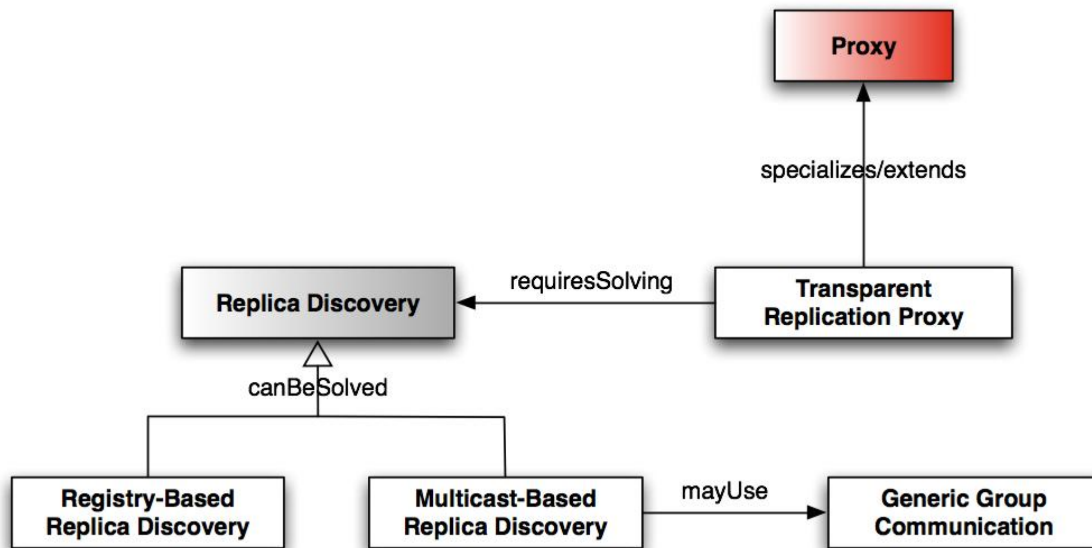
Finally, the last pattern does not form a sub-domain by itself, and presents a technique applicable to some of the patterns in the Database Replication sub-domain in order to improve the maintainability of the implemented solutions. This has been expressed through *mayUse* relationships that depart from the gray-box and black-box DB replication approaches.

- **Reflective Database Replication**: This pattern allows independent design and implementation of DBMS servers and replication protocols, allowing pluggable modules with different consistency and availability trade-offs, while at the same time fostering more efficient and maintainable implementations.

Helper/Low Level Replication Patterns Domain

The following figure presents the patterns in this helper domain and how they are interrelated.

creative commons



In here, three sub-domains can be also identified. The first one includes the well-known proxy pattern and a specialization that provides replication transparency to the clients (specified through a *specializes/extends* relationship):

- **Proxy**: This is a well-know design pattern that "makes the clients of a component communicate to a representative rather than to the component itself. Introducing such a placeholder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access" [BMRS+96].
- **Transparent Replication Proxy**: The Transparent Replication Proxy pattern is a specialization of the well-known Proxy pattern. It can be used in clients when the server part is replicated in order to provide them replication transparency and transparent failover.

The second sub-domain includes the patterns for performing the discovery of replicas. The *requiresSolving* relationship that departs from the *Transparent Replication Proxy* pattern to the *Replica Discovery* problem means that this pattern requires one of the following patterns in order to solve the replica discovery problem:

- **Registry-Based Replica Discovery**: The replica discovery pattern decouples the client from the particular set of nodes where the replicated service is running. With this pattern, clients look up connection information in well-known registry or registries that are kept updated with the current list of available replicas.
- **Multicast-Based Replica Discovery**: This replica discovery pattern decouples the client from the particular set of nodes where the replicated service is running. Following the Multicast-Based Replica Discovery pattern, a multicast service must be used.

Finally, the last pattern can be considered part of a sub-domain that has to do with the communication among replicas:

creative commons

- **Generic Group Communication**: This pattern defines a generic interface that may be used to wrap multiple group communication toolkits.

The *Multicast-Based Replica Discovery* pattern described above *mayUse* this pattern if the solution requires using several group communication systems.

Instantiation Process

This section helps system and application architects in order to derive the proper system architecture with regard two main non-functional aspects/requirements, high availability and scalability, when they are critical for that particular architecture.

This is achieved by means of an instantiation process. This process provides to the architects a set of steps to follow –the different phases of the process- in order to enhance an architecture that lacks one or both of these requirements by means of applying the most appropriate patterns from the set of patterns described in the previous section taking into account the existing trade-offs. In the end, the aim is to fulfil the non-functional requirements for the resulting architecture without disrupting the functional requirements.

In order to make the instantiation process less subjective and more reliable, the architect can configure (if possible) a group of experts in order to assess the instantiation process. From this point on, this group will be known as the *evaluation/assessment team*.

Taking this into account, from this point on, the main **prerequisites** for an architect before continuing reading, is to have as input parameters for the process both, **1)** *the list of requirements* (both functional and non-functional) and **2)** *a first functional architecture* of the desired resulting system. That is, once a system architect has compiled and discussed with the stakeholders the requirements for the specific resulting system, she/he requires to instantiate the NEXOF-RA for producing a first functional architecture (or maybe more that one) for the system he wants to build, taking into account in the design process the functional aspects to fulfil. In addition to these two prerequisites, a set of operative scenarios can be useful in order to contextualize some of the requirements in the list and prepare tests for their validation.

As the initial architecture(s) only addresses functional requirements, still does not address the high availability and scalability. The following phases will guide her/him in the process of selecting the most appropriate HA and scalability patterns for the desired system obtaining in the end an enhanced architecture with these two non-functional requirements. If the high level functional architecture is too complex, is recommended first to split it in several subsystems in order to derive the right high available and/or scalable architecture for each one of them. The phases can be sum up as follows:

1. Confront the assumptions made by the different patterns in terms of architecture (e.g. a multi-tier architecture or the use of a database component) against the initial functional architecture to be enriched with high availability and/or scalability. This will help the architect in selecting

creative commons

the most appropriate pattern domain/s described in Section "Pattern Categorization".

2. Given a pattern domain and the prioritization of the non-functional requirements for the architecture to be instantiated in this phase, a trade-off analysis is performed to select the most appropriate pattern from the domain.
3. For some applications, some of the non-functional requirements are expressed quantitatively (e.g. response time below 10 ms). After improving the functional architecture to deal with qualitative non-functional requirements, it is needed to evaluate whether the architecture will fulfil as well the quantitative non-functional requirements. Three different methodologies are proposed to achieve this evaluation with an increasing level of accuracy and effort: 1) analytical evaluations; 2) simulations and 3) proofs of concept.

At the end of phase 3, if all the input requirements can be fulfilled, the process is completed. Otherwise, it will be necessary for the architect to re-think and re-structure the initial architecture passed as input and return to phase 1.

The next figure (Figure 5) offers an illustrative schema of the whole process and the following sections offer the details on the three main phases.

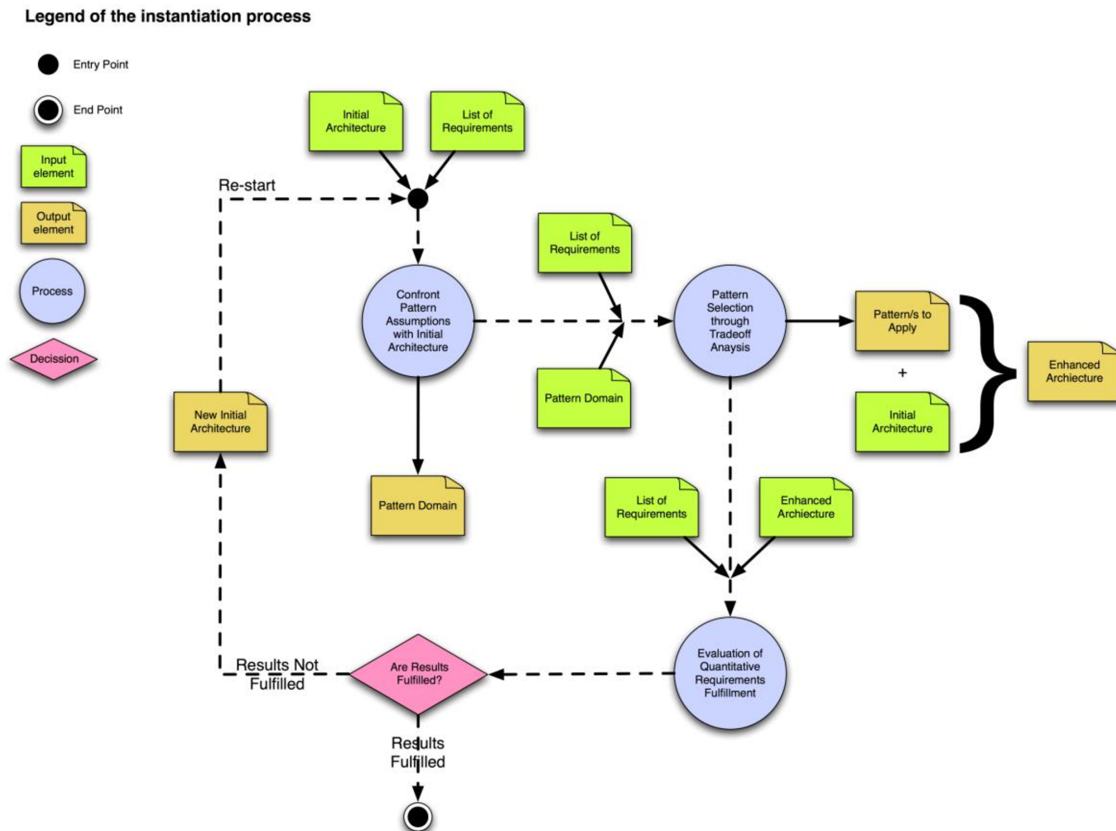
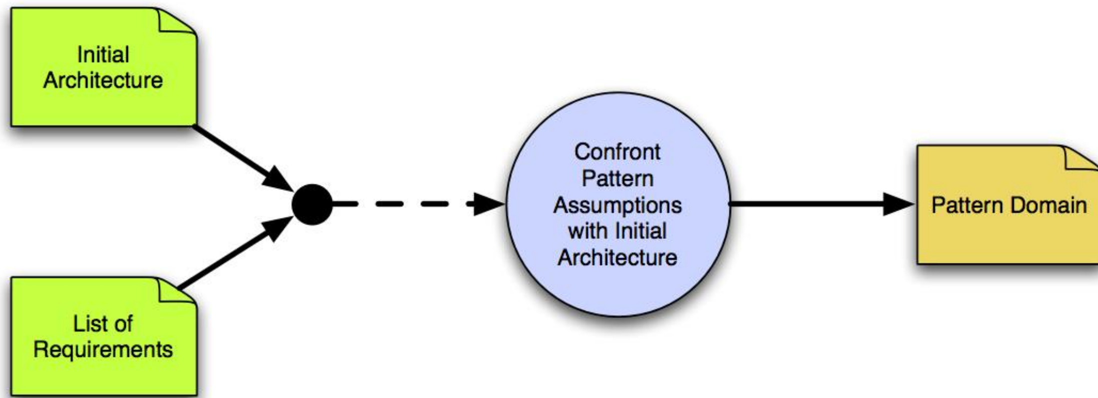


Figure 5 Schema of the instantiation process

creative commons

Phase 1: Confront Pattern Assumptions with Initial Architecture

The following figure specifies the input and output elements in this phase:



As it is shown in Figure 5, this is the entry point to the instantiation process of the high available and/or scalable architecture.

The two input parameters are:

1. **Initial List of Driving Architectural Requirements**. It is the list of requirements (both, functional and non-functional) for the system to be built. It must be configured in the different project meetings with the involved stakeholders. In this list must be identified what are the non-functional quality attributes to take into account and the quantitative requirements for those that are critical and measurable. It is also important to involve the stakeholders in the prioritization of these elements in the list in order to know the most important non-functional attributes that the resulting architecture must fulfil. In order to obtain the requirements, ATAM's step 5 –"Generate Quality Attribute Utility Tree"- may be used. In this step the involved stakeholders (mainly customer representatives, the architect/s and project managers), identify, prioritize and refine the quality attributes that are required to accomplish the project goals. The output is what is called in ATAM a *utility tree* that corresponds to our list of driving architectural requirements. Utility trees are a mechanism that allows translating the business requirements of the system into quality attribute scenarios and helps in concretize and prioritize them.
2. **Initial High Level Functional Architecture**. It describes an initial proposal for the architecture of the system developed by the architect. In order to build it, she/he has taken into the functional requirements of the stakeholders but still has not taken into account the non-functional requirements related to high availability and scalability. If the initial architecture is too large, it is recommended to divide it into different subsystems and apply the steps of the instantiation process on each one of them.

creative commons

At the end of this phase, the architect will have identified the main pattern domain from which select the most appropriate patterns (See Section "Pattern Categorization").

In order to share to share a common base of knowledge, it is necessary to offer to the evaluation team an overall view of the initial system in conception. So, first of all, the architect must present the requirements identified by the stakeholders.

Once the requirements have been presented, it will be necessary to contextualize them on top of the initial architecture, taking into account their priorities. This means to identify the critical hot spots or components where those requirements impact the architecture. In order to do this, the architect may describe to the evaluation team the proposed global initial architecture at the proper level of detail, focussing on how he plans to address the business drivers, for example, high availability, time to market, integrability, interoperability or security. For this purpose, the proper level of detail means for example a block/component diagram identifying the main architectural elements related to the system functionality and maybe how are planned to be deployed.

After presenting the overall view, as this process addresses just high availability and scalability, the architect will offer an overview of the architecture focusing mainly on which parts of the architecture he thinks will affected by these two requirements. At the same time, the architect may also present where the other driving architectural requirements (e.g., security, modifiability, interoperability, integrability) may impact on the achievement of high availability and scalability for the functional architectural elements presented before.

At this point, if the architect or any other member of the evaluation team detects that conflicts may arise with other requirements, he can also offer his opinion to the other members starting an open discussion.

After presenting and discussing the requirements on top of the initial architecture, it is necessary to identify and/or define the new additional structures/architectural approaches for the system that will be critical to allow it to grow/scale and be high available. An implicit requirement of this task is try to keep the architecture adaptable smoothly to further changes that may arise. However, once they have been defined, these approaches will not be analyzed in detail at this point.

Finally, in order to obtain the required output for this first phase of the instantiation process (i.e. the main pattern domain for the initial architecture) the architect must check the current structural requirements of the initial architecture (joint with the ones introduced by the architectural approaches identified with the evaluation team in the previous step), against the technical functional/structural requirements that imply the use of HA and scalability patterns. In order to guide the architect in this task, we have developed the simple diagram shown in Figure 6.

creative commons

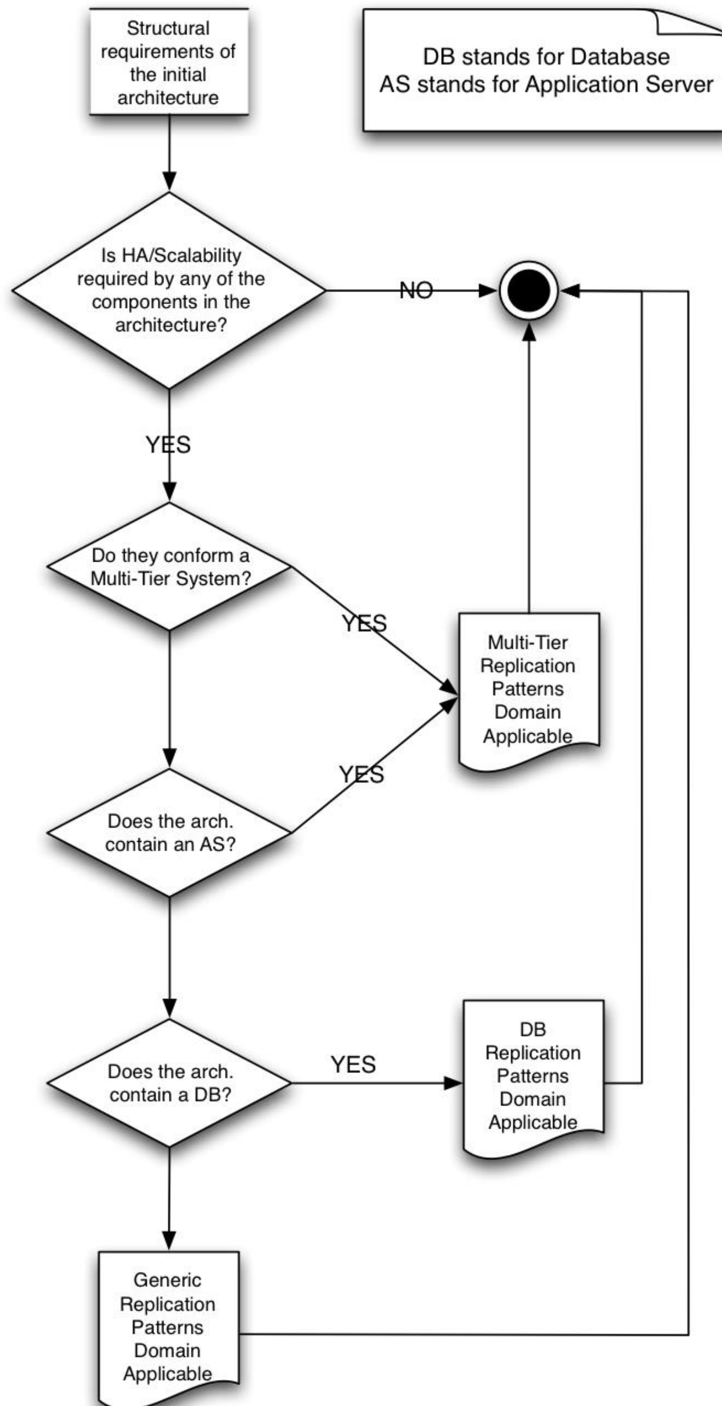


Figure 6 Diagram for selecting the pattern domains

If the main domain obtained is either Multi-Tier or DB, maybe it is necessary to check if it can be combined with any other pattern of the Generic Replication domain. The diagram in Figure 7 helps in deciding if patterns of the Generic Replication domain are also required in the resulting architecture.

creative commons

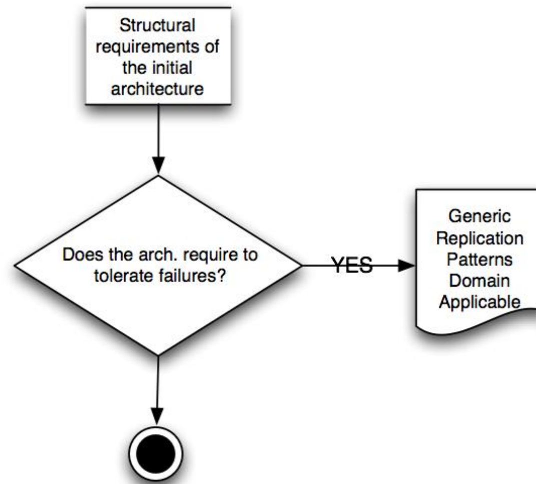


Figure 7 Diagram for deciding if the Generic Rep. Domain is also applicable

Finally, the following diagram (Figure 8) helps in deciding if the domain of Helper Replication patterns is also applicable:

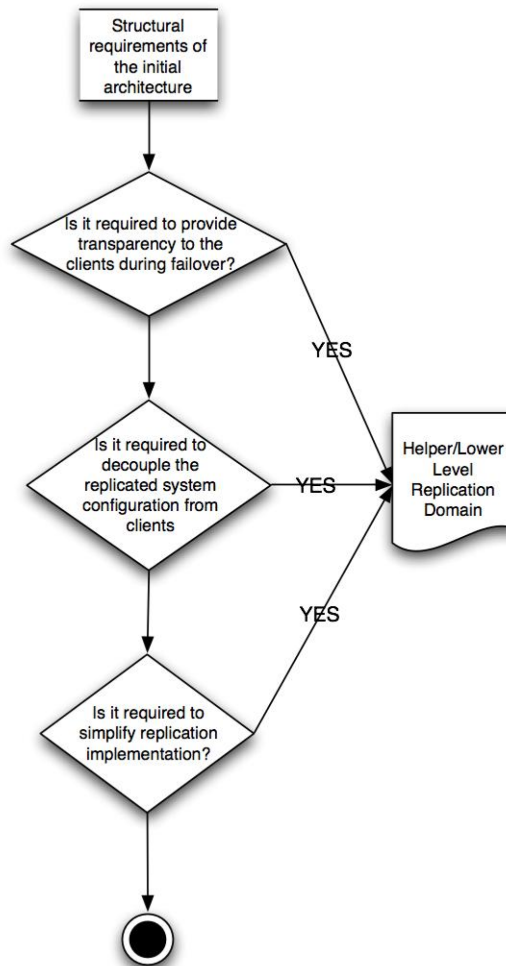
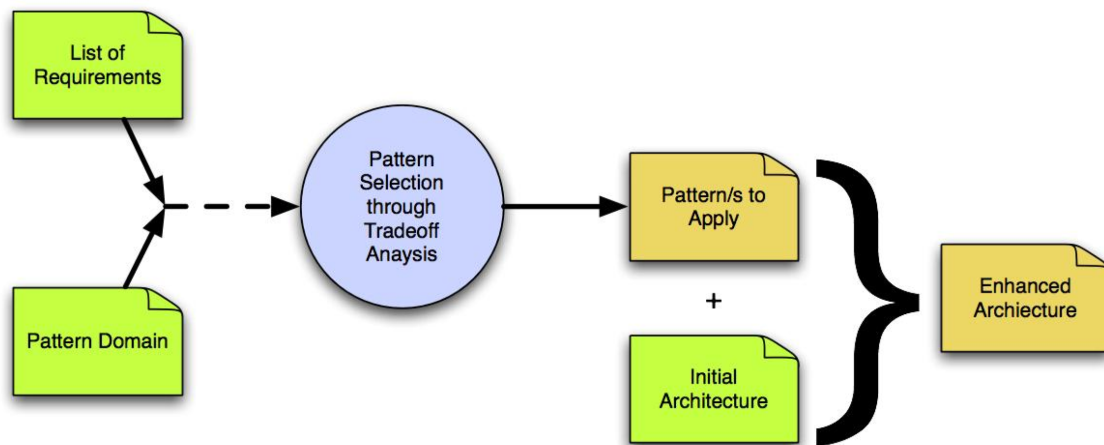


Figure 8 Diagram for deciding if the Helper/Lower Level Rep. Domain is also applicable

creative commons

Phase 2: Pattern Selection Through Trade-Off Analysis

The second phase is shown graphically in the next figure extracted from Figure 5:



This phase consists in using the list of desired qualitative requirements for the target system in order to select the most appropriate patterns from the domains identified in phase 1. The architect must perform the selection of pattern/s comparing this list with the non-functional quality attributes offered by the patterns. At the end of this process, the initial proposed architecture will be enhanced by applying the identified high availability and scalability patterns. Depending on the sensivity points and tradeoffs identified, it is possible to obtain more than one enhanced architecture (produced by applying different patterns) ready to be validated in phase 3.

The first task that the architect must perform is to filter just those attributes of the input list of quality attributes that are related to or can be affected by high availability and scalability.

The list of qualitative input requirements can be potentially long and use different terminology. For the sake of helping the architects in comparing the input requirements, we have identified in each one of the high availability and scalability patterns that quality attributes affected[2]. The following is a list describing each one of them:

- **Scalability**. This term refers to a desirable property of a system or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged. There are basically two ways of scaling a system: scale-up and scale-out. Scale-up refers to the ability of single node system to increase its computing/storage capacity (i.e. increase its throughput) adding more resources, such as CPUs, memory, disks, etc. On the other hand, scale-out means the ability of a distributed system to increase its computing/storage capacity

[2] The NEXOF-RA quality model provides an extensive list of quality attributes that can be taking into account when instantiating NEXOF-based architectures.

creative commons

by adding more nodes. The patterns included in this set of patterns allow improving the architecture of a system by means of scale-out.

- **Availability**. It refers to the proportion of time in which a system is operational, and satisfies its specification. It is quantified as the uptime divided the uptime + downtime. Most of the patterns of this set of patterns contribute to improve availability by means of replication.
- **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.
- **Maintainability**. This term refers to the ease with which maintenance of a functional unit of a system (or the whole system) can be performed in accordance with prescribed requirements.
- **Replication Transparency**. A term used to refer to the ability of a replicated system 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 many replicated systems. It is taken into consideration in all the patterns that involve data/state replication to achieve high-available and/or scalable solutions.
- **Performance**. A quantification of the goodness of the service provided by a system. Performance involves metrics such as response time, throughput, reliability, etc.

Each individual pattern affects one or more of these non-functional quality attributes either in a positive, neutral or negative way (See Section 4 in each pattern template), so the architect will have to balance the desired requirements for the final architecture against the quality attributes of each pattern and select the one/s that he believes is/are more appropriate/s. Then, the next task is to adapt the initial architecture with the architectural changes implied by the pattern/s selected, producing an enhanced architecture ready to be validated quantitatively in step 3. Each pattern includes the required assumptions (Section 5 in the pattern template) and the rules to follow (Section 6 in the pattern template) in order to guide the architect in transforming the current architecture into the one enhanced with the pattern features.

Along this process of architecture enhancement, the highest priority quality attributes drive the process of selecting the patterns. The output of applying the patterns to the initial architecture is a set of sensivity points, tradeoff points, risks and non-risks identified for each of the resulting enhanced architecture/s. Sensivity points are properties of one or more components and their

creative commons

relationships that are critical for achieving a particular quality. For example the availability of a system may be highly correlated to the reliability of a particular communication channel. Tradeoffs arise in the architecture "when a parameter of an architectural construct is host to more than one sensitivity point where the measurable quality attributes are affected differently by changing that parameter". For example, when increasing *the speed* of the previous communication channel, we can improve the throughput but we can reduce the reliability. Both, sensivity points and tradeoffs are usually translated into risks that affect the architecture.

Before passing the enhanced architecture/s to the validation performed in the third phase of the instantiation process, the results obtained should be checked against well-known topologies, anti-patterns, best practices etc. for high availability and scalability described in the existing bibliography (e.g. [Tate02, AF09]) to identify potential issues. In this way, the risks and potential issues identified can be used to discard certain resulting enhanced architectures derived from this phase, avoiding the corresponding evaluations.

So, last but not least, it is also important to start to identify and discuss what will be the metrics related with high availability and scalability that will be taken into account, identify the points in each enhanced architecture/s where they will be obtained, and any existing standards/models/approaches for meeting them.

The following sub-sections offer to the architects the trade-offs to take into account in each pattern domain related to high availability and scalability (See Section "Pattern Categorization").

Trade-offs for the Generic Replication Pattern Domain

The main goal of these two general patterns (*Active Replication* and *Passive Replication* patterns) is to provide high availability to critical components of an infrastructure by means of redundancy. They use several replicas of the critical component in order to mask failures, keeping the system online. In the following paragraphs, the trade-offs related to quality attributes are discussed.

With regard to performance, in case of the *Active Replication* pattern, the performance of the system is not altered by the pattern, so it is does not imply any trade-off. The overhead introduced by this pattern in terms of time penalty is low because it does not introduces any synchronization overhead among the replicas in both, error-free and failure scenarios. Considering the trade-offs with regard applicability –that is, the number and strength of assumptions that must be taken into account in order to apply the pattern- the *Active Replication* pattern restricts a little bit the applicability, because it is only applicable to stateless components or stateful components that behave *deterministically*. Finally, with regard to maintainability, the *Active Replication* pattern introduces a low complexity when implementing the replicated system solutions. Just a distributor is necessary to spread the request to all the replicas and a comparator to collect the responses. These components can be embedded in all the replicas of the critical component, being active in only one of them.

However, when using the *Passive Replication* pattern with stateful components

creative commons

the performance can be affected depending on the implementation. If the primary replica sends to the backups the system state each time it is changed and must wait for receiving an acknowledgement message from the backup replicas in order to continue processing requests (synchronization), the performance can be affected negatively. If asynchronous messages are used for this purpose from the primary to the backups, the performance should not be altered. In this case the pattern can be applied to both kinds (stateful and stateless) of components without restrictions. Finally, the trade-offs related to maintainability are also minimal because only a Coordinator component is required in order to assign the roles of primary and backups in the set of replicas. This component can be embedded in all the replicas being only active in the replica chosen as primary.

Trade-offs for the Database Replication Pattern Domain

The 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 order to extract the data to be replicated into a set of replicas (called writesets). This first sub-domain of patterns related to writeset extraction includes the following trade-offs.

The *Trigger Writeset Extraction* pattern relies on the facilities provided by the trigger mechanisms included in almost all relational databases, both commercial and open-source or commercial, what impacts positively the pattern applicability Moreover, the internals of the target database does not need to be modified in order to implement the writeset extraction what can be taken into account because improves the maintainability of the solution. However, this mechanism was not originally implemented for writeset extraction, what impacts negatively the performance of the implemented solutions. Triggers are heavyweight and when activated frequently as in the case of writeset extraction they consume excessive computing resources.

The *Log Mining Writeset Extraction* pattern shares the same trade-offs as the previous pattern. In this case, the log mining mechanism is not as common as triggers, but the most important relational databases include them tools to inspect the log. Also, in this case, implementing this pattern does not require modifying the internals of the database component. The performance of the solutions can be impacted negatively because is expensive in terms of computing resources consumption to extract the writesets with this mechanism (it is not devoted to this function).

On the other hand, the *Writeset Extraction Based on Extended DB Interfaces* pattern offers a great performance in writeset extraction/injection. This pattern implements a well-defined interface in the database code, to access the writesets of transactions. The ad-hoc implementations of this pattern, is what

creative commons

allows increasing the performance of the solutions with regard the other two patterns. However, this is also the reason why the applicability is restricted. Unfortunately, most database vendors do not offer this interface as a de facto feature in their products. In this way, this pattern can be only applied to those databases that provide the source code of the internals, what restricts the applicability mainly to open-source databases. However, once the interface is provided, the maintainability of the final solution is not affected, because the replication middleware that the interface is not affected by the changes produced in the database internals.

With regard to the database replication patterns, i.e. *Black-Box*, *Gray-Box* and *White-Box Database Replication*, the following trade-offs apply.

In general, the shared feature that the three patterns allow achieving is high availability. This is done by means of replicating the databases that contain the critical data. However, each pattern affects other attributes in different ways.

The *Black-Box DB Replication* offers a modest degree of scalability. This is because the writeset extraction must be performed using standard mechanisms (See *Trigger Writeset Extraction* and *Log Mining Writeset Extraction* patterns above). These mechanisms are too heavy-weight, resulting in saving very low computing capacity when using asymmetric update transaction processing and therefore this pattern allows low scalability for update workloads. However, because of the use of these writeset extraction patterns, the applicability of the pattern is high. So, this pattern can be applied to any standard database that provides either triggers or log mining and does not require access to the DB source code. Moreover, the solutions that implement this pattern are highly maintainable. 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.

In contrast, the next pattern -*White-Box DB Replication*- offers a very high degree of scalability. This is because the implemented solutions use the *Writeset Extraction Based on Extended DB Interfaces* pattern, so the writeset extraction is performed very efficiently. 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. However, the applicability is limited. This is mainly due to two reasons; first, it requires the DB source code to be available. Because of this fact, the pattern can be applied mainly in open-source databases. In commercial databases, the writeset extraction interface must be ordered on demand to the specific database vendor; and second, it requires modifying large sections of the DB code. This is not a trivial task and requires highly skilled engineers. Finally, maintainability is difficult because it requires keeping consistent the DB code with respect the DB kernel, what is a very expensive task.

creative commons

Finally, the *Gray-Box DB Replication* also offers a very high degree of scalability. It also uses the *Writeset Extraction Based on Extended DB Interfaces* pattern, what increases the efficiency of writeset extraction compared to the two other writeset extraction patterns. As this pattern requires implementing the interfaces for writeset extraction in the database, the applicability is also limited mainly to open-source databases. On the other hand, with regard to the *White-Box DB Replication pattern*, the required extensions on the DB kernel are quite localized and small, what makes it quite feasible in most cases. This also contributes to the maintainability of the solutions implemented with this pattern. With this pattern, only the DB replication code (e.g. an external middleware outside the DB kernel) needs to be updated. Only the writeset extraction mechanism needs to be introduced/adapted in the DB kernel accordingly in order to be coherent with the rest of the system.

The goal of the *Reflective Database Replication* pattern is to use multiple instances of the database running on different nodes coordinated by pluggable DB replication protocols that can be specified depending on the application requirements (e.g. consistency constraints). This pattern can be used in combination with the *Black-Box* or the *Gray-Box DB Replication* patterns. So, in this case the degree of scalability, applicability and maintainability will depend on the chosen database replication pattern and the replication protocol implemented. Therefore, the main advantage of applying this pattern in combination with the other two patterns is related to maintainability, since it contributes to decouple the replication protocols from the underlying replication infrastructure.

Trade-offs for the Multi-Tier Replication Pattern Domain

The patterns in this domain are related to the replication of the main important tiers of multi-tier architectures (See *Multi-Tier Transactional Service Runtime* pattern). These tiers are the middle-tier (a.k.a. business or application server tier) and the data-tier (a.k.a. database tier). If only one tier is replicated, the other tier becomes a single point of failure for the multi-tier architecture what affects the availability of the solutions. So, most solutions will require replicating both tiers.

The tiers can be replicated independently or as a whole. The trade-offs of the patterns related to the independent replication of the database tier have been commented in the previous section. The other options provided by the patterns in this domain are discussed in the following paragraphs.

The first pattern related to high availability and scalability to be discussed is the *Session Replication with Multi-Tier Coordination* pattern. This pattern is related to the replication of the middle-tier. The main goal of the pattern is to enhance

creative commons

the availability and scalability of the client sessions in the middle-tier for 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. In the coordination process performed at the application server level, the replicas use the database tier as persistent storage for coordination information. Different application servers might serve different clients leading to load-distribution, and thus, potential for scalability. The pattern encapsulates the replication logic for the session components in the application server, what results in a good applicability since implementations can be based on the use of standard databases. Of course, the source code of the application server must be available to perform the required modifications. The maintainability is considered neutral since it does not require maintaining database code. It requires only maintaining the replication code within the application server.

The next pattern, the *Horizontal Replication with Replication Awareness*, provides a replication approach that replicates the middle and data tiers independently. In principle, this allows to attain high availability and scalability of both tiers, but it requires to perform extra work in order to each tier be aware of the replication of the other. As the replication of each tier is independent, it becomes absolutely necessary to perform a coordination of the replication of the elements in each tier in order to guarantee the consistency of the solution. However, this additional processing time can affect negatively the performance, and thus the scalability. In order to mitigate this, transactions can be processed following the read-one write-all (ROWA) strategy and asymmetric update processing, what reduces redundancy of transaction processing across replicas. The approach can encapsulate the replication logic of the two tiers in the application server, what results in a good applicability since the solutions can be based on standard databases. However, the use of standard databases may affect negatively the performance (e.g. using *Black-Box DB Replication* pattern). If the performance penalty at the database level wants to be avoided, the Gray-Box approach (or the White-Box if available) should be used. The maintainability is considered negative since the solution requires maintaining both, the implementation of application server and database replication mechanisms.

Another option is presented in the *Vertical Replication* pattern. This pattern presents a holistic replication approach that replicates all tiers simultaneously. As in the previous pattern, the aim is to enhance high availability and scalability of middle and data tiers. The solutions can also use transactions processed following the read-one write-all (ROWA) strategy and asymmetric update processing in order to improve the scalability. This reduces redundancy of transaction processing across replicas. In contrast to the previous approach, only one replication mechanism is required. The approach encapsulates the replication logic in the application server, what results in a high applicability since standard databases can be used. The maintainability is considered

creative commons

neutral since it does not require maintaining database code, but still requires maintaining the replication code within the application server.

Trade-offs for the Helper/Lower Level Replication Pattern Domain

As high availability and scalability are common requirements in current service-oriented applications, many system architectures are being replicated in order to achieve them. The patterns in this domain contribute/help in achieving simplify or guarantee other requirements in replicated architectures related closely to high availability and scalability, so their trade-offs are discussed in the following paragraphs.

For example, it is very important that applications could run on the replicated architecture transparently. This means that the application should not be aware about if the underlying infrastructure is replicated or not. The *Transparent Replication Proxy* pattern helps in achieving this task. It is a specialization of the well-known Proxy pattern that allows clients to transparently tolerate node crashes, attaining high availability. It basically includes the required logic to mask the failures to the clients. It can be used also to improve scalability, because the proxy can connect transparently to the most appropriate replica in each case. For example, the client can be redirected to that replica that is less loaded when client request arrive. The applicability of the pattern can be considered very high, because this pattern can be applied to the client side of any component susceptible to be replicated (e.g. application servers, databases etc.). The maintainability of the transparent proxies is considered neutral, since in many cases it is necessary both, to update the proxy code and to coordinate the underlying replication mechanism for components in order to tolerate their crashes.

Another important task in replicated systems is the communication among the replicas. Point-to-point communication can be used in many cases, but it is not always the best option, for example when considering cluster of replicas running in local area networks (LANs). Moreover, point-to-point and ad-hoc protocols contribute to increase the complexity of the system. Group communication is a coordination paradigm that eases the development of multi-participant applications. Currently there are a lot of group communication toolkits in the market. However, each toolkit offers a different interface, which differs from every other interface in subtle syntactic and semantic aspects, impacting the design of applications using these features. To solve this problem, the *Generic Group Communication* pattern defines a generic interface that may be used to wrap multiple toolkits decoupling the application from the specific toolkits. As the pattern provides a generic interface in order to manage group communication, it contributes to the enhancement of the applicability of solutions implementing this pattern, In this way, any application that require group communication functionality just has to access a single common interface that hides the specific interfaces of particular group communication toolkits. Maintainability is also considered enhanced. When an application using the generic group communication interface requires a specific group communication

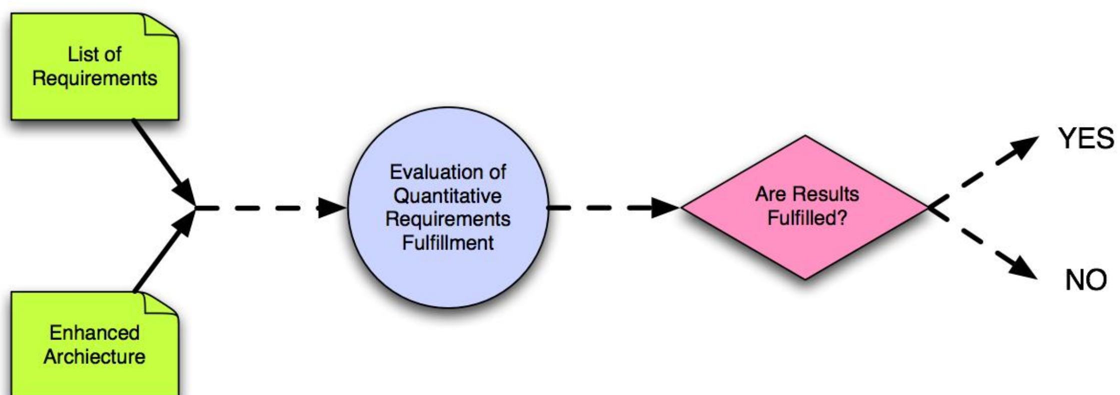
creative commons

toolkit, the changes required in the current implementation are avoided. The only changes in source code are related to the adaptation of the toolkit functionality to the generic interface.

Finally, in replicated systems, it is also necessary that the clients would be able to locate the list of currently available replicas of a particular service or component. The replica discovery patterns help in this task, decoupling the client from the particular set of nodes where the replicated service/component is running. These are the trade-offs of the two patterns proposed.
Both patterns, *Registry-Based Replica Discovery* and M*ulticast-Based Replica Discovery* contribute to the transparency of the replicated infrastructures. With regard to the applicability, in both patterns can be considered as high, because they are useful in order to locate any component susceptible to be replicated (e.g. application servers, databases etc.). Due to the multicast requirements, the *Multicast-Based Replica Discovery* pattern offers more performance when is used in local area environments, but it can also be used in WANs. Moreover, this pattern does not incur in single points of failure, because the discovery mechanism is implemented in all the replicas. On the other hand, the *Registry-Based Replica Discovery* provides a central repository that includes all the information of the available replicas. In this case, the information can be collected using push or pull mechanisms depending on the application requirements. This pattern is more suitable for WAN environments (but it can be used also in local environments) because the replicas can refer to a single component across the Internet in order to find the available replicas of a service or component. However, if the registry itself is not replicated, it becomes a single point of failure. The registry can store the information about the replicas in a persistent storage in order to improve durability. All these tasks increase the maintainability of the solutions that implement this pattern, what can be avoided using the multicast-based approach.

Phase 3: Evaluation of Quantitative Requirements Fulfilment

Finally, the third phase of the process is shown graphically in the following figure:



creative commons

This third phase of the process consists in checking if the resulting alternatives of high available and scalable architectures obtained in phase 2, fulfil the quantitative input requirements. Most part of the quality attributes of the patterns for high availability and scalability can be quantified in some way or another. For example we can quantify performance in terms of the number of completed transactions per second (Tx/Sec).

There are three main techniques to evaluate the quantitative requirements of a system (See Section 2.7.1 of the methodology for writing instantiation guidelines):

1. Analytically
2. Using simulations
3. Implementing prototypes and Proofs of Concept (PoC)

After performing the required evaluation/s we'll get results confirming or not the quantitative requirements. If the results are good enough with regard the input requirements, the process is finished. On the other hand, if the non-functional quantitative requirements are not going to be fulfilled with the resulting architecture, it is recommended to check the initial high-level system architecture used as input in the first phase of the process in order to be re-thought and re-structured.

The following subsections show of how High Availability and Scalability can be evaluated in the resulting architectures using the techniques described in Section 2.7.1 of the methodology for writing instantiation guidelines.

Evaluating High Availability

A system that is high available ensures that it will continue being operational (running/processing information) during a period (in which it is said that the system is uptime) with a high degree of probability. When a failure occurs, the system is said to be downtime.

A common way to express availability is to denote it as the percentage that a system is uptime and running in a given year. The *mean time between failures* (MTBF) measures the elapsed time between failures produced in an uptime and running system. This measure assumes that the system under test is recovered when it fails. In contrast, the *mean time to failure* (MTTF) does not assume this. In order to measure high availability, simulations and analytical techniques are basically used. Through these techniques not only the system can be described, but also the environment can be recreated, simulating failures and recoveries of the system under test in order to measure MTBF/MTTF. It is not possible to use prototypes to measure parameters such as MTBF/MTTF. Of course this is due to that running forever a system in a testing environment is neither affordable nor a realistic approach for stakeholders. However, prototypes can be used in order to apply and test architectural choices and solutions that imply patterns related to redundancy for system

creative commons

availability/recoverability, such as the ones described in the PoCs of NEXOF-RA related to high availability and scalability [FPP]. For example, one of the PoCs validates three different patterns for writeset extraction. Writeset extraction is a process that allows extract database data from a DBMS in order to provide high availability. So, this PoC evaluates different architectural choices by means of prototypes of each one of them in top of different DBMSs, measuring the quality attributes affected (i.e. performance, applicability and maintainability). The results of the PoC show that there is a trade-off between applicability and maintainability, and performance when applying the different patterns proposed by means of prototypes.

Evaluating Scalability

Scalability is "a desirable property of a system … which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged" [NG].

In order to measure scalability, a common measure used is the relative throughput of the distributed system with respect the original centralized systems. For example, in transactional systems, the throughput is measured as the number of transactions that the system is capable to process per second (Tx/sec). The analytical techniques, simulations, prototypes and PoCs can be used to evaluate the scalability of a system.

For example, in [JPAK03] is described an analytical study that shows the scalability limits of full data replication in databases with eager techniques in update-everywhere database clusters using symmetric and asymmetric processing. Update-everywhere means that all the replicas can process update transactions. Eager means in this case to propagate changes produced by each transaction in a replica, before the transaction commits. In symmetric processing all update transactions are completely executed in all the replicas, whilst in asymmetric processing an update transaction is first executed at one replica and the changes are replicated and applied in the rest of them without processing the complete sentence.

Also in [SPJK07] is described an analytical evaluation of partial replication in databases with respect to full replication. In order to do so, a mathematical model for quantifying the *scale-out* [NG] (i.e. how many times the replicated system increases the performance of a non replicated system) has been built.

PoCs related to high availability and scalability [FPP] have already been used in the context of NEXOF-RA to evaluate the scalability of architectural choices in the E-SOA. In these PoCs, several prototypes have been built in order to evaluate the scalability properties when applying several patterns from the set of patterns found in Section "Pattern Categorization", to certain non-replicated systems and architectures. One of the PoCs validates the *Vertical Replication* pattern combined with the *Session Replication* pattern applied in a multi-tier architecture in terms of scalability, availability, applicability and maintainability. In order to do so, a prototype of the multi-tier system has been implemented and deployed in several nodes. The prototype includes a replication protocol

creative commons

implementation at the level of the application server replicas, which inject the persistent changes in standard database replicas. In the PoC results is shown that despite the good scalability provided by the replication both, the *Session Replication* 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.

Finally, there are two PoCs that validate the replication of the database layer (accessed by an E-SOA web application) through the *Gray-Box DB Replication* pattern in two different environments: LANs and WANs. In both contexts, the validation is done in terms of availability, scalability, applicability and maintainability over a DBMS prototype that includes a replication middleware and extracts transaction writesets using extended interfaces, deployed in several nodes. The results of the PoCs in the two different environments show improvements in availability, scalability and performance of the replicated system with respect to a centralized approach, but also 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.

The Role of ATAM in the Instantiation Process for HA and Scalability Properties

Certain steps and parts of the *Architectural Trade-off Analysis Method* (ATAM) [KKC00] have been taken into account in this process (being integrated or adapted) in order to make it more robust. ATAM methodology was originally developed to assist architectural decisions by taking into account the quality attributes early in the design process. ATAM is one of the most used industrial practices in order to evaluate software architectures.

The adoption of ATAM is motivated by the fact that, as other Scenario-based evaluation techniques[3], it fits well in the architecture definition phase where these guidelines are included. Moreover, despite the steps that describe ATAM are numbered sequentially, ATAM is not a waterfall process, so they are used/suggested/adapted in the process described in Section "Instantiation Process" when required.

The following features and steps of ATAM described in the paragraphs below have been taken into account in the instantiation process.

In the first phase of the process (Section "Phase 1: Confront Pattern Assumptions with Initial Architecture"), in addition to the *utility tree*, two additional steps of the ATAM methodology can help the architect to accomplish this step; ATAM's step 3, "Present the Architecture", and step 4, "Identify Architectural Approaches".

---

[3] A review and comparison among five well-adopted scenario based evaluation techniques (i.e. SAAM, ATAM, CBAM, ALMA, FAAM) is provided in [IHO02].

creative commons

The third step of ATAM can help the architect in presenting the architecture to the evaluation team, whilst the fourth step may help the architect in identifying possible architectural approaches and styles that can be applied to the initial architecture presented. ATAM states that these approaches and styles "represent the architecture's means of addressing the highest priority quality attributes; that is, the means of ensuring that the critical requirements are met in a predictable way [BMRS+96, SG96]". So, in the case of this guidelines, the main critical requirements will be related to HA and scalability.

When filtering the quality attributes in the second phase of the instantiation process (Section "Phase 2: Pattern Selection Through Trade-Off Analysis"), the ATAM methodology offers an extensive characterization and categorization of quality attributes that may help the architect in identifying those ones related to high availability and scalability.

Finally, in ATAM's step 6 "Analyze Architectural Approaches" is stated: "the key though to keep in mind is the need to establish some link between the architectural decisions that have been made and the quality attribute requirements that need to be satisfied". This step of ATAM can be used in the second phase of the process in order to check if the all the requirements have been taken into account in the enhanced architecture/s that is/are the output of that phase.

creative commons

ACRONYMS

ATAM: Architectural Trade-off Analysis Method

DB: Database

DBMS: Database Management System

E-SOA: Enterprise Service-Oriented Architecture

HA: High Availability

IoS: Internet of Services

LAN: Local Area Network

NCI: NEXOF Compliant Infrastructure

NCP: NEXOF Compliant Platform

NEXOF-RA: NEXOF Reference Architecture

MTBF: Mean Time Between Failures

MTTF: Mean Time To Failures

PoC: Proof of Concept

WAN: Wide Area Network

creative commons

References

[NG] NEXOF-RA WP-6, Glossary - **The NEXOF Glossary**. NEXOF Deliverables, available at http://www.nexof-ra.eu/?q=node/187

[NRM] NEXOF-RA WP-6, Deliverable 6.3 - **The NEXOF Reference Model V3.0**. NEXOF Deliverables

[ESO] NEXOF-RA WP-7, **Enterprise SOA Pattern.** NEXOF Deliverables

[AFP] NEXOF-RA WP-7, Deliverable 7.2c - **Definition of an Architectural Framework & Principles.** NEXOF Deliverables

[PPSC] NEXOF-RA WP-8. D8.0 - **Processes, Principles and Selection Criteria behind PoC**. NEXOF Deliverables

[FPP] NEXOF-RA WP-8. D8.1 – **First Phase PoCs**. NEXOF Deliverables

[RR] NEXOF-RA WP-10, Deliverable 10.1 - **Requirements Report**. NEXOF Deliverables

[KKC00] R. Kazman, M. Klein, P. Clements. ATAM: Method for Architecture Evaluation. Technical Report, CMU/SEI 2000-TR-004 ESC-TR-2000-04. August 2000, available at http://www.sei.cmu.edu/reports/00tr004.pdf

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

[BMRS+96] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture - A System of Patterns. Wiley, 1996

[PVPJ06] F. Perez-Sorrosal, J. Vuckovic, M. Patiño-Martínez, R. Jiménez-Peris. Highly Available Long Running Transactions and Activities for J2EE Applications. IEEE Int. Conf. on Distributed Computing Systems (ICDCS), 2006

[SPJK07] D. Serrano, M. Patiño-Martinez, R. Jimenez-Peris, B. Kemme. Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. 13th IEEE Pacific Rim Dependable Computing Conf., 2007

[SG96] Shaw, M. and Garlan, D. Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, NJ: Prentice-Hall, 1996

[Tate02] B. A. Tate. Bitter Java. Manning, 2002

[AF09] M. L. Abbot and M. T. Fisher. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Addison-Wesley Professional, 2009

[JPAK03] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are Quorums an Alternative for Data Replication? ACM Transactions on Database Systems (TODS), Vol. 28, N. 3, pp. 257-294, ACM Press. Sept. 2003

[BJPQ+08] Roberto Baldoni, Ricardo Jimenez-Peris, Marta Patiño-Martınez, Leonardo Querzoni, Antonino Virgillito. Dynamic Quorums for DHT-based Enterprise Infrastructures. Journal of Parallel and Distributed Computing (JPDC). Vol. 68. pp. 1235-1249. 2008

creative commons

[BJPQ+05] R. Baldoni, R. Jiménez-Peris, M. Patiño-Martínez, L. Querzoni and A. Virgillito. Dynamic Quorums for DHT-based P2P Networks. 4th IEEE Int. Symp. on Network Computing and Applications (NCA). Cambridge, MA, USA. July 2005

[IHO02] M. Ionita, D. Hammer, H. Obbink: Scenario Based Software Architecture Evaluation Methods: An Overview. Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering, Orlando, Florida, USA. 2002