Grant Agreement N° 215483

Title:	*CD-JRA-2.3.2: Basic Requirements for self-healing services and decision support for local adaptation*
Authors:	*INRIA, SZTAKI, TUW*
Editor:	*Jean-Louis Pazat (INRIA)*
Reviewers:	*Fabrizio Silvestri (CNR)*
	Olivier Barais (INRIA)
Identifier:	*Deliverable #CD-JRA-2.3.2*
Type:	*Deliverable*
Version:	*1*
Date:	*March 16, 2009*
Status:	*Final*
Class:	*External*

Management Summary

One of the goals of S-Cube is to look for general solutions by integrating research agendas from diverse research areas, such as business processes, service-oriented and grid computing. The world of web services already provides solutions for complex user tasks. The web service model is based on three actors: a service provider, a service requester and a service broker. There are also well established and widely used technologies that enhance the collaboration of these three parties to fulfil service executions required by users. The newly emerging demands of users and researchers call for expanding this service model with business-oriented utilization (agreement handling), support for human-provided and computation-intensive services. This evolution also affects the service infrastructure; new components appear that need to provide self-* operation.

The purpose of this document is to capture the basic requirements for self-healing and decision support in service execution, deployment and runtime management for services including core services such as discovery and registries.

Concerning service execution, we describe what kind of functionalities and tools should be provided at the infrastructure level in order to be able to implement a self-healing service. We restrict the scope of this document to the adaptation of one service, not of a coordinated set of services.

Concerning deployment and run-time management, we envision a conceptual architecture for SLA-based on-demand service provisioning and, based on this framework, three main functionalities are separated: negotiation, brokering and deployment. The document investigates the requirements in details for each of these fields.

This document mainly addresses Threads C1 and C2 of the WP 2.3 research architecture and partly applies to A1 and B1. See also the companion deliverable CD-JRA-2.3.3 which addresses service discovery and registries (Thread A2, A3, B2 and B3). C3 will be addressed in deliverable CD-JRA-2.3.8.

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 215483 (S-Cube).

Members of the S-Cube consortium:

University of Duisburg-Essen (Coordinator)	Germany
Tilburg University	Netherlands
City University London	U.K.
Consiglio Nazionale delle Ricerche	Italy
Center for Scientific and Technological Research	Italy
The French National Institute for Research in Computer Science and Control	France
Lero - The Irish Software Engineering Research Centre	Ireland
Politecnico di Milano	Italy
MTA SZTAKI – Computer and Automation Research Institute	Hungary
Vienna University of Technology	Austria
Université Claude Bernard Lyon	France
University of Crete	Greece
Universidad Politécnica de Madrid	Spain
University of Stuttgart	Germany
University of Hamburg	Germany
Vrije Universiteit Amsterdam	Netherlands

Published S-Cube documents

All public S-Cube deliverables are available from the S-Cube Web Portal at the following URL:

http://www.s-cube-network.eu/results/deliverables/

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 215483 (S-Cube).

The S-Cube Deliverable Series

Vision and Objectives of S-Cube

The Software Services and Systems Network (S-Cube) will establish a unified, multidisciplinary, vibrant research community which will enable Europe to lead the software-services revolution, helping shape the software-service based Internet which is the backbone of our future interactive society.

By integrating diverse research communities, S-Cube intends to achieve world-wide scientific excellence in a field that is critical for European competitiveness. S-Cube will accomplish its aims by meeting the following objectives:

- Re-aligning, re-shaping and integrating research agendas of key European players from diverse research areas and by synthesizing and integrating diversified knowledge, thereby establishing a long-lasting foundation for steering research and for achieving innovation at the highest level.
- Inaugurating a Europe-wide common program of education and training for researchers and industry thereby creating a common culture that will have a profound impact on the future of the field.
- Establishing a pro-active mobility plan to enable cross-fertilisation and thereby fostering the integration of research communities and the establishment of a common software services research culture.
- Establishing trust relationships with industry via European Technology Platforms (specifically NESSI) to achieve a catalytic effect in shaping European research, strengthening industrial competitiveness and addressing main societal challenges.
- Defining a broader research vision and perspective that will shape the software-service based Internet of the future and will accelerate economic growth and improve the living conditions of European citizens.

S-Cube will produce an integrated research community of international reputation and acclaim that will help define the future shape of the field of software services which is of critical for European competitiveness. S-Cube will provide service engineering methodologies which facilitate the development, deployment and adjustment of sophisticated hybrid service-based systems that cannot be addressed with today's limited software engineering approaches. S-Cube will further introduce an advanced training program for researchers and practitioners. Finally, S-Cube intends to bring strategic added value to European industry by using industry best-practice models and by implementing research results into pilot business cases and prototype systems.

S-CUBE materials are available from URL: http://www.s-cube-network.eu/

Contents

1 Introduction

1.1 WP Vision

The heterogeneity, dynamicity and ever-growing size of distributed infrastructures that will support future SBAs have reached a complexity that cannot be overseen and controlled only by humans anymore. There is an evident need for systems that are able to maintain and control themselves (at least partly) in an autonomic manner. We assume an environment of heterogeneous services (entities) e.g. any type of computing resources, storage devices, application servers, load-balancers, and resource brokers.

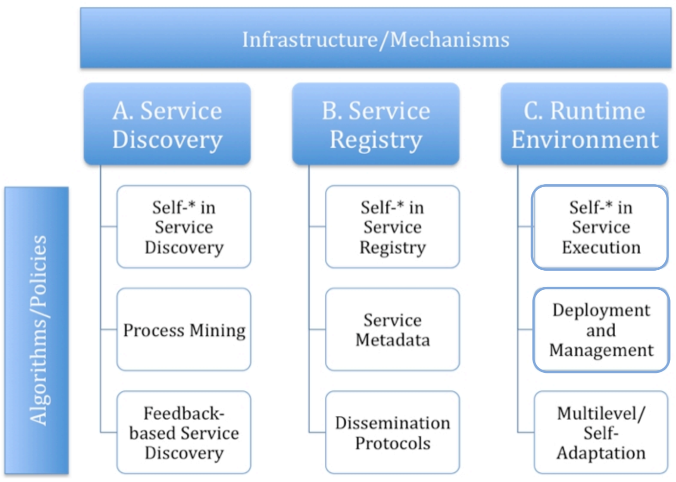
Infrastructure/Mechanisms
Algorithms/Policies
A. Service Discovery
Self-* in Service Discovery
Process Mining
Feedback-based Service Discovery
B. Service Registry
Self-* in Service Registry
Service Metadata
Dissemination Protocols
C. Runtime Environment
Self-* in Service Execution
Deployment and Management
Multilevel/Self-Adaptation

Figure 1: WP-JRA-2.3 Research Architecture

In Figure 1, we give an overview of the overall research architecture of WP-JRA-2.3: research on service infrastructures is structured in three threads, Service Discovery, Service Registries and Service Execution. To each of those categories, a number of concrete research topics (research directions) are denoted:

- Service Discovery - Service discovery is a fundamental element of service-oriented architectures. Other, and more complex, services heavily rely on it to enable the execution of service-based applications. Current discovery mechanisms are not well prepared to deal with the huge number of Internet-scale service ecosystems. Novel discovery mechanisms must be able to deal with millions (or even billions) of services. Additionally, these discovery mechanisms need to consider new constraints, which are not prevalent today, such as Quality of Experience requirements and expectations (feedback) of users, geographical constraints, pricing and contractual issues, or invocability (not every service can be invoked and used by every client).
- Service Registry Research Thread- service registries are tools for the implementation of loosely-coupled service-based systems. Current registries such as UDDI are not fault-tolerant. They are centrally managed entities, which do not scale well and do not offer support for rich service descriptions. With the advent of the Internet-scale service ecosystems a number of new challenges for the next generation of registries will arise. In such

systems, fault tolerance and scalability of registries is of eminent importance. Autonomic registries need to be able to form loose federations, which are able to work 24/7, in spite of heavy load or faults. Additionally, a richer set of metadata (data describing services) is needed for services in such ecosystems, in order to capture novel aspects such as self-adaptation, user feedback evaluation, or Internet-scale process discovery. Another research topic is the dissemination of metadata: the distributed and heterogeneous nature of these ecosystems asks for new dissemination methods between physically and logically disjoint registry entities, which work in spite of missing, untrusted, inconsistent and wrong metadata.

– Runtime Environment Research Thread- the obvious need for automatic, autonomic approaches have been addressed but either the scope is slightly different or the thoroughness, completeness of the solution does not fully meet the requirements of Internet-scale service ecosystems. Current adaptation methods focus on components and put little emphasis on services. Opposed to the self-* idea, they are merely targeting one or few reflective properties, typically some form of fault tolerance. Fault tolerance itself is usually a less complete solution than self-healing. Most approaches are targeted at short-term remedies, typically eliminating one problem. They are also acting locally (both spatial and temporal), not taking into consideration the overall state of the system or the chain of potential/possible causes and actions. There are various adaptation mechanisms for initial/bootstrapping configuration but there are few solutions that solve run-time dynamic adaptation by re-configuration. As opposed to current approaches we envision an infrastructure that is able to adapt autonomously and dynamically to changing conditions. Such adaptation should be supported by past experience (learning), should be able to take into consideration a complex set of conditions and their correlations, act proactively to avoid problems before they can occur and have a long lasting, stabilizing effect. All these three research topics possess a common feature that is characteristics of the entire research thread: policies for adaptation, and a knowledge base for adaptation strategies are defined.

1.2 Outline

Our main goal is to provide autonomic behavior for services, and to maintain the autonomic behavior of interoperable, cooperative services in a seamless and effective way. This deliverable involves establishing methods for problem identification, analysis of symptoms, repair and healing, specifying high-level policies and objectives and planning the execution according to the chosen adaptation policies. Here, we refer to an application as a composition of services. We consider three levels of adaptation. At the lowest level we consider the self-adaptation of one service, at the second level we consider adaptation inside an application to satisfy its specific needs and at the highest level we investigate the adaptation of several applications, running at the same time.The requirements for the first level of adaptation is described in this document.

There are certain cases when the adaptability, self-healing or other self-* properties of a service (or the service infrastructure) can be established in a coarsely controlled way by deploying and/or decommissioning services in an on demand, dynamic way. It is also strongly related to the service lifecycle and obviously, applies to the initial deployment of services. Deployment and runtime management (see Figure 1) deals with on-demand, dynamic provision of services, either due to temporal, spatial or semantic requirements. We envision a conceptual architecture that is able to provide such dynamic deployment and runtime management. Based on the concept, we separated three main functional phases. (i) There must be a *negotiation* phase when it is specified, what service is to be invoked and what are the conditions and constraints (temporal availability, reliability, performance, cost, etc.) of its use. (ii) Subsequently, in the *brokering* phase, an agent must select available resources that can be allocated for providing the services.

These resources can be provided in many ways: clouds (virtualized resources configured for a certain specification and service level guarantees), clusters or local grids (distributed computing power with limited service level guarantees) or volunteer computing resources (no service level guarantees at all). (iii) Finally, the service (or instances of a service) must be *deploy*ed on the selected resources in an automatic manner. In the scope of this report we analyze the requirements for decision making in (meta)negotiation, (meta)brokering and deployment layers of the architecture.

In this document we describe the challenges for self-* in service execution (C1) and deployment and management issues (C2) in more details. These challenges are highlighted in Figure 1. Applications of self-* in service registry and service discovery are studied in this WP but are not described specifically here. Multi-level self-adaptation (C3) will be studied in more details in CD-JRA-2.3.8. The other research threads A and B are described in CD-JRA-2.3.3. Note that this document purposefully excludes some aspects of services infrastructure, since they are out of scope of the WP: service composition is handled by WP JRA-2.2, and has therefore been excluded here; similarly, we do not consider implementation processes for service engineering, since this is the focus of WP JRA-1.1. Security as a crosscutting concern is not studied in this activity.

The remaining of the paper is organized as follows: Section 2 presents some definitions and Section 3 gives a typical use case of services in the ecosystem of services. This example is used along the deliverable and will be used during our research works. Section 4 presents local adaptation and self-healing requirements for service execution. Section 5 is devoted to present deployment and runtime management requirements. Section 6 summarizes these requirements as research challenges and future works.

2 Definitions

2.1 Services in the large

We consider the underlying architecture as being composed of *machines* capable of running one of more service. A machine has the ability to communicate with other machines in order to allow the service(s) to be invoked remotely and in order to be able to invoke remote services (we rely on a communication backbone but we don't make any precise assumptions on its capabilities). A machine can range from a handheld device such as a smartphone to a cluster in a large data or computing center, including personal computers and laptops. Machines are running an operating system and/or some middleware providing all the necessary core services needed such as communication facilities and resource brokering.

The infrastructure supporting this ecosystem is **distributed** meaning that system and middleware do not provide seamless data sharing, there exists delays when communicating between machines and there may be some failures among machines. The infrastructure will be considered on a **large-scale** basis (e.g. internet wide or even more including ad-hoc networks, etc.). This ecosystem of services is not only populated by services, business processes and other computer "stuff". -centered but may also includes **human** beings.

Figure 2 represents the different dimensions of this ecosystem. Within the distributed dimension, adaptation of services, including self-healing can be coordinated (e.g. services should be able to agree before any adaptation). In the large-scale dimension, services should be able to adapt themselves independently and locally. Self-healing is mainly for *non-human* dimensions, while in the Human-centered dimension, adaptation should be controlled (e.g. non decided by automated processes alone). The big challenge is to be able to take into account these three dimensions at a time.

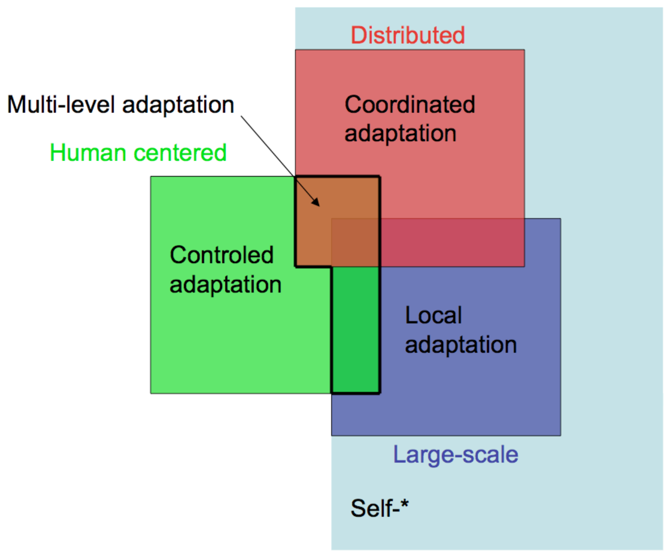

Figure 2: Domains of Adaptation

2.2 Local adaptation

In this section we present different definitions of locality. The goal is to have some clear definitions of "local adaptation".

Underlying infrastructure We call a *node*, a piece of hardware with one of more processor(s), memory, disk and network access. Nodes can range from simple PCs to high performance servers. A *cluster* is a small network of nodes. High performance networks are often used in clusters.

The process executing a service and its adaptation mechanisms have to be supported by the runtime environment. Any execution of a service within the architecture involved may be either supported on one or more node, inside one or more clusters. Figure 3 explains how local and distributed executions of one service can be performed on the runtime environment.

- *Local execution*: all the actions of a service are executed on the same node.
- *Distributed execution*: the actions of a service are executed at least on two distinct nodes belonging to the same cluster.
- *Inter-cluster distributed execution*: the actions of a service are executed at least on two distinct nodes belonging to different clusters.

In addition to the local and distributed executions of a service mentioned earlier, a service may be also classified with respect to its parallelism. A service is *sequential* if it is composed of only one process and one thread (such a service has only the ability to execute one instruction at a time). In contrast, a *parallel* service is composed of more than one thread or process. These concepts are complementary to the other ones related to the local and distributed executions.

A sequential service is bounded to use one node at a time (*local sequential execution*), for instance, it may use the node number 1 from the Cluster 1 (see Figure 3).

A parallel service have the ability to use more than one processor and may also have the ability to run on more than one node. For example, we have represented a parallel service running

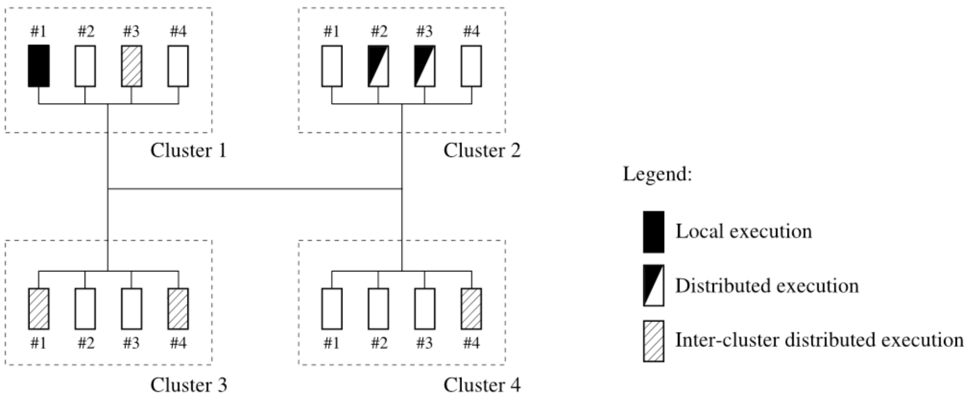

Figure 3: Local and distributed executions of one service

on the Cluster 2 and using the nodes numbers 2 and 3. This is a *distributed parallel execution*. Ultimately, a service could also use more than one cluster to execute its parallel code by using the nodes from Cluster 1, 3 and 4, thereby taking advantage of a inter-cluster distributed execution. This is the usual scenario for compute intensive services such as numerical simulation or a bioinformatics software when using resources distributed among different companies or research labs.

On the opposite, in case of a lack of resources, such a service has the ability to run on one node. This is a *local parallel* execution.

Services A service can be a composition of services or it can be a *atomic* service (i.e., a service that does not call any other service). In this document we consider both kinds of services. Thereby, such a single service performs a *local adaptation* when its adaptation has no coordination with other services. This does not mean that the local adaptation of one service may not affect other services. Even though it may perfectly occur, we do not consider the coordination of the adaptation process in this document. Furthermore, it is important to remark that the meaning of *local* here is different from the previous definition related to the underlying architecture. In this document we refer to local adaptation with the service point of view.

2.3 Self-healing

With the Internet rapidly growing in the nineties and the first "computer viruses" becoming a threat, researchers such as S. Forrest, A. Somyaji, and S.A. Hofmeyr, inspired by the capabilities of Mother Nature's immune system, began to explore similarities between the defense mechanisms of organisms and computer systems [9]. As technology advanced over the years, from these early attempts to copy natures immune system features for "self-intrusion-detection", the spectrum eventually broadened to the research field of self-*. Self-healing part of self-* is one of the sub-fields still closely related to the immune system analogy.

A comprehensible and contemporary definition of self-healing can be found in [11]:"the key focus or contrasting idea as compared to dependable systems is that a self-healing system should recover from the abnormal (or "unhealthy") state and return to the normative ("healthy") state, and function as it was prior to disruption." A self-healing system shall tend to become sound with infrequent faults, though, accept temporarily malfunctions. More formal, the mean time between faults (MTBF) shall be much larger than the mean time to repair/recover (MTTR). Overall, a "fit" system is desired running for a maximum possible term. Apart from the two obvious states "healthy" and "unhealthy" transitions to a third state "degraded", modeled after

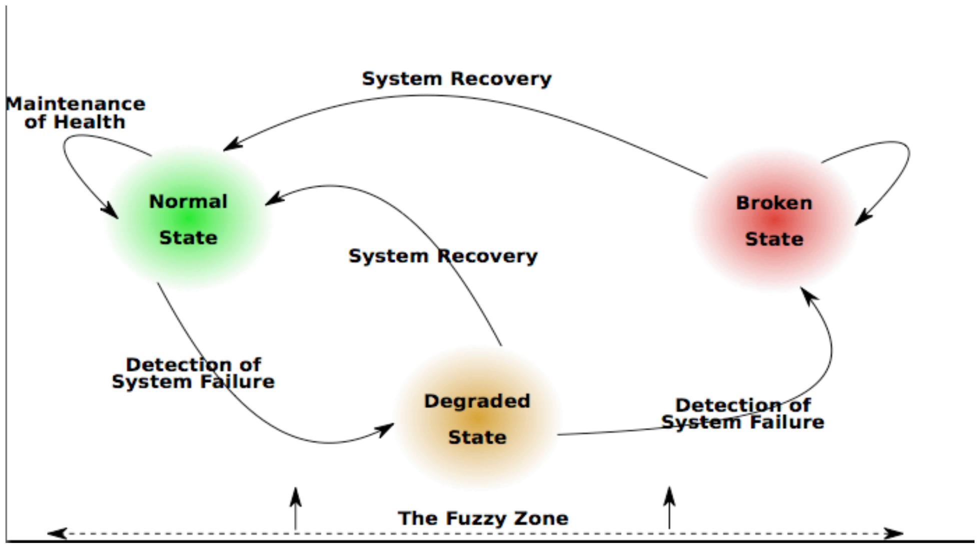

Figure 4: State diagram of self-healing

natures latent period, are introduced referring to a system on the verge of "unhealthy", however, hidden to the participants. This is also known as the "The Fuzzy Zone".

Figure 4 represents the previously described self-healing state diagram including the transitions labeled with the accompanied actions. The challenges of introducing self-healing into a system newly developed or already existing are twofold. On the one hand, the state of "health" or more precisely what presents the boundaries of "healthy" in the system must be defined and agreed. The challenges here are not only a reliable monitoring and correct interpretation of sensed data but also the possible variation and discrepancy of the "health" definition in changing and/or evolving systems and the opinion on that matter by the different stakeholders. On the other hand, recovery complexity increases with system size. Therefore, approaches are various and include off-line related "healing" by allowing a possibly longer "degraded" state while estimating the appropriate recovery strategies with time consuming methods of artificial intelligence [5] but also more on-line methods with short or no "degraded" state by cutting off the "infected" parts of the system during repair and designate a temporary substitute [10].

Following an example of self-healing in the context of the NavInc. case study, we refer to the implementation of the service running on the GPS car unit as HealthyNavInc. The task of HealthyNavInc is to monitor the fuel-status of the corresponding car in comparison to the distance of the next gas station(s) on the current computed route. The three states are easily identified:

1. "healthy": The fuel level in the car is sufficient to reach the closest known patrol station.
2. "unhealthy": The car has stopped because of the lack of fuel.
3. "degraded": The systems estimations indicate that the fuel level is to low to reach the next known patrol station, but the car is sill running.

Obviously, we assume that the starting state is "healthy" with the system in total "health". Further, assuming a perfectly running system transitions to the undesirable states are unlikely. However, let us assume the dominating stakeholder, namely the driver, despite of Healthy-NavInc's warnings, decides to take an unexpected route. After revising the decision Healthy-NavInv could find itself quite instantaneous in the "degraded" state and tries by any means to return to "healthy" state and back to a safe route where a gas station is reachable. This could be as simple as a “give-in” by the HealthyNavInv and requesting the driver’s intervention or estimate alternatives by any information sources available. This could also include just established ad-hoc information services and in the best-case lead to a newly detected gas station in range.

The requirements of self-healing in S-Cube offer a great chance for collaboration with other WPs. As aforementioned, at the beginning we must differentiate as precise as possible the ranges between “healthy” and “unhealthy”. Here the opinions of all stakeholders found in the various layer of S-Cube must be evaluated and respected most evident, i.e., *JRA-1.3 End-to-End Quality Provision and SLA Conformance* with their SLAs and *JRA-2.1 Business Process Management* with their KPIs. Wherein the infrastructure itself must be considered as an important stakeholder. Its capabilities to monitor, adapt and evolve limit the desires of the upper layers. In this matter, a strong collaboration with *JRA-1.2 Adaptation and monitoring* is required and an integration of the results of their research must be considered. Regarding the applied strategies for self-healing we might consider an adoption of both of the previously described on- and off-line integration as we consider local and multilevel adaptation.

Self-healing can apply to one service or to a set of services. In the case of one service, self-healing is a special case of local adaptation.

3 Illustrative example

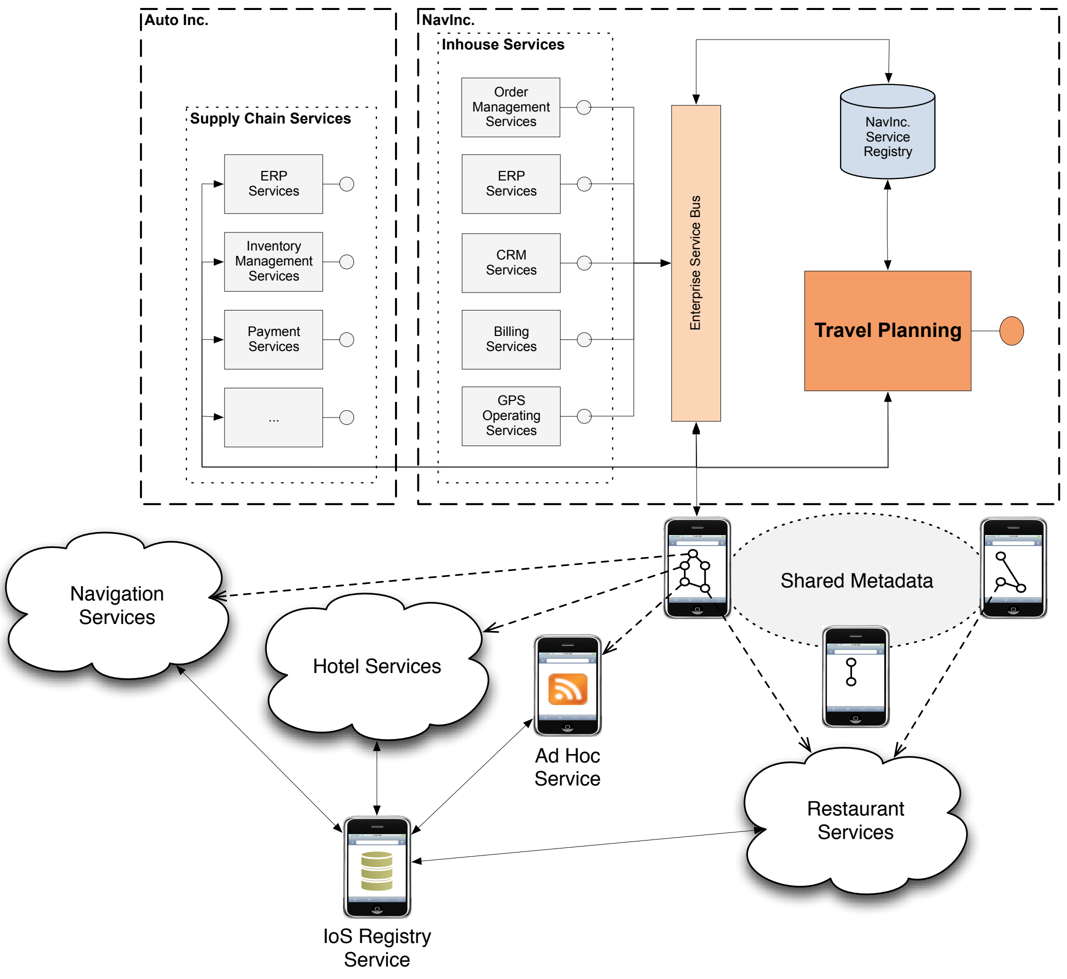

Figure 5: Illustrative example

We consider a fictional producer of car GPS units as described in deliverable CD-JRA 2.3.3. The whole case study is also available in the CD-IA-3.1.1 deliverable.

In addition to the services presented in this deliverable, we consider navigation services such as safety warnings, real-time traffic and road condition information, weather reports, which are now standard features of GPS units. We also consider that ad hoc services may appear and disappear both for safety or information purpose.

In the scenario, we consider that the GPS car unit allows the computation of the route and regular calls to some base station supporting access to standard information services. In case of any problem or accident reported by these services, the GPS unit displays a message and may recompute a new route. In case of a geographically close event, some ad hoc services hosted by maintenance vehicles, by the police teams or by other drivers may appear and the GPS unit may connect to these new services in order to get more precise information. Because most of the time traffic conditions are not that bad, we think of other ad hoc services such as services to help people who travel together to be located on the same map and to share common services (such as restaurant proposals). Advertising services may also appear when approaching some hotel, restaurants or gas stations for example.

Self-healing requirements We consider the "basic" behaviour of the GPS unit as the following: the route is already computed and displayed, the GPS unit periodically connects to the nearest base-station to get local information and reports a "green light" if traffic conditions are normal. We may think of three or more "modes" for the GPS unit related to traffic conditions such as : "normal", "alert", "major hazard".

In case of an alert concerning safety, the GPS have to *self-adapt* its behaviour : for example, it could try to find other sources of information given by ad hoc services. These ad hoc services are registered by the IoS registry services (see deliverable CD-JRA-2.3.3). The GPS unit then can collect, use and possibly resend the information provided.

In case of poor connectivity (or even a total failure) with the base-station preventing to contact navigation services, we may think of building a P2P navigation service with other drivers. This is another example of *self-adaptation* at the level of the GPS car unit.

At the level of information services, there is also a need for *self-adaptation* because ad hoc services provided by drivers may also be useful to compute information that could be broadcasted.

In addition, some new maintenance or security vehicles (also running ad hoc services) may be sent to the place where there is an accident and join the P2P community.

In case of an accident, there is a strong need of *self-healing* for information services because the volume of information exchanged may increase, so more resources may be needed. Moreover in case of a major crisis, probably some information services will no longer be available (such as hotel or tourist information) and the behaviour of the traffic information services may be different than in normal traffic conditions.

Another example is the GPS car unit that may contain services such as delivering the coordinates of the car to authorized people. In case of a good connectivity they can use both satellites and network base-station to compute localization. If there is a poor connectivity only satellites should be used. This is another kind of self-healing need.

Deployment and run-time management requirements We also consider cases where a service should be deployed dynamically, on-demand, from an image stored in the registry (or at a location indicated by the registry) to enable a navigation (or other computationally intensive) service on the cloud. The basic assumption is that there are certain services that do not need to or cannot be provided in a static, permanent way. Instead, these services should be created and decommissioned in an adaptive, dynamic, on-demand way for temporal, spatial or semantic reasons.

For example, there is no need to dedicate certain resources to certain services as they occur rarely or at least infrequently (temporal availability).

In some other cases resources cannot be assigned to certain services as they have to be spatially available. For instance, if the surrounding service providers do not offer a navigation service, but a user needs a more precise solution than its GPS can offer, the service may be deployed for just the timeframe the user is on site. When the user leaves, this navigation service is decommissioned.

It is also possible that certain services are needed upon a certain event (semantic availability). For instance, an unforeseen event in the travel scenario, like an accident may fundamentally change the requirements posed to certain services.

Sometimes, it is unknown how many instances or what version of a certain service is required. For instance, in case of a congestion, overload on a certain service, new instances may be deployed or alternatively, new resources may be assigned to the existing services.

As some ad hoc services may be mobile, they cannot be assumed constant. In certain cases dynamic re-deployment of a service not available locally may be necessary.

A more complex case involves spatial and temporal emergence for services as well as a strong need for adaptation. The user offers the video stream of the volcano eruption (like in the example in section 5) then service providers will deploy on-demand a streaming service close to the user's location. This streaming service then will take over the burden of broadcasting the streamed content to the viewers of the eruption by acting as a huge bandwidth relay for the viewers. It is possible to adapt the eruption's movie to the different viewing devices: it is not just the bandwidth but also the computing power required to transform the user's video stream to various formats on the fly. When the stream is ended the broadcasting service would be decommissioned.

4 Requirements for Adaptation and self-* in Service Execution

This section refers mainly to Thread C1 of the WP 2.3 research architecture but may also apply to A1 and B1.

4.1 Requirements for a local adaptation and self-healing infrastructure

In order to help developers and designers of adaptable services, we think that a general model is needed to describe and to implement adaptation mechanisms.

An adaptation framework should follow the functional decomposition of an autonomic manager suggested in [14], dividing the adaptation in *monitoring*, *analyzing*, *planning* and *executing* parts. Each part triggering the next one: the monitoring gather contextual information used by the analyzing part to decide whether an adaptation is needed or not; from this need, the planning part builds an execution plan to be executed by the executing part. Figure 6 shows how those parts could be organized.

The contextual information can be gathered by probes through events and measures. Events can trigger an adaptation while measures are done on demand by the analyzing part when complementary information is needed. The monitoring is not only platform specific, it can also be application or domain specific when adaptation is not due to resources. The application itself can be monitored, for self-healing purpose for example, by a machine learning software, the user or by using ad-hoc metrics.

The analyzing part is done by the *decider*. When receiving an event, the decider chooses if an adaptation is needed by following a specific decision policy. This policy can be expressed by

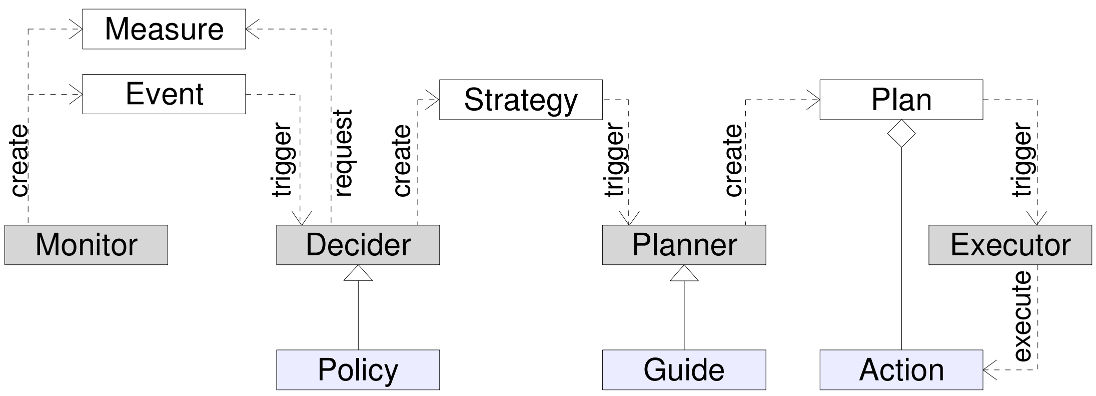

Figure 6: Structural decomposition of the general adaptation framework.

"event-condition-action" rules or goal-based functions for example, depending on the decider's implementation. This structure enables to choose the decider the best suited to each decision problem without imposing one way to write every algorithm. Furthermore, this structure enables to use the same decider for different services; only the policy is specific to each service adaptation.

Once an adaptation is chosen, the *decider* sends a *strategy* to the planning part, implemented by the *planner*. The planner has to work out how to apply the strategy to the service to adapt. Which means that the decider has to decompose the received strategy into elementary tasks to be executed. In order to better know the current state of the service to adapt, the decider can request contextual measures to the monitoring part. In the same way and for the same reasons that the decider follows a policy, the planner follows a *guide*. For example, the guide can be an oriented graph representing every possible configurations connected by the actions to perform to go from one configuration to another. In this case, the planner would use the graph to go from the running configuration to the one described by the strategy in order to build the plan.

Then the action plan is sent to the executing part, which is the *executor*. Its role is to execute each action specified in the plan, taking into account the execution of the service to adapt. To do so, the executor may intercept the execution flow to execute adaptation actions. This part of the framework is discussed later in the document as *concrete actions* in section 4.6.

This description can cover both self-adaptation and controlled adaptation, depending who is responsible for the policy. In order to cover pro-active adaptation (for example for self-healing purpose), one should provide monitors or interfaces to monitors able to generate events when a deviation of the QoS or in the use of resource that may lead to a failure or a SLA violation is detected. The evaluation of the opportunity to adapt or not as well as the evaluation of the consequences of an adaptation prior to its execution can be included in the plan through the definition of a specific guide. It is clear that some refinement of this framework may be useful and should be done in cooperation with WP 1.2 works (see deliverable CD-JRA-1.2.2).

4.2 Requirements for Monitoring

Designing a service capable of adaptation consist in correctly providing it with monitoring services or interfaces to monitoring services, adaptation policies and support for concrete actions. Monitor services provide events relative to the service's context, for the adaptation system to base its decisions upon them. Monitoring has to provide information related to health of services. In order to know if a service behaves properly, 3 different things have to be monitored :

– is the usage of resources compatible with the allocated resources ? If not, some resources have to be reallocated or the instance of the service have to be replaced. This needs to know the actual use of resources of a service, the limits of the resource usage and to be able to allocate dynamically new resources to a service or to replace an instance of a service.

– does the actual QoS of the service conform to the SLA ? Can we predict it will continue to conform to SLA for further requests ? This needs to have a run-time representation of SLAs allowing to know what should be monitored or to have the monitoring provided together with an adaptation strategy.

– even if services are stateless from the client point of view, during execution of one call, services have transient states (they are implemented as programs anyway). Is this internal state healthy ? degraded ? To know this is very difficult because there is both the need for a model of the service behavior and some between the model and the internal state of the running service.

4.3 Requirements for decision support

In this section we describe the types of functionalities needed to be able to implement the decision in the adaptation process at the infrastructure level. The decision support takes input from monitoring mechanisms and produces a strategy (see section 4.4) as a result of a decision.

There are many ways to implement decision mechanisms depending on the complexity and the accuracy of the decision to make and on the response time of the system one wants to design. The simplest solution is to implement a simple Event-Condition-Action rule based system. Each event triggers one (or more) rules if and only if some condition holds. This is a very simple and efficient mechanism that is able to take into account the internal state of the system if needed. The main drawback is that such a mechanism is only able to react properly to known events and conditions. Moreover, if the system takes a bad decision once, it will take it at each time it reaches the same configuration. This decision mechanism should be used only for very simple and well known cases. Its main advantage is that it can be very lightweight. We have used successfully such a decision system for the proactive adaptation of high performance parallel programs to resource availability in [2].

Such a decision system can be more powerful if it can take into account patterns on events. A survey on fault diagnosis using patterns for electronic systems can be found in [7].

If one needs to evaluate the consequences of an adaptation before triggering it (for pro-active adaptation purpose), such a simple system is not enough.

Table 1 summarize this discussion. In order not to take always the wrong solution, it may be necessary to have a memory of the previous decisions and results. In this case, a nice implementation of a decision system could be done using neural networks [13]. Decision support systems are a wide area of research. One can find a comprehensive bibliography on this topic in [17].

4.4 Requirements for definitions of strategies

Monitoring provides information related to health. When monitoring reports degraded or broken state there is a need to find a strategy for adaptation. If the broken state is not reached, pro-active adaptation is still possible. The following strategies, as the result of the decision process have to be studied : adapt, renegotiate or fail.

Adapt Different characteristics of services can be adapted. One of the simplest characteristics to adapt are parameters as they are built to have their value changed during the services run time and because they only need an write access through an interface.

The service's internal behavior is another adaptable characteristic, representing how the service achieves its purpose. So usually, changing a behavior means changing an algorithm.

Table 1: Decision making tools

strategy based on		consequences	decision system
1 event	no state	predefined	simple ECA (Event Condition Action) rule based system
1 event	current state	predefined	simple ECA (Event Condition Action) rule based system
pattern on events	current +past states	memory	IA based decision system
*	*	evaluation	pro-active adaptation

Another characteristic that can be adapted is a service's set of interfaces. Indeed, depending on its environment, a service might enable or disable some of its functionalities that are reflected on its interfaces. For example, a service might want to discard to provide secured communications while accessed through a virtual private network (VPN).

However, sometimes some behaviors cannot be known at design time, for example because a human decision is involved in the conception of the behaviors at run time. In those cases, a parameter cannot be used to choose a behavior. Therefore, the adaptation mechanism have to be able to bind a sub-service implementing the required behavior to the main service.

Renegotiate If the service is unable to adapt its behavior or its parameters to maintain the required QoS, it should renegotiate the (terms of the) QoS. This is studied in WP 1.2.

The service may also renegotiate with the underlying system in order to claim more accurate resources, for example, in case a service needs more memory or more processors. The claim of memory has been done in a virtualized system supporting a tomcat server [1].

Fail In case no adaptation strategy can be found, no resource claiming is possible nor able to solve the problem and no negotiation is possible, the service could decide to fail living the underlying system in a consistent state instead of waiting to crash. In this case the service gets one chance to report his failure together with a diagnosis. This may help the designer to define adaptation strategies avoiding to use this service in these conditions.

There may be nothing to report or no time to do it, so even if the service fails silently, living the underlying system in a clean state, it is better than waiting for a crash.

4.5 Requirements for planning

A plan should be defined as a list of actions to execute for implementing the adaptation strategy. Several actions often have to be performed, for example load a new implementation. Some of these actions may be performed simultaneously (in parallel), but some have to be synchronized. Finding a good schedule of these actions may be a difficult task. For each strategy it may be possible to have predefined plans or one may rely on planning algorithms for finding the best schedule of actions to execute. It may be possible to decompose a plan into elementary phases. Some elementary phases may be reused to implement several plans; so it could be useful to provide a repository of such elementary plans. For example, if it has been decided to migrate the service from one node to another, the service have to be stopped, pending requests have to be stored; then the service code have to be copied on the new node, and then restarted.

4.6 Requirement for the support of concrete actions

In this deliverable we describe mechanisms at a conceptual level, so we are as much as possible technology independent. If we want to implement concrete adaptation, we will have to know how the service is implemented and what is the underlying infrastructure.

support for behavior adaptation If there exists multiple instances of different version of one service, an adaptation can be to change from one version to another. If there is no deployed instance of the new version, it could be deployed at that time. The versions of a service can be programmed in different ways. For example, if an instance of a service seems to get out of memory after some requests, it could be replaced by another one after some calls. For a composite service S using S1; if we have some diagnosis on invocations of S1 telling us that S1 seems to be not healthy or to have already failed, a self-healing adaptation of S could be to replace the running instance of S by another one that does not use S1. In this case, there is a need to have access to the description of composite services.

Another way to change a service's behavior is to use *sub-services* for each different behavior and to switch between them when appropriate. This makes a good use of the dynamism offered by SOA. For some services, the full list of possible behaviors can be known at design time (i.e. statically), while for other this list can only be known at run time (i.e. dynamically). This leads to two possible implementations of ways to switch between services. When all behaviors are known at design time, the pattern state can be used to select the sub-service to use. Then a behavior adaptation consist only in modifying a parameter. How to modify the parameter then depends on the mechanisms selected to perform parametric adaptations.

If one has access to different *implementations* of a service, many adaptations can be achieved inside the service container by changing the active implementation. For example, security checks can be turned on/off depending on the environment. This needs to have for one service a list of implementations within a description of their properties.

In case the service provider wants to add some self-* properties to a service, he has access to the implementation of the service (the code). In this case it is possible to prepare the changes of the internal behavior of the service. Adaptations can be entirely pre-programmed or only prepared and then inserted at run-time using Aspect Oriented Programming Techniques [15]. In some languages, built-in support for reflection is a useful tool to use [8]. However, it is always desirable to separate the business logic from the adaptation code.

support for parameter adaptation This is the easiest way for adapting a service, because the possible values that a parameter should have been taken into account at the design time of the service. But this type of adaptation is in general limited to very simple changes in the behavior of a service such as: response time or frame-rate adaptation, change of coding format, etc.

support for interface adaptation Dynamically changing interfaces is a way to support adaptation but is not possible in all technologies. When this is not possible, it is still possible to provide this functionality by using services as proxies for each interface. The different interfaces of the service would by accessible by other services only through those proxies-services. Then, the adaptation mechanism can stop and start the proxies when a need arise.

4.7 Requirement for Integration level

Adaptations policies are rules to follow or objectives to meet. They state when a adaptation is needed and how to adapt. Concrete actions are methods to use in order to modify the service, like accessors (setters) to its parameters or bindings. The adaptation system has to provide a clear interface to request those means of adaptation. This interface will have to be defined.

On one hand, to take advantage of the potential of SOA, the adaptation system can be considered as a service. This means that a service needing adaptation would have to be designed to require an adaptation service. Ideally, the service to adapt would be provided with the adaptation service since it would be necessary, however another adaptation service could be chosen at run-time. On the other hand, to take advantage of a close integration with service implementations, the adaptation system should be implemented inside service containers or even inside service implementations. These two points of view are conflicting and different solutions should be studied depending on the balance between deeply embedded adaptation and reusable adaptation.

4.8 Example of integrating self-adaptation in a master/worker framework

In order to present what might be needed to design an adaptation algorithm, we present here the design process of an adaptation algorithm by an example coming from component oriented computing. Despite this example deals with components instead of services, it is still relevant since the design process of an adaptation algorithm is similar in both programming paradigms.

The application to adapt is an implementation of the master-worker paradigm with components and is intended to be deployed on grids. Figure 7 depicts the components' organization inside the framework. This framework is to be used by developer to build distributed compute-intensive applications without worrying about details of the grid infrastructure or its dynamicity. Since grids are used by multiple users, their performance and availability characteristics can vary largely during an application life-time. In order to use the framework, two components are to be developed: the master component and the worker component. The framework distribute tasks sent by an instance of the master component to multiple instances of worker components. It use a simple algorithm to choose how many worker to use and which tasks distribution pattern to use. When characteristics of the application (e.g. the ratio of tasks to be processed over the number of workers) or the grid (e.g. resources usage) change, the framework has to decide to change the number of workers or the tasks distribution pattern (a.k.a. the master-worker pattern).

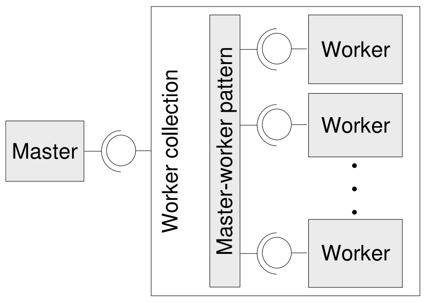

Figure 7: The master-worker paradigm, in software components

We have used the Dynaco framework based on the concepts described in 4.1 and described in [3] to implement the adaptation of the master/worker. We present here how the adaptation algorithm (deciding when to change the master-worker pattern) has been designed.

Characterization We followed a down-top approach to design the adaptation algorithm, keeping in mind the goal of the adaptation. So the first step in designing the adaptation algorithm was to characterize the system to adapt.

The first thing we did when designing the algorithm was to state the different aims for the adaptation—*why* do we adapt the component. We studied three QoS objectives:

– Maximize the throughput of the application. It is measured by the average number of executed requests by second. We assume that every pattern ensure that every request sent by the master is processed (i.e. there is no starvation).

– Minimize the processing time of any request, independently of the other requests. This objective should not be confused with the preceding one. This objective enables to consider the case where only some of the requests' processing time are to be minimized.

– Respect a maximum time limit to process a request. This objective enables to design "soft real-time" applications. It is the user responsibility to specify reasonable time limits, i.e. that can be met, otherwise the satisfaction of this constraint cannot be ensured.

Then we selected the different behaviors for the collection, implemented in components by task distribution patterns.

Many master-worker pattern exist and could be integrated in the framework by embedding them into a component. We selected three of them: the simple and fast round-robin pattern, the load-balancing pattern and the DIET [4] pattern with a modified scheduler.

The round-robin pattern distributes the number of *request* fairly among the workers, while the load-balancing pattern distributes the *workload* fairly among the workers. The DIET pattern uses a request sequencing policy, it uses a distributed architecture with many agents and has probes to its disposal to estimate the workers' processing speed. We use a modified version of DIET, which differs from the original by a scheduler that sorts the requests by the estimated time to process the requests, when possible.

In order to be able to decide to adapt, the application needs to monitor itself and the distributed system. To this end, we have to select relevant parameters for the adaptation. We consider a potential parameter as useful if its modification can generate a need to adapt, or if it can be used to describe the state of the application's part to adapt (here, the number of workers and the master-worker pattern). Then, we decompose the parameters into elementary events to monitor. So the first step of the algorithm is to build the parameters from the monitored events.

The decision algorithm Upon the characterization of the system to adapt, we were able to build the algorithm.

The decision algorithm is based on the description of the behavior of the patterns, which depends on the state of the pattern and the distributed system, and on a QoS objective. It is designed to be generic: new patterns can be added to the application without modifying the existing parts of the algorithm implementation. To this end, the behavior description of a pattern is independent from those of the other patterns.

The algorithm is a compromise between performance and an ideal solution: it does not aim to select the pattern the best fitted to every situation. Instead, it aims to discriminate a pattern between the best fitted in a short time. Moreover, it is not always useful to select the optimal pattern among two almost equivalent ones, as long as the inadequate ones are filtered.

The principle of the algorithm is to compare the patterns using a description of their behavior. This comparison is done using positives scores: the pattern with the lowest score is the best fitted to the situation for the given QoS.

The behaviors of the patterns are divided into elementary behaviors for each of which a *cost function* is defined. A cost function represent the extra-cost induced by a behavior, with regard to a QoS objective. Each of these functions take as input a *characteristic* built from monitored parameters.

These functions can be discovered using simulations, by monitoring the behavior of the pattern in controlled environment or by knowing how the pattern behaves. This can be a *long* process, depending on the complexity of the behaviors.

The identified characteristics are:

– the number of workers;

– the number of requests being processed;

– the number of requests being processed divided by the number of workers;

– the variability of the workload of the workers due to the environment exterior to the workers;

– the variability of the processing power of the workers;

– the variability of the time to execute requests;

– the heterogeneity of the network;

– the time to execute requests added to the time to send them across the network;

– all of the parameters.

For each of these nine characteristics, a cost function compute an index, which is a positive number (within $\overline{\mathbb{R}}^+$) estimating the extra cost resulting of the behavior described by this function. It is without unit, since the estimation depends of the QoS objective. A value of 0 means there is no extra cost for this behavior. A value greater than 1 means that the extra cost is significant. The significance threshold is arbitrary set to 5%. The value $+\infty$ means the impossibility for the pattern to respect the QoS objective.

With each of the first seven characteristics an index is associated that represents the extra cost resulting of the characteristic. Two special indices are associated with the last two characteristics: the *relative extra cost by request* and the *extra cost specific to the pattern.*

The index of the *relative extra cost by request* is calculated using an *extra cost function* given by each pattern to get the extra cost by request (in seconds) which is then divided by the time to process the request (in seconds). So it uses all of the parameters as a characteristic, since the function may need any of those. The *extra cost specific to the pattern* is used to take into account all the specificities of a pattern that cannot be expressed by the other indices.

Those indices are then weighted by two classes of positives coefficients. General coefficients are common to all of the patterns and are used to weight the indices between them. Specific coefficients are specific to each pattern; they are used to adjust the indices within the patterns and to adjust a pattern against the others. The coefficients can be calibrated by hand, or modified by a learning algorithm, by measuring the results of successive adaptations.

Once the indices are weighted, they are added to make a score by pattern. Then the scores are compared; the pattern with the lowest score is selected. If the difference between this score and the score of the pattern currently used is greater than 5% of the latter (that is, if the difference between the scores is significant), then an adaptation is triggered. If all the scores are infinite, there is no lowest score, so there is no pattern selected and no adaptation triggered. In case of conflict between patterns, one is to be arbitrarily selected, the last used might be a good choice to avoid the overhead of deploying a new pattern.

To validate the decision algorithm, two simulators were developed: one to simulate master-worker patterns on distributed systems, the other to take decisions according to the algorithm. The first one is used to simulate the behavior of the various schedulers of the patterns and characteristics of the distributed systems (like the computers workload). The second one offers to simulate the evolution in the time of the parameters used in the decision algorithm, in order to visualize adaptation choices done by the algorithm.

5 Requirements for self-healing and decision support in deployment and runtime management

5.1 Introduction

Deployment and runtime management is aimed at providing or modifying services in a dynamic way according to temporal, spatial or semantic requirements. Among many other puposes, it has also strong relation with adaptation and self-healing. The material presented in this section is related to planned research work in research topic C2 in Figure 1. We take into consideration two motivating examples. The first one is from the general scenario (Section 3), the other is an additional, more specific one.

Scenario 1 In the NavInc. scenario [6] there are services that do not need to or cannot be provided in a static, permanent way. Instead, these services should be created and decommissioned in an adaptive, dynamic, on-demand way for the following reasons:

– There is no need to dedicate certain resources to these activities as they occur rarely or at least infrequently.

– It is possible that certain services are needed upon a certain event.

– It is unknown how many instances or what version of a certain service is required.

– As some ad hoc services may be mobile, they cannot be assumed constant.

– In certain cases dynamic re-deployment of a service not available locally may be necessary.

Scenario 2 In the CarInc. scenario [6] there are certain procedures in the Production and Test activity that may require specific services and resources in an ad hoc, temporal way. Definitely, there is no reason to provide these services in a static 24/7 manner with performance guarantees, instead, these services should be created and decommissioned in a dynamic, on-demand way for the following reasons:

– These tasks represent fundamentally different computations that cannot be re-used or composed, potentially not even overlapped, e.g. air tunnel simulation, crash test analysis, various optimization procedures, and so on. These services must be provided independently form each other in a well defined and disjoint time frame.

– There is no need to dedicate certain resources to these activities as they occur rarely or at least infrequently. Resources used by the optimization services, for instance, can be re-used for other purposes when no optimization service is required.

The aim of this section is to investigate the requirements of the self-healing and decision support aspects of runtime management. The requirement analysis is facilitated by a conceptual architecture (Figure 8). Temporal provision of services requires certain infrastructure features that we classified into three groups. There must be a negotiation phase when it is specified, what service is to be invoked and what are the conditions and constraints (temporal availability, reliability, performance, cost, etc.) of its use. Subsequently, an agent must select available resources that can be allocated for providing the services. These resources can be provided in many ways: clouds (virtualized resources configured for a certain specification and service level guarantees), clusters or local grids (distributed computing power with limited service level guarantees) or volunteer computing resources (no service level guarantees at all). Finally, the service must be deployed on the selected resources in an automatic manner.

Our aim is to investigate (meta)negotiation, (meta)brokering and (auto)deployment issues. The purpose of the conceptual architecture is to provide an integrated framework to investigate the three areas in a unified way. Albeit, the architecture could be implemented this way, other realizations are also possible. In this sense, the architecture is not a concrete system design rather a conceptual framework.

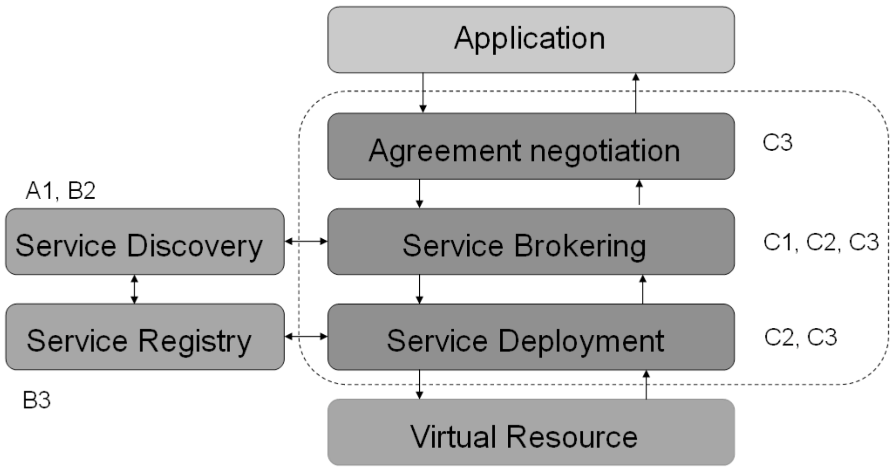

Figure 8: The conceptual architecture for deployment and runtime management showing the relations to research topics

In the followings, requirement analysis is presented for negotiation, brokering and deployment with respect to self-healing and decision support features.

5.2 Requirements for Agreement Negotiation

In the early days of Grids users had to commit themselves to dedicated Grid portals to find appropriate services. Thus, service providers and consumers had to communicate using proprietary negotiation formats supported by the particular portal limiting the number of services a consumer may negotiate with. In present day Grids/Clouds service provider and consumer meet each other dynamically and on demand. Thus novel negotiation strategies and formats are necessary supporting the communication dynamics of the present day Grids/Clouds. In this section we address the second point of the Scenario 1 where certain services are needed upon a certain event. If for example, an unforeseen event in the travel scenario, like an accident may fundamentally change the requirements posed to certain services. Thus, we assume that we have to find services from publicly available registries and to negotiate with those services even if we do not know about the supported negotiation protocols, document specification languages and similar.

Before committing themselves to an SLA, the user and the provider may enter into negotiations that determine the definition and measurement of user QoS parameters, and the rewards and penalties for meeting and violating them respectively. The term negotiation strategy represents the logic used by a partner to decide which provider or consumer satisfies his needs best. A negotiation protocol represents the exchange of messages during the negotiation process. Recently, many researchers have proposed different protocols and strategies for SLA negotiation in Grids. However, these not only assume that the parties to the negotiation understand a common protocol but also assume that they share a common perception about the goods or services under negotiation. In reality however, a participant may prefer to negotiate using certain protocols for whom it has developed better strategies, over others. Also, a participant may choose to only allow certain aspects of a good or a service to be negotiated which may not be acceptable to others. In other words, the parties to a negotiation may not share the same understanding that is assumed by the earlier publications in this space.

In order to bridge the gap between different negotiation strategies we need a mechanism we refer to as meta-negotiations. The aim of a meta-negotiation is to find appropriate services a service consumer may negotiate with. The following aspects of meta-negotiation should be considered / solved:

- Develop a scenario where service provider and consumer may exchange their prerequisites for the negotiation. Based on the scenario service consumer and provider should meet each other, exchange the documents, start and conclude meta-negotiation. The outcome of the scenario should be a bunch of candidate services to which service consumer may start negotiation.

- Meta-negotiation should by represented by means of a meta-negotiation document where participating parties may express:

  - the pre-requisites to be satisfied for a negotiation, for example a specific authentication method required or terms they want to negotiate on (e.g. time, price, reliability);
  - the negotiation protocols and document languages for the specification of SLAs, e.g. Web Service Level Agreement (WSLA) or WS-Agreement that they support;
  - and conditions for the establishment of an agreement, for example, a required third-party arbitrator. The meta-negotiation document should be specified using standard technologies and languages.

- Development of the appropriate meta-negotiation middleware consisting of following components:

  - Registry: The registry should be a searchable repository, like a service registry, for meta-negotiation documents that are created by the participants. The participants should submit their documents and request matchig services which provide and understand specific negotiation protocols, or specification languages. We should consider development of the distributed and federated registries in order to avoid performance bottleneck. Registries requirements can be found in deliverable CD-JRA-2.3.3. Also the concepts like Cloud computing could help to deal wiht performance bottlenecks.
  - Meta-negotiation middleware. The meta-negotiation middleware should facilitate the publishing of the meta-negotiation documents into the registry and the integration of the meta negotiation framework into the existing client and/or service infrastructure, including, for example, negotiation or security clients. Besides being as a client for publishing and querying meta-negotiation documents the middleware should deliver necessary information for the existing negotiation clients, as for example information for the establishment of the negotiation sessions and information necessary to start a negotiation. We assume that each service consumer may negotiate with multiple service providers concurrently. Also even the reverse may happen as well, wherein a consumer advertises a job. In such cases, the providers would negotiate with multiple consumers.

Managing service executions, the following means of negotiation need to be provided:

- User - Negotiator: user supplies input for a meta-negotiation document.

- Negotiator - Service brokering: they need to agree on a specific negotiation document using a specific negotiation strategy and negotiation protocols to lower components.

– In Service brokering: the agreed negotiation document needs to be filled by parameters like concrete execution time, concrete price, etc. (i.e. the negotiation terms).

– Service brokering - Deployment: they should agree on a specific service to be available on the managed resources with the resource constraints resulted from the higher level negotiations. During this phase the deployment component need to share information on resource availability and deployment costs.

5.3 Requirements for Service Brokering

In this subsection we are focusing on the Service Brokering component of Figure 8. Brokers are the basic components that are responsible for finding and selecting the required services. This task involves various activities, such as service discovery and selection, and interactions with deployment, information systems, service registries and repositories. Looking at the general architecture in Figure 8, we can see that this level is in connection with Agreement Negotiation to upper level and Service Deployment to lower level. Subsection 5.2 described the agreement negotiation requirements and steps. Service brokering should participate in the negotiation process by providing information on dynamic brokering, deployment and execution capabilities. After the agreement is made, a service broker should be selected for the service request that is able to provide the service execution. The broker may require service deployment (in cases when a service is only available in the repository in image form). The interaction with a deployer and the deployment requirements and steps are detailed in subsection 5.4. The invoked broker should select a service and execute it taking into account the user requirements and the SLA terms.

We start examining the service brokering layer requirements from lower to upper levels. Below this layer we can find the service deployment and the services of various providers. The following requirements emerge from interacting with these lower layers:

– Service monitoring: information on service properties and states are needed by the brokers. This data can be published by the services themselves, or gathered by an external service or agent. Additional dynamic and availability information can be gathered by the deployment service (e.g. a service is deployed or only available as an image, what are the costs of deploying a service).

– Service description: there should be a common format, a service description language (an example is introduced in [12]) to store the service properties, semantics along with the additional information provided by service monitoring (including costs of service deployment and execution in terms of time, price, etc.). Note that unifying different service description languages is necessary for efficient central management, there will always be certain service groups that use an own standardized language for this purpose (e.g. grid services use the JSDL standard [19]).

– Service registry: the description of each service should be stored in a registry that can be queried by the service broker. The design of this component is the task of research thread B in Figure 1, and deployment, to be described in the next subsection, can also use and extend this service registry. Most requirements arise from the Service Brokering layer itself as follows:

– Service discovery: the role of this process is similar to the definition given by research thread A of the vision shown in Figure 1. For a service broker this task means information retrieval from the service registry.

- Service selection: this task requires self-adaptability – different processes are needed depending on the information found in the registry:
  - first a matchmaking process should be started in order to select the most suitable service by matching the requester's needs to the service properties; if there are more instances of a similar services available, the dynamic performance data, the SLA terms and the requester satisfaction (QoE - quality of experience [18], this also represents a link to the A3 research topic on Figure 1) rate of the services should be examined and compared. Various selection policies need to be defined to be able to adapt to different situations.
  - if there no such service is found (or only one or some of it are found, but they are unavailable or poorly performing), but a service image is available for deployment, the broker should decide whether to ask for a deployment or respond with rejection of the service request.
  - a basic self-healing property is the re-selection process, which is needed in case of a service execution failure – in this case the selection process should be restarted.
- Self-monitoring: another step towards self-healing operation is a monitoring agent that watches the heartbeat of the broker (this task is in connection with JRA-1.2):
  - to preempt overloading, under heavy load (when the number of service request exceeds a certain threshold) it notifies the broker to reject further requests in order to remain in a healthy state.
  - in case of halting due to some system failure, a process should be triggered that restarts the service.
- Agreement enforcement: brokers also need to take part in the agreement negotiation process. They provide information on service execution costs to upper layers (which will be stored in a broker registry detailed later). Once an agreement is made by higher level mediators, it should accept (or reject) the terms (e.g. agreed price and deadline for executing a service) that need to be ensured by the broker during service execution. Whenever an execution failure occurs, the broker needs quick reactions with high adaptability to find another service or restart the failed one, etc.; this kind of behavior should be governed by predefined policies that also affect the service selection process. If the broker rejects the agreement (not able to fulfill them) or cannot enforce it due to some failure, it should notify the mediator to select a different broker or renegotiate the SLA.

To deal with heterogeneity and with the growing number of services, special purpose brokers (for human-provided or computation-intensive services) or distributed broker instances should also be utilized and managed by the system. This requires a higher-level management of these brokers within this layer. This high level manager is responsible of brokering brokers, therefore we name this manager as a meta-broker, which component has the following requirements:

- Broker monitoring: the states and the performances of the brokers should be tracked.
- Broker description: brokers have various properties, which should be expressed with a general broker property description language. These properties include:
  - static properties: some of them are specialized for managing human-provided services, others for computing-intensive or data-intensive services;
  - and dynamic properties: if two brokers are managing the same type of services some of these may have longer response times, less secure or more reliable. Information on

the available services (e.g. type, costs, amount) also belongs to this category, since it changes over time.

- Broker registry: The broker properties should be stored and updated in a registry accessible by this higher-level manager. The update intervals of broker state changes and the amount of data transferred should also be set automatically, with respect to the number of available services and utilized brokers. Too frequent updates could lead to an unhealthy state and rare updates cause higher uncertainty and inefficient management. This registry should be more like a local database that makes this higher level brokering able to decide, which broker could provide the fittest service (according to the user requirements and SLA terms).
- Broker selection: Selecting a broker for a service request means a higher-level brokering problem. Self-adaptability also appears in this task: generally a bit higher level of uncertainty exists at this process compared to service selection, since the high dynamicity of the broker (and forwarded, filtered service) properties and availability cause volatility in the information available for this manager. To cope with this issue, several policies could be defined by sophisticated predicting algorithms, machine learning techniques or random generator functions. Load balancing among the utilized brokers should also be taken into account during broker selection. Finally basic fault tolerant operations, such as re-selection on broker failure or malfunctioning also need to be handled.
- SDL and translation: A generalized Service Description Language (SDL) is needed to be able to express all kinds of service requests (or properties). Different brokers may accept different languages, therefore the translation of SDL to these formats is also important (the manager should be able to speak with the utilized brokers).
- Broker invocation: Different brokers may have different interfaces that the meta-broker should be able to use.
- Self-healing: self-healing operation of the meta-broker is also needed. Just like in the case of brokers, a monitoring agent should watch the heartbeat of this manager component (this task is in connection with JRA-1.2):
  - to preempt overloading, under heavy load (when the number of service request exceeds a certain threshold) it notifies the manager to reject further requests in order to remain in a healthy state;
  - in case of halting due to some system failure, a process should be triggered that restarts the service.

Interacting with the upper layer means participation in the agreement negotiation. This step has the following requirement:

- Agreement negotiation and enforcement: In order to communicate with the higher level negotiator component (i.e. the meta-negotiator detailed in the previous subsection), a negotiation protocol should be selected and implemented. The meta-broker should gather and propagate information (stored in the broker registry) needed for agreement creation (available services of brokers and their execution costs). The agreed SLAs should be forwarded to the selected broker together with the translated service description. If the broker accepts the agreement, it should start its brokering process for finding, selecting and executing the required service. In case of a failure (in selection or execution), it should inform the negotiator of breaking the SLA or call for modification or renegotiation if it is possible.

5.4 Requirements for Service Deployment

Automatic service deployment is on the higher-levels of the service management concepts (see figure 8). It provides the dynamics to SBAs – e.g. during the SBA‘s lifecycle services can appear and disappear (because of infrastructure fragmentation, failures, etc.) without the disruption of their overall behavior, for other examples see the scenarios discussed in section 5.1. Service deployment process is composed of 8 steps:

1. *Selection* of the node where the further steps will take place
2. *Installation* of the service code
3. *Configuration* of the service instance
4. *Activation* of the service to make it public
5. *Adaptation* of the service configuration while it is active
6. *Deactivation* of the service in to support its offline modifications
7. Offline *update* of the service code to be up-to-date, after update the new code optionally gets reconfigured
8. *Decommission* of the offline service when it is not needed on the given host anymore

Automation of deployment is automation of all these steps. However each step requires different approaches, therefore they are discussed separately here. First of all the automation of the *target site selection*, which is required for local adaptation or for adaptation on the infrastructure layer level. Automation also includes the automated *preparation of the service‘s code and configuration* for installation. This preparation usually means the service code is split into smaller pieces to optimize code delivery to a given site. These smaller pieces are called packages, and they include dependency information about what other steps are needed to install themselves on a given host. The dependencies could include required configurations or packages which should be installed prior the current one. Packages help optimize the code delivery because they can be replicated on the infrastructure depending on their need, thus the more frequently requested packages can be acquired from local or at least closer sources (e.g. from a service code repository or from a simple HTTP/gridftp/etc. server). Further automation options are discussed among the different requirements proposed in the following sections.

Assumptions. In this section we call services *higher-level*, when they are built on top of automatic service deployment. Even though there could be other services which are built on top of service deployment, in this section we place deployment under the Meta-Negotiation (sec. 5.2) and Meta-Brokering (sec. 5.3) services as described in the previous sections. Therefore the offered interfaces and functionalities for these higher-level services will be further discussed during the collection of the requirements.

As an opposite to higher-level services, in this section we are going to refer *low-level services* as the ones which are used during the automation of the deployment process. A low-level service operates on the same node where its instance is running. Typical low-level services are including the management, adaptation, configuration and installation services. Adaptation, configuration and installation services are offering the functionality of a given deployment step discussed in the previous paragraphs. Management services however, are usually composite of the adaptation, configuration, and installation triplet. Installation services should be available on each node of the service infrastructure because they let other services to be installed on their node. It would be beneficial that all the other previously mentioned services are available on the nodes, however using the installation service it is possible to install and activate these services on-demand. Management, configuration and adaptation services are necessary to accomplish self-healing and local adaptation because the execution of the self-healing and local adaptation strategies going to involve them. As an exception, installation services are usually not useful for self-healing. Because installations can further degrade the health state of the already deployed parts of the service, which is the situation self-healing tries to avoid.

5.4.1 Deployment requirements to support self-healing services

The definition of *self-healing* is discussed in detail in section 2.3, however in case of service deployment there are further assumptions. In this section self-healing is the automated execution of an adaptation strategy on a service instance, where the automation is initiated within the service instance‘s node and the adaptation strategy only involves low-level services. The definition of the adaptation strategies can be found in the CD-JRA-1.2.2. In the deployment scope most important aim of self-healing is to try to avoid SLA violations and still maintain a healthy state. It is discussed about behaviour adaptation (see section 4.6 on page 14) that a single service instance should have multiple service interfaces available, and adaptation is simply achieved by activating one interface for the current needs and also deactivating an unhealthy one. In this section it is assumed that no new service interfaces are deployed on the node as a result of an adaptation strategy, however already deployed interfaces could be activated and deactivated with the help of the deployment system as required.

As it can be seen on figure 6, there are several decisions to make during the adaptation of a service. First of all, the decider requires monitoring events, and policies, secondly the planner requires guides, finally the executor takes actions. All of these have to be provided for a properly working self-healing infrastructure. In the next paragraphs we discuss how the decider, planner and executor can be supported by the deployment system, and what are their requirements to be able to make plans including deployment tasks.

Management services are essential parts of the automatic service deployment as they are providing among others adaptation and configuration services. Such features are commonly offered by service management systems, like the WSDM [16] OASIS standard. The management services are essential because they are required by the executor (see section 4.6 on page 14) in order to extend the possibly executed actions with the deployment steps. Having these services on a node the planner can include them in its plan. The advantage of the use of management services over arbitrary actions, that the deployment related actions can be uniform, and the plans could became simpler.

Because the management services offer sensitive functionality of a node, they should be secured and they might allow only internal access to them. As an option for higher-level adaptation these management interfaces might be accessible from composition, brokering or other higher-level services. This option needs the management services to be secured properly, resulting that only these higher-level adaptation mechanisms can access their interfaces. This way the management interface would open the possibility of reconfiguring a service as required for the higher-level services.

Packaging. The smallest individually manageable part of the service is called the package. It is the smallest, because it encapsulates the management metadata with the managed part. The easiest packaging option is to pack a service and its configuration options together. However services can be packaged in smaller pieces also. As a result these smaller pieces can be individually replaced or corrected if needed, thus there is no need for redeploying the entire service again from. However by separating the service into smaller packages, these packages should also include dependency information between the different portions of the service. Using the dependency information the service can be reunited again. Thus the management service should be able to handle dependencies between packages and it should be able to individually configure them.

Healing strategies are discussed in later paragraphs, however they are also concerned with packages because they usually involve configuration options of the packages. As a result the healing strategy does not need to deal with the full reconfiguration of the service if the reconfiguration of one of its tiny parts is enough to achieve healthy state again. These small adjustments are beneficial for the self-healing system's decision-making algorithm, because these steps usually affect fewer parts. Because the planning will use less configurable options it would be able to finish the plans earlier. This is not possible when the services are offered in a single package because to activate a new configuration, they have to be deactivated, decommissioned, installed, configured and then activated again. With the help of the packages only the necessary parts of the service has to go through this long process.

Monitoring terminology and requirements are discussed in workpackage JRA1.2 as the higher-level monitoring model, here we reuse their methodology, and identify the concepts of the monitoring in the scope of the requirements for service deployment. The decider from the figure 6 is connected with the monitoring system. It is the actual requester and consumer of the monitoring. The decider will request various domain specific measurements of the service instance, and it can also request the list of packages, or the list of interfaces available through these packages, etc. These last two examples are the monitored properties which should be collected by the deployment system. Depending on the different service interface configurations the decider should determine differently the current health state of the service instance. As a source for the monitored properties the information sources about the locally installed packages can be the deployment system's own registry or the packages themselves. In the packages the deployment system can also store the policies which can be applied when a given package is available. These policies should be offered towards the decider, which will meld the different package policies with its own policies together in order to determine whether there is a need for self-healing.

Healing strategies are adaptation strategies used in self-healing. Adaptation strategies are defined in CD-JRA-1.2.2 higher-level adaptation model. As we did with the monitoring, we also reused their adaptation methodology here to find the requirements of the service deployment for healing strategies. When the decider realizes that its policies suggest adaptation, then it defines the adaptation requirements for the planner. The planner determines what adaptation strategy to use. The deployment system should offer smaller strategy portions in which it describes how to achieve a deployment goal required by the planner:

– How to activate an inactive service interface

– What steps needs to be taken in order to change a specific configuration value

– It should be able to tell whether it is possible to do a runtime change or the healing strategy for a given configuration change should also include the deactivation and the activation of the service.

These strategy portions then will be embedded in the final plan. When the plan is executed the management services will do the adaptation of the adaptation subject - the service instance which is under self-healing.

5.4.2 Deployment requirements for local adaptation

Local adaptation is defined in section 2.2, however in the scope of deployment local adaptation is a simple extension of self-healing by extending the decider policies with conditions about the service instance‘s context. As an example during the decision making process the decider should take into consideration the health state of the surrounding services which offer the same funcitonality.

Service instances should offer interfaces to *share their health status* independently from an information service. The health status of a service instance does not need to be externally understood, the only requirement that other service instances, which offer the same interface should understand them. A monitoring infrastructure should be built similarly to the self-healing monitoring solutions, however the events and adaptation strategies should take into consideration the health state of the connected service instances. For example the service instance now can make decisions whether it has to prepare for an increased amount of requests. As a result it should use its management interfaces to *reconfigure itself* to make sure it will bear the future load. In case the local adaptation is offered without self healing capabilities, then the management interfaces will not be available for the service instance, therefore the locally adaptable services should decide together to use the automatic service deployment system to *deploy a new instance* which can cope with the increased needs and it could also decide to request the *decommission of a underperforming service* instance from the group.

Discovery is detailed in the deliverable CD-JRA-2.3.3, thus this paragraph just shortly summarizes the discovery needs of the deployment system in case of local adaptation. Service instances should not build on centralized discovery mechanisms to find other instances offering the same service interface in the SBA. Service instances should have *embedded discovery mechanisms* and they should use it as a failsafe solution. For example by using peer-to-peer mechanisms the service instances can decide to *locally increase the processing power* of a given service by deploying new instances in the neighborhood without even affecting the entire SBA. This could be useful when the SBA becomes partitioned or the service instances further away cannot feasibly serve the locally increased service requests.

The packages should be stored in a repository as part of the automatic service deployment system. This repository is a package repository, and it is not a single entity in the infrastructure but replicated. In case of new deployments, frequently used components can be replicated and also merged when the package retrieval patterns suggest – e.g. two packages are frequently downloaded together. Packages should also be stored with their configuration options, because healing strategies are usually simple maps between different situations and configuration options.

5.4.3 Deplyment requirements for the adaptation in the infrastructure layer

Adaptation in this case is restricted to the infrastructure level. There are no restrictions on what services can be included to adapt a service, however the whole infrastructure layer should offer a “single” interface towards the composition and BPM layers in order to enable information flow between the layers, and make it possible to adapt on information available only in the higher layers.

In the infrastructure layer the connection with the deployment system and the broker has to be investigated. This interface is partly discussed in the meta brokering section (see 5.3 on page 21). As it can be seen there the adaptation decisions affecting the deployment system are made on information mostly offered by third parties - e.g. service registries, information systems. However the deployment system should also provide information to the higher-level services. This information helps the these services to include deployment tasks in their adaptation strategies. For example the deployment system should provide information about the pricing of the different resources it can deploy services on. The pricing information can be forwarded to the meta-negotiation services. Another important example is the deployment system's estimates about deployment time of a given service on a given resource. With this information the service broker could decide on deploying a new service instance, if the current demand cannot be fulfilled without breaking the SLAs of the ongoing requests.

5.4.4 Deployment requirements for interface changing adaptation

Finally there is the case when the already deployed service interfaces are not sufficient for the adaptation of the service. In this case a new service interface will be introduced for the service, however this interface might be unknown to the SBA. Therefore the SBA also needs to adapt to the new, more suitable service interface offered after the adaptation procedure. As a result this adaptation procedure should also initiate adaptation on higher layers like composition or BPM - these layers and their interfaces are discussed in JRA2.1, and JRA 2.2. On the infrastructure layer the deployment system is used to change the interfaces on a given node. This should result the decommission of the previously installed service code. After this step the service code supporting the new service interface(s) is installed and configured. An intelligent deployment system should try to reuse all the previous packages, which are both supporting the old and the new service interfaces. To hold a package the deployment system should make sure the already available packages can be trusted because they are unchanged or they can be reconfigured to fit the needs of the new service interface.

5.5 Summary of requirements

Table 2 summarizes and structures the different aspects of requirements for each of the three functionalities of the conceptual architecture.

Table 2: Summary of requirements

Adaptability requirements for	Agreement negotiation	Service Brokering	Service deployment
Self-healing	negotiator failures	Broker failures, Meta-broker failures	management services, packaging
Decision support	selection of negotiation protocols, selection of the agreement languagues	Service selection, Broker selection, SLA negotiation, SLA enforcement	monitoring, healing strategies

6 Conclusions

The requirements for self-healing and local adaptation have been studied in the document regarding service execution, deployment and runtime management.

Local adaptation is only a first step in our research works but is always needed for self-healing purpose of services. We will use and refine the Dynaco operational framework that fits the conceptual adaptation framework of deliverable CD-JRA-1.2.2. Decision tools for local adaptation and relations with monitoring tools will be studied.

The requirements for self-healing and decision support in deployment and runtime management have been analyzed and classified in the framework of a conceptual architecture. The framework unifies three main functionalities that are subject of planned research work: negotiation, brokering and deployment. The aim is to provide and modify (adapt) services dynamically and in a (partially) autonomic way at run-time, according to some temporal, spatial or other needs.

In our view, some the problems illustrated in the use cases in the vision document [6], in section 3 and in Section 5.1 can be addressed by local adaptation, negotiation, brokering and deployment.

These are considered as the cornerstones of the research work in Thread C of WP2.3 research framework and hence, the requirement analysis in this deliverable is a baseline for future research. By revealing the requirements of these areas, we are able to target these needs in our future work in order to build a service infrastructure that will be capable of self-adaptation within service executions with diverse service types and environments. Particularly, we identified connections with several WPs (e.g. JRA1.2, JRA2.2) in the scope of self healing and local adaptation.

We also found requirements for self-healing on management, packaging, monitoring and healing strategies. These basic requirements lead us to the importance of the health status sharing between service instances and three new adaptation actions for reconfiguring, deploying and decommissioning of these instances. In the later phases of this project we will further develop ideas on these new requirements against the service infrastructure.

References

[1] Yolanda Becerra, David Carrera, and Eduard Ayguad. Batch job profiling and adaptive profile enforcement for virtualized environments. In *Proceedings of 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, Weimar, Germany, February 2009.

[2] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Dynamic adaptation for grid computing. In Peter M. A. Sloot, Alfons G. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers)*, volume 3470 of *LNCS*, pages 538–547, Amsterdam, June 2005. Springer-Verlag.

[3] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Supporting adaptable applications in grid resource management systems. In *8th IEEE/ACM International Conference on Grid Computing*, 19-21 September 2007.

[4] P. Combes, F. Lombard, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In *Proceedings of the 7th Asian Computing Science Conference*, pages 110–124, janvier 2002.

[5] D. Dasgupta, Z. Ji, and F. Gonzalez. Artificial immune system (ais) research in the last five years. *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, 1:123–130 Vol.1, Dec. 2003.

[6] Marco Pistore (editor). Integration framework baseline. Technical Report CD-IA-3.1.1, S-Cube, 2009.

[7] W.G. Fenton, T.M. McGinnity, and L.P. Maguire. Fault diagnosis of electronic systems using intelligent techniques:a review. *IEEE Transactions on Systems, Man, and Cybernetics*, 31(3):269–281, 2001.

[8] Ira R. Forman and Nate Forman. *Java Reflection in Action.* Action series. Manning, 2004.

[9] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Commun. ACM*, 40(10):88–96, 1997.

[10] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, 2003.

[11] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, 2007.

[12] Willem-Jan van den Heuvel, Jian Yang, and Mike P. Papazoglou. Service representation, discovery, and composition for e-marketplaces. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 270–284, London, UK, 2001. Springer-Verlag.

[13] Nikola K. Kasabov. *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering.* MIT Press, 1996.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedins of European Conferebce on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg and New-York, 1997. Springer-Verlag.

[16] Heather Kreger, Kirk Wilson, and Igor Sedukhin. Web services distributed management: Management of web services (wsdm-mows) 1.1. web, August 2006.

[17] G. M. Marakas. *Decision support systems in the twenty-first century (2nd edition).* Prentice Hall, 2002.

[18] Aad Van Moorsel. Metrics for the internet age: Quality of experience and quality of business. Technical report, 5th Performability Workshop, 2001.

[19] OGF. Job Submission Description Language, 2006.