AutoI Orchestration Plane and Interfaces

Deliverable D2.2

Autonomic Internet (AutoI) Project

FP7-ICT-2007-Call 1 - 216404

Abstract: This document describes the Orchestration Plane of the Autonomic Internet architecture and its interfaces. It improves the description of the orchestration components provided in previous deliverables D2.1 and D6.2. It presents the implementation of the orchestration components and its workflow. It also provides scalability and stability results for each of the orchestration tasks performed.

Document Properties

Properties:

Document Number:	FP7-ICT-2007-CALL1-216404 AutoI/D2.1
Document Title:	AutoI Orchestration Plane and Interfaces
Document responsible:	Zeinab Movahedi, Daniel Fernandes Macedo, Guy Pujolle
Author(s) / Editor(s):	Daniel Fernandes Macedo (LIP6), Zeinab Movahedi (LIP6), Guy Pujolle (LIP6), Javier Rubio Loyola (UPC), Wei Koong Chai (UCL), Alex Galis (UCL)
Reviewed by:	Joan Serrat (UPC), Javier Rubio Loyola (UPC), Alex Galis (UCL), Richard Clegg (UCL)
Dissemination Level:	Public
Status of the Document:	Final Version
Version:	1

Revision History

Revision	Date	Issued by	Description
V0.1	15/07/09	Daniel Macedo	Initial draft.
V0.2	20/07/09	Daniel Macedo	Initial description and interfaces of the Dynamic Planner and Basic Behaviour.
V0.3	27/07/09	Zeinab Movahedi	Description of federation and corresponding diagrams.
V0.4	03/08/09	Daniel Macedo	First draft of the governance Behaviour.
V0.5	10/08/09	Daniel Macedo	First draft of the description of the Dynamic Planner. Governance modifications suggested by Joan Serrat and Javier Rubio Loyola (UPC).
V0.6	24/08/09	Daniel Macedo	Incorporated the contributions of UPC and UCL to the negotiation behaviour.
V0.7	19/09/09	Zeinab Movahedi	Improvements to the federation behaviour.
V0.8	28/09/09	Daniel Macedo, Zeinab Movahedi	Added policies for the distribution, negotiation and federation behaviours, and the dynamic planner. First draft of the scalability and stability of the dynamic planner.
V0.9	19/10/09	Zeinab Movahedi	Federation Behaviour scalability and stability.
V0.10	23/10/09	Daniel Macedo	Improvements in the interfaces section, added scalability and stability of the governance behaviour. Initial text for the scalability, stability and policies of the negotiation behaviour.
V0.11	02/11/09	Daniel Macedo	Added a new section describing the changes produced from D2.2 when compared to D2.1.
V0.12	12/12/09	Alex Galis	Comments/review
V0.13	15/12/09	Zeinab Movahedi	Added detailed description of the Coalition Formation Negotiation approach by Javier Rubio-Loyola (UPC) and the Bargaining Negotiation approach by Wei Chai (UCL). Review of the document
V0.14	31/01/10	Zeinab Movahedi	DOC implementation description added (sent by Giannis)
V0.15	10/06/2010	Richard Clegg	Comments/review
V1.0	23/06/2010	Alex Galis	Final edit

Table of Contents

Executive Summary

Next generation network and services require the optimisation of multiple types of orchestration mechanisms, enabling network operations (i.e. optimisation for initialisation, dynamic reconfiguration, adaptation and contextualisation, dynamic service deployment support and other tasks) and service tasks to be optimised. In the AutoI project, orchestration refers to the mediation of several Autonomic Management Systems with each other in an attempt to deal with the problem of management domain heterogeneity and integration. The later is a relevant innovation of the Autonomic Internet (AutoI) architecture as it provides the autonomic integration and federation of several autonomic management systems.

The objective of this deliverable is to refine the description of the Orchestration Plane (OP) within the overall AutoI architecture. This document improves the functions, requirements and the concepts behind the operation of the components that make the orchestration plane, the DOCs, which were presented in previous deliverables. A DOC, or Distributed Orchestration Component, is a functional entity of the AutoI architecture that deals with inter-domain management tasks, such as the federation, negotiation, governance, and distribution of management domains. More details are presented for the implementation of each of the orchestration tasks. We define the policies needed for those tasks, and their interaction flow within the OP and with other components of the AutoI architecture. Finally, we analyse the stability and scalability of the OP and its components.

This deliverable is related to the following overall project objectives: *Objective 1 - Define an autonomic architectural solution for supporting Internet services.* The OP will allow the interoperation of different management domains, which may use different sets of business goals/policies, and which may have different QoS requirements. The OP will serve as a broker, arranging the interconnection of different management domains. It will automatically try to solve conflicts and suggest solutions to optimize the behaviour of autonomic FCAPS solutions. *Objective 7 - Control of virtual resource algorithms.* A new set of control algorithms will be defined providing a faster, more optimized response to changes on resources or to user demands. These control algorithms will be contained within an orchestration plane. This optimization is achieved via re-configuration of the parameters of the virtual resources (e.g. routing tables, attribution of classes to flows).

Abbreviations

AMS	Autonomic Management System
ANPI	Autonomic Networking Programming Interface
AutoI	Autonomic Internet Project
DOC	Distributed Orchestration Component
DP	Dynamic Planner
DSL	Domain-Specific Language
FCAPS	ISO Telecommunications Management Network model and framework for network management. FCAPS is an acronym for Fault, Configuration, Accounting, Performance, Security, which are the management categories into which the ISO model defines network management tasks.
FIN	Future Internet Network
IBM	International Business Machines
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
KP	Knowledge Plane
MP	Management Plane
NE	Network Element
OP	Orchestration Plane
QoS	Quality of Service
SE	Sequential Equilibrium
SLA	Service Level Agreement
SP	Service Enablers Plane
SPE	Sub-game Perfect Equilibrium
VP	Virtualisation Plane
vSPI	Virtualisation System Programmability Interface

1. Introduction

Autonomic management systems, as initially described in the IBM manifesto [23], have been defined as management systems of a single system. In networking, a number of autonomic management systems have to perform different management tasks covering various nodes, links and services. Due to the existence of several management standards, different protocols and different vendors, managing a network is much more complex than managing a single isolated system. Thus, it is not practical to devise a single autonomic control loop that autonomically adjusts all the FCAPS (fault, configuration, accounting, performance and security) aspects of a network. This means that we need to define one or more autonomic loops for each of those management aspects in order to simplify the design of each control loop.
However, the operation of the network management system will depend on the interaction of all those control loops, which must ensure, amongst other key aspects, that the network operates within normal parameters set by the business goals of the operators. Also, the decisions of a control loop may go against the objectives of another one. As an example, an autonomic security component may use a heavier encryption scheme to improve the security of the network, however this encryption scheme may require too much processing and bandwidth, reducing the maximum throughput of the network to a level below the performance dictated by the Service Level Agreement (SLA).
In networking, two sub-networks having different managers (or two administrative domains) must interconnect. This requires that the protocols as well as the configuration of the domains (i.e. security policies, QoS and SLAs) are compatible. If they are not, either a re-negotiation and re-configuration process is required, or translation services (gateways) must be installed in the border of the two networks.
In order to solve those problems we introduce a new system, or plane, called the Orchestration Plane (OP). It enables cooperation of the various autonomic control loops ensuring their decisions are not orthogonal. This cooperation, or orchestration, ensures that the overall optimization goals of each autonomic component and control protocol are aligned with the goals and SLAs defined for the entire network. Orchestration also allows administrative management domains run by different operators or administrators are able to automatically adjust their configuration to accommodate the federation of networks.
The need for an OP arises from the deployment of several autonomic control loops with different administrators or management goals, which would not be able to interoperate without a set of translation, negotiation, federation, and deployment functions. Thus, orchestration deals with the meta-management of autonomic management systems, that is, the deployment and reconfiguration of autonomic management control loops in order to allow their interoperation. This is achieved based on a set of high-level goals, defined for each of the managed network domains that form the orchestrated network. The OP ensures the interoperation of management systems, even though those systems use different set of high-level goals and management standards. This process may be accomplished through the negotiation of new SLAs and policies, the deactivation of conflicting management systems followed by the activation of other management systems, or the migration of such systems or parts of them within the orchestrated network. The entire orchestration process is governed by Orchestration Policies, which dictate what are the compromises that each of the managed domains are willing to make for the sake of interoperability.

This deliverable refines the Orchestration Plane concept as defined in deliverable D2.1 [19]. It provides an enhanced description of the components that make up the

Distributed Orchestration Component (DOC), which is the component that implements the OP concept. The policies that manage the operation of each of the components are presented, as well as the interfaces used by the DOCs to communicate with the other components of the AutoI architecture.

1.1. Orchestration Improvements from deliverable D2.1

This deliverable improves the concepts and architecture of the orchestration plane that was presented in deliverable D2.1. An in-depth description of orchestration components and their operation is provided as well as policies and analysis of their stability and scalability.

The following aspects of the orchestration concept are improved:

- The topology of the orchestration plane is defined with regards to the administrative domains deployed on the network.
- The interations of the DOCs and AMSs are clarified, defining that one AMS interacts with only one DOC, in order to simplify the architecture.

On the orchestration architecture, the following contributions are added:

- A more complete description of the Dynamic Planner and its operation. This description considers the stability and scalability of the dynamic planner.
- A language for the description of workflows, which is used in the dynamic planner to control the Behaviours and AMSs.
- Policies for the Dynamic Planner.
- A generic interface for all behaviours has been proposed, along with its policies. This interface defines the minimum control functions needed in the behaviours for their effective control.
- The federation, governance, distribution and self-governance are presented in details. This deliverable improves their description in the following ways:
    - We reviewed their functions of the Behaviours with regards to D2.1.
    - This deliverable presents the operation in a high-level description, using state and class diagrams.
    - The policies specific of the four Behaviours are presented.
    - We analyse the stability and scalability of those Behaviours.
- Two proposed negotiation approaches for the negotiation behaviour. One of them is based in coalitions, while the other is based on a bargaining approach.
- Some prototype details regarding the implementation of enhanced DOC is described which includes appropriates interfaces to other planes.

1.2. Organization of the deliverable

This deliverable is organized as follows. Section 2 describes the orchestration plane concept and highlights its architecture, showing the motivation for the use of a Dynamic Planner and Behaviours. Section 3 describes the Dynamic Planner, its policies and workflows. Section 4 presents the Behaviours, and the four core behaviours. The interfaces of the DOCs with other components of the AutoI architecture are discussed in Section 5. Section 6 concludes the deliverable.

2. Orchestration Plane Definition Refinement

The purpose of the Orchestration Plane is to govern and integrate the behaviours of the network in response to changing context and in accordance with applicable high-level goals and policies. It supervises and integrates all other planes behaviour insuring integrity of the Future Internet management operations. The Orchestration Plane can be seen as a control framework into which any number of components can be plugged in order to achieve the required functionality.
The Orchestration Plane would also supervise the optimisation and the distribution of knowledge within the Knowledge Plane to ensure that the required knowledge is available in the proper place at the proper time. This implies that the Orchestration Plane may use either very local knowledge to deserve a real time control as well as a more global knowledge to manage some long-term processes like planning.
The Orchestration Plane would host several Autonomic Management Systems (AMSs). It is made up of one or more Distributed Orchestration Components (DOCs), and a dynamic knowledge base consisting of a set of data models and ontologies and appropriate mapping logic and buses. A DOC, which controls a single orchestration domain, enables the AMSs of the domains to communicate and cooperate with each other. Orchestration domains coincide with the administrative domains (resources under the same administrative responsibility) and, as a consequence, the set of orchestration policies within the orchestration domain is homogeneous.
Each AMS represents a set of virtual entities, which manage a set of virtual devices, sub-networks, or networks using a common set of policies and knowledge. The set of virtual resources managed by each AMS are non-overlapping. As a consequence, the responsible AMS will forcibly implement all self-* functions for its managed resources. This assumption does not imply that each resource will be used for only one service or network. A physical resource, being divided into several virtual resources, can be used for different services or networks. Each of those virtual resources, however, will be dedicated to a single service, and as such will be managed by only one AMS. The AMSs access a knowledge base, which consists of a set of data models and ontologies.
The Orchestration Plane acts as control workflow for the AMSs of the orchestrating domain, ensuring bootstrapping, initialisation, dynamic reconfiguration, adaptation and contextualisation, optimisation, organisation, closing down of AMSs. The Orchestration Plane provides assistance for the Service Lifecycle Management, namely during the actual creation, deployment, activation, modification and in general, any operation related to the management services. The DOC enables the following functions across the orchestration plane:
Federation: each AMS is responsible for its own set of virtual and non-virtual resources and services that it governs as a domain. The Federation enables a set of orchestration domains to be combined into a larger orchestration domain, guided by common high-level goals, while maintaining local autonomy.
Negotiation: in AutoI negotiation can take place between autonomous entities with or without human intervention. DOCs and AMSs are the entities engaged in negotiations to achieve their goals. Each AMS advertises a set of capabilities (i.e., services and/or resources) that it offers for use by other AMSs. The DOC acts as an arbitrator, performing negotiation between the AMSs for the fulfilment of a specific SLA, defined by the orchestration policies, user requirements and high-level goals.
Distribution: the DOC provides communication and control services that enable tasks to be split into parts that run concurrently on multiple AMSs within the Orchestration Plane.
Governance: each AMS can operate in an individual, distributed, or collaborative mode

(i.e. in federation). Thus, AMSs may from time to time take local decisions that go against the goals of the orchestration domain. Thus, the DOCs monitor the actions of the AMSs, triggering responses (i.e. re-negotiation of AMS and/or DOC policies, closing down or bootstrapping of AMSs) to such violations.

System Views: the DOC is responsible for managing the system views that are stored and diffused using the knowledge plane (detailed in D4.1, section 3.3.4 [16]). DOCs will fetch the information required for their operation from the AMSs as well as the services and resources (through the vSPI interface defined in D1.1, section 7 [17]).

Throughout this document the term high-level policies or business objectives are particularly linked to high-level aspects of management and control over a given system. This is, by no means, aligned to economical aspects such as pricing strategies. As described in the deliverable D3.1, policies in the AutoI architecture follow a *policy continuum* [15]. In the *AutoI Policy Continuum*, policies are refined from high-level policies into lower level policies that can be applied to a specific instance of equipment. In AutoI, the following levels of policies are defined:

- **Orchestration level policies –** These policies control the negotiation, distribution, governance, and federation tasks of the orchestration plane.
- **System level policies –** Those policies are related to the autonomic management systems (AMSs). System level policies are used to define the operation of the AMSs.
- **Component level policies –** Component level policies manage the virtual components defined in the AutoI architecture.
- **Instance level policies –** Instance level policies are embedded inside devices that can perform their own decision functions.

The orchestration plane deals mostly with orchestration level policies. Within orchestration policies, negotiation, federation, distribution and governance policies are defined. Those control the possible actions of the DOCs on each of those tasks. DOCs deal with lower level policies only during negotiation processes. In those cases, the DOC must check that the *system level policies* produced by the AMSs are aligned with the *orchestration policies*. To improve the readability of this document, from now on the term *policy* refers to *orchestration policies*. Other levels of policies (i.e. *system level policies*) will be clearly contextualized by using their full name.

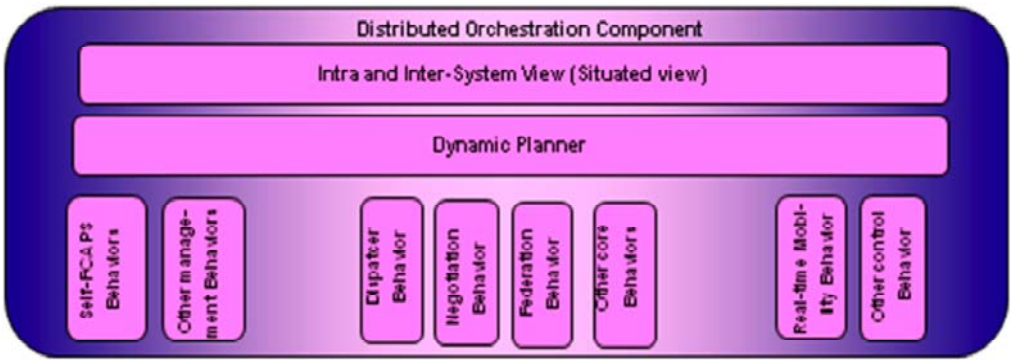

Figure 1: Architecture of the DOC

The architecture of the DOC is shown in Figure 1. The Dynamic Planner component bootstraps and closes down AMSs and Behaviours, following a set of workflows. Its actions are dictated by the orchestration policies and the workflows. Behaviours complement the Dynamic Planner, implementing the task specific actions that are required from the OP.

They use specialised knowledge of the problem at hand in order to solve it efficiently. It is up to the DP to choose the right Behaviour for each given task, controlling its lifecycle: instantiation, execution, and finalization.
Behaviours perform the specific/individual orchestration actions required to be performed by a DOC and also represent specific management tasks on the network. The main behaviours are *distribution*, *federation*, *negotiation* and *governance*. Those Behaviours are described in detail in the following sections.
DOCs use the knowledge plane described in deliverable D4.1 to store and disseminate the information required for their operation. This information can be decoupled into two parts, or views, according to their relevance to a given DOC. The Intra-System View concerns information required to orchestrate the services within the orchestration domain, while the Inter-System View deals with the orchestration of several orchestration domains. The Intra-System View contains information that enables DOCs to become aware of the particular situation that they are now in; the Inter-System View provides similar information for collaborating DOCs.
The following sections of this document provide an in depth description of the DP, the Behaviours and the interfaces of the DOCs. For each component of the DOCs, we describe their interaction within the OP and with other planes. We also describe their policies and comment on their scalability and stability.

3. Dynamic Planner

The Dynamic Planner is the central entity of the DOCs. It is responsible for dispatching Behaviours, which will take care of specific orchestration tasks. The DP relies on behaviours for the implementation of specific orchestration behaviour, so its design can be as generic as possible. The DP relies on orchestration policies and workflows, defined by the operators. Those will define the actions to be performed by the DP to accomplish any orchestration tasks. To avoid unnecessary resource consumption of the networking components, the DP is an event-based component. It relies on the monitoring facilities of the knowledge plane, as well as those of the governance behaviours, to trigger notifications of important event changes. Finally, the DP distributes policies and SLAs to the AMSs. This is performed at the deployment of new AMSs, or whenever the orchestration policies change.
The DP, being a generic execution engine of orchestration tasks (or meta-management of self-governing management systems), requires tools to describe the tasks that must be executed when certain events take place. This is performed using the workflows detailed in the following section.

3.1. Orchestration Workflows

Orchestration workflows are event-based, that is, the DP acts only when events occur. Events may be triggered from within the network (i.e. a conflict has happened), or from the outside (i.e. the operator defines a new set of goals for the OP). The following types of events exist in the OP:

- **Parameters changes:** A change occurred in a set of variables, which have values within a certain range. One such event could be the delay of a link becoming too high, which may require the renegotiation of the SLAs or the redeployment of an AMS.
- **Conflicts:** A conflict within two AMSs or behaviours has been detected. Such conflicts will usually come from the governance behaviours.
- **Federation requests:** The operator, or an AMS, has requested a federation action.
- **Separation of a federation requests:** The operator, or an AMS, has requested the separation of two networks.
- **Distribution requests** – The operator, or an AMS, has requested that a new AMS to be deployed.
- **Request to close down a component:** The operator, or an AMS, has requested the closing down of an AMS.
- **New goals and high-level policies:** The operator has defined new goals, policies or user requirements that must be satisfied by the DOC.

Those events trigger workflows, which are described in a domain-specific language (DSL) [15]. The language uses an orchestration vocabulary to simplify the programming of workflows. The idea of this meta-language is to offload most of the functionalities to the Behaviours or to the AMSs, so the DP stays simple and fast. Thus, the vocabulary of this DSL is quite small, and the DP relies on the implementation of specific Behaviours to perform the bulk of the orchestration tasks. For example, if there is a need to solve a specific problem, the DP will identify that such a problem occurred by monitoring certain variables within a set of pre-defined boundaries, and then it will trigger a Behaviour that is suitable for that situation. The vocabulary is as follows.

- **wflow**: Defines the name of the workflow, which will be used in internal operations of the DP (i.e. to call another workflow in the middle of the execution of another workflow).
- **startup**: This primitive is used to start up Behaviours or AMSs. The caller defines the name of the component that must be started up, and next the distribution Behaviour is called to start up the required component.
- **closedown:** Used to close down a Behaviour or AMS.
- **call**: Calls a function on a Behaviour or AMS. The functions are defined by each of the behaviours. All the behaviours export some basic functions, described in Section 4.1. Furthermore, behaviours may implement its own functions, which are accessible for invocation from the DP workflows.
- **do … until**: Makes a loop of calls, which stop after a certain condition is reached. This condition will be defined by watched properties of a system or by a timeout event.
- **timeout … otherwise**: Defines timeouts for the orchestration process. The timeout can be used in **until** loops, or in blocks. In the second usage, the timeout primitive is used in conjunction with the **otherwise** primitive to specify that the block of commands *after* the timeout *up to* the otherwise call must complete within a certain time limit. The time limit is defined in seconds.
- **if/then/else**: Used to build conditions in the workflow. This may be useful, for example, to define actions to be performed if a certain deployment terminates correctly, and another if this deployment does not terminate correctly.
- **loadwflow**: This primitive loads another workflow and terminates the current one. This is useful for treating specific behaviour that may arise when one workflow is running, which is not possible to be treated within this workflow. For example, if a workflow dealing with the distribution of a web services encounters a complex conflict in the policies of the components, it may load another negotiation workflow.
- **trigger:** Used within a workflow to associate another workflow to a certain event, defined by a set of conditions. This is useful for defining the actions that must be taken whenever a conflict or an abnormal configuration occurs. It may also be used to automatically replace components when the current network context is not favourable to the existing set of deployed AMSs.
- **parallel block:** This block, which starts with the keyword **parallel**, indicates tasks that can be done concurrently. The aim of this primitive is to reduce the deployment and execution time of the orchestration tasks. As an example, if the distribution of a new service requires the deployment of several AMSs, the start-up of all the new AMSs can be done simultaneously using this primitive.
- **accept/reject:** Those primitives are used for policies, goals and user requirements. If the policies are accepted, they are added to the set of old policies. If they are rejected, then a signal is sent to the caller of the workflow. This is used, for example, to signal an AMS that a certain set of policies cannot be applied to the entire network because they are not aligned with the orchestration policies.
- **Events:** Workflows are triggered by the events described in the previous section. Hence, each of the events described has an associated keyword in the DSL. Those are indicated at the start of the workflow, in order to identify on which situations the workflow must be triggered. The types of events are **Onchange**, **Conflict**, **Federate**, **Unfederate**, **Distribute**, and **Closedown**.
- **Mathematical and logic operations on control variables**: Finally, some tasks may require counters or other types of control variables. For example, we may wish to abort a process if it does not complete after ten iterations. Thus, the DP provides

setting a value, summations, subtraction, division, multiplication and modulo operations over integers.

Let's illustrate the usage of this language with an example. Suppose that we will define a workflow for the federation of IPv4-IPv6 networks. This will require the negotiation of certain parameters, as well as the deployment of a tunnel. To allow this process to complete within acceptable time limits, we define that the two networks being federated must agree to a common set of policies after ten configuration negotiation rounds or five minutes. Further, if the IPv4 network is upgraded to IPv6, the FederateHomogeneous *workflow is triggered in order to re-configure the network. This workflow could be expressed as in*
Figure 2:

```
wflow FederateHeterogeneous;
Federate:

If N1.IPversion == 4 && N2.IPversion == 6 {
        timeout 300 {
                trials = 0;
                do  {
                        startup IPNegotiation;
                        call IPNegotiation.triggerNegotiation();
                        trials = trials + 1;
                } until trials == 10;
                if trials == 10 {
                        loadwflow FederationHeterogeneousFailed;
                }
                else {
                        startup IPv6Tunnel (N1, N2);
                }
                closedown IPNegotiation;
                trigger onchange N1.IPversion > 4, FederateHomogeneous;
        } otherwise  {
                loadwflow FederationHeterogeneousFailed;
        }
else  {
        loadwflow FederateHomogeneous;

}
```

Figure 2 Example DP workflow.

3.2. Interfaces of the Dynamic Planner

The interface described in this section is used by Behaviours and AMSs to have access to the facilities of the OP. The DP is the core component of this plane, acting as the coordinator of all orchestration tasks. The OP uses the functions described below to control the behaviours. Those are triggered by the workflows, in the case of calls from the DP to the behaviours, or are mapped into events when the behaviours access the DP. For example, the function call **notifyServiceFeatures** will be mapped into a **Distribute** event on the DP workflow. Thus, while administrators will use the DSL described previously to program the DOCs, the behaviours will rely on the interfaces described in this section.

Behaviours may extend the functions listed in this section with its own, in order to accomplish its goals. The basic DP interface is composed of the following functions:

- **setHighLevelGoals(SLA, UserRequirements)**: Used by operators to set the high level goals for the network. Those are pushed into the AMSs.
- **notifyCompletion(Status)**: This is called by AMSs or Behaviours to indicate that their task has been completed. The status is equivalent to classic UNIX return values, in which the component indicates if it was successful in its task or not, and what was the error in this case.
- **Boolean = arePoliciesAligned(Policies)**: This function is used by the Behaviours or AMSs to check if their policies are aligned with the high level goals of the network, specified by the SLAs and user requirements. The return of this function is a Boolean, indicating if the policies are aligned or not. If they are not aligned, the Dynamic Planner may either terminate the Behaviour or the AMS, or it may trigger a negotiation.
- **notifyServiceFeatures(SLAs, UserRequirements)**: Used to identify that a new service must be deployed.
- **trap(Parameter):** This call occurs whenever a certain event, defined using the **monitor()** primitive of a Behaviour, has occurred. As in classic management, a TRAP allows the OP to receive event-based notifications of important changes. This may in turn trigger an internal reaction, such as the execution of a new workflow or the (re)- deployment of an AMS.

3.3. Dynamic Planner Policies

The DP uses policies to set limits on the execution of the workflows defined by the operator. Those policies define the maximum amount of resources (CPU, memory, time, number of running workflows) used for each of the workflows. Orchestration policies also define the action taken by the DP when a policy fails or when a certain orchestration event is not captured by any workflow. The DP policies also provide means to improve the evolution of the installed workflows and policies. By means of default policies, operators are able to identify situations that were not foreseen at the design of the workflows and policies. Further, policies are one of the tools to ensure the stability of the DP.

Resource limiting policies: resource limitation policies at the DP level in essence assure the stability of the system. They are put in place to terminate ill-behaved workflows, which in our definition are workflows using too much resources or taking too much time to complete. Those policies are defined on a per-workflow basis, as well as a generic policy for all the workflows. In case a certain workflow does not have an associated resource-limiting policy, the default limiting policy is used if it exists. Being the centralised entity that controls the bootstrap and closing down of Behaviours, the DP also defines resource limitations to Behaviours by means of specific policies. Those policies have the same properties and functions of the policies described previously for workflows.

“Multi-tasking” policies: they limit the amount of multi-tasking of each workflow, that is, it constrains the number of workflows that may be triggered by each workflow. A DP-wide policy can also be defined, in order to specify the maximum amount of workflows running at each DP. Excess workflows may be put in an execution queue, or simply rejected. Similar to the resource limiting policies, multi-tasking policies are defined to improve the stability of the DP platform. They may catch ill-behaved workflows or design flaws in the writing of the workflows. Again, those policies are either defined for each specific workflow, or for the all the workflows of the DP, allowing certain workflows to create workflows in excess of the other workflows’ default limits.

Default action policies: those policies define actions to perform when a workflow fails, or when no workflows are attached to a certain event. This approach is similar to the *default* action of *switch* clauses in programming languages, which is used to catch unpredicted situations or to handle errors. One of the uses of default action policies is to communicate to the operator that the OP found a situation that was not contemplated by the installed workflows. As an example, when an ill-behaved workflow overflows its allocated resources, the operator could be notified, receiving the name of the workflow as well as some measurements related to the state of the network and the state of AMSs and Behaviours. Similarly, in some situations the installed workflows may not deal with a certain event. In case this happens, a default action is taken. This action may consist of notifying the operator, as well as shutting down or restarting the problematic AMSs.

3.4. Stability and Scalability of the Dynamic Planner

Stability: as described in the previous section, the Dynamic Planner incorporates a set of policies in order to limit the resources and actions of the workflows, and also to reduce the effect of design failures in the writing of workflows. The same policies also diminish the impact of malicious workflows in the context of a DoS attack by constraining the amount of allocated resources used by the workflows.
Besides the measures presented below, which are embedded in the DP, one could also create specific Behaviours tailored to the monitoring and detection of failures and/or misbehaviours in other behaviours or AMSs. Those Behaviours could monitor a set of Behaviours or AMSs, ensuring that they operate according to certain limits. Since no entity would monitor the “stability behaviours”, they should be as simple as possible in order to be easily verifiable and to avoid implementation errors.
Scalability: the definition of different workflows, which may activate other workflows, allows for the scalability of the DP. It reduces the complexity of the workflow, since they may be broken down into smaller elements, each handling a subset of cases or configurations. At the same time, the managers and operators of the network must be aware of this in order to avoid the conflict of different workflows. This is similar to the issue of several policies being triggered for the same event in policy-bases systems. In those systems, the solution is to define priorities, so a certain policy may overwrite others, or be executed first or last. A similar approach could be applied in the DP.

4. Behaviours

Behaviours are the specialized components of the orchestration plane that are responsible for implementing the orchestration tasks. Due to the specification of the Behaviours, the OP is able to cope with different management and virtualization technologies. Behaviours are essentially software components that are executed by the Dynamic Planner on a per-case basis. The Dynamic Planner starts up the appropriate Behaviours, as defined by its workflows, passing them the parameters that they need to perform their function. As an example, the DOCs could implement Behaviours for the negotiation of high-level policies, the distribution of tasks, the creation and destruction of services and virtual routers. Behaviours interact with each other when necessary, i.e. the federation Behaviour may interact with QoS-related Behaviours if the required QoS couldn't be met when two networks are joined.
This section details the basic behaviour interface of the behaviours as well as the most important behaviours, which implement the governance, distribution, negotiation, and federation tasks that are carried out by the OP. Those are called the **core behaviours**, since they will be required for the operation of any autonomic network based on the AutoI approach. Other types of behaviours can also be defined for an improved management of specific cases. For example, mobility management behaviour, an accounting behaviour, among others, could be deployed.
Finally, knowledge update behaviours may be defined. Those behaviours are responsible for updating the intra- and inter-system view that is needed for the core behaviours or for the specialized behaviours. Knowledge update behaviours define the "What, When and Where" of the information: What information to collect, when to collect, and from whom (where). Those Behaviours are specific to each service, however the whole set of these Behaviours supervises storage of information in the Knowledge Plane. Each AMS requires pre-defined knowledge as well as runtime data. Mapping logic enables the data, represented in a standardized information model, to be transformed into knowledge and combined with knowledge represented by ontologies. Since those behaviours are specific to each application and implementation, this deliverable only presents the interfaces required for their interaction with the knowledge plane.
In order to control the behaviours, a set of standardized functions must be implemented. Those control the lifecycle of the behaviour (start up, closing down) and its parameters (policies and goals). Specific behaviours (i.e. a negotiation behaviour, a distribution behaviour) may specify more functions that may be called by the DP or by external components. The basic behaviour interface is described in the following section.

4.1. The Basic Behaviour Interface

In order to communicate with the Dynamic Planner, all Behaviours implement this set of basic primitives. Those primitives were derived from the basic functions that most behaviours do, based on previous deliverables (D4.1, D2.1, D5.1) [16],[17],[19]. This allows the DP to perform a set of basic operations on the Behaviour, such as starting up and closing down and changing their policies. Hence, every core behaviour or AMS Behaviour (a stub to each AMS implementation) implement the primitives below:

- **startUp(ParameterList)**: Called to create a new Behaviour or AMS stub, which may then create an AMS. This function receives a generic list as parameters, allowing the service to be created, the goals that must be respected or the user requirements that are relevant to this task to be passed at initialisation to the

Behaviour.

- **closeDown()**: Deactivates the AMS or Behaviour.
- **Boolean = addPolicy(Policy)** and **Boolean = addGoal(SLA)**: Used to push new policies and high level goals, respectively, into Behaviours or AMSs after they have been initialized. This may be necessary in cases where the operator sets a new set of high-level goals, or as an outcome of a negotiation process. The return value of those functions indicates if the component has accepted or not the new guidelines. Since the AMSs are self-governing, the return value allows the DP to identify that an AMS has rejected the order of the OP.
- **removeGoal()/removePolicy()**: Interface used to remove goals and policies.
- **getPolicies()** and **getSLAs()** – Allows the OP to retrieve the current policies or SLAs, respectively, which are being used by an AMS or Behaviour. This will be used mostly for AMSs obtaining the goals and policies of the AMSs, since those are self-governing.
- **subscribe(Parameter, Threshold):** This call allows the DP to perform simple monitoring (threshold-based) on the Behaviours and AMSs, allowing it to identify changes in the condition of the network which may trigger an orchestration event (federation, negotiation, closing down of an AMS, etc...) in case the threshold crossing puts in danger the high-level goals of the DOCs.
- **notify(Parameter):** This function is used to notify the DP of a parameter change.

Note that this does not mean that those are the only possible calls that the DP will perform on certain Behaviours. Specific workflows may incorporate specialised interfaces (i.e. in a policy negotiation workflow the DP may call a **negotiate()** function of a certain type of AMS). The following sections describe the operation, functions and policies of the core behaviours, namely the distribution, negotiation, governance, and federation core behaviours.

4.2. Distribution Behaviour

The distribution behaviour provides communication and control services that enable management tasks to be split into parts that run on multiple AMSs within an Orchestration Plane. The distribution behaviour, thus, controls the deployment of AMSs and their components. Using orchestration policies and based on user requirements, this behaviour will the policies and SLAs driving each of the deployed components, as well as which virtual resources they will control (in the case of the deployment of multiple components). It is up to the SP and the VP to define on which virtual resources the AMSs should be deployed.

Further, the distribution behaviour only specifies the characteristics of AMS deployment. It is up to the Service enablers Plane (SP) to perform the deployment. As such, the distribution behaviour uses the service deployment interfaces, defined in D5.2. In order to completely define the needs of a deployment, the distribution behaviour defines the information that is needed for the AMSs, and on which set of resources the AMS must be deployed. The deployment of customer-facing services (i.e. a web server, a streaming media application) is delegated to the AMSs.

4.2.1. Operation of the Distribution Behaviour

Besides the basic functions of a basic Behaviour, the distribution behaviour adds the following functions to its interfaces:

- **derivePolicies(SLAs, UserRequirements)**: Based on the SLAs and user

requirements, this function identifies the orchestration policies that will be necessary to this particular deployment. Those will be forwarded to the SP, which will then provide them to the newly deployed AMSs.

- **deriveListOfAMSs()**: This functions defines which AMSs must be deployed to govern the resources defined by the SP. The Behaviour then calls the SP, which returns a list of AMSs that are able to provide the requested features. If the SP does not find any AMS fulfilling the requested characteristics, the distribution Behaviour requests a renegotiation to the DP.
- **startAMSDeployment(Services, Nodes, OrchestrationPolicies)**: This function is called by the distribution behaviour to notify the SP that a certain AMS must be deployed. This is called once for each of the needed AMSs. The AMSs, then, will deploy the user facing services (i.e. a web server, a streaming server) required for the entire deployment task.

The distribution process begins with one AMS or the operator indicating to the DP that a new service or AMS must be deployed. The DP then starts up the distribution Behaviour, indicating the high level goals that must be respected during this specific deployment. The behaviour, in turn, starts a new deployment iteration, deriving the set of policies that will be used as well as the AMSs that should be deployed. Although this behaviour defines the AMSs to be deployed, it does not interfere in the deployment of user-facing services (a web or streaming server, for example), since the objective of the OP is to orchestrate the AMSs. Further, the AMSs and the SP will define the best way to deploy them. Once the policies and the AMSs for the new distribution are defined, the distribution behaviour triggers the SP plane, which is the entity in charge of actually deploying the AutoI components. The DOC will define the characteristics of the services and AMSs to be deployed, such as the required QoS of the service, the type of the service and some high level parameters. The SP, in turn, will indicate to the Behaviour if it is possible to deploy an AMS that fulfils the QoS restrictions demanded by the DOC. If the service cannot be deployed, the Behaviour signals the DP that the deployment is not possible. As a consequence, the DP triggers a re-negotiation of the distribution parameters. After the parameters are re-negotiated, the distribution retries the deployment by notifying the SP the new parameters of the components that must be deployed.

After the SP performs the deployment, the AMSs must notify the DOC if the policies and SLAs that have been provided are acceptable. Those are allowed to reject then and, in this case, it is up to the DP to decide if the distribution must be cancelled, or if a new attempt must be performed. In case a distribution action is cancelled, a new distribution decision may require the use of a negotiation behaviour, which we describe in a separate section of this document. Once the new distribution decision is taken, the DP deletes the components deployed in this previous iteration, and then retries the new deployment. This time, a new set of policies and requirements, defined either by a negotiation process or explicitly after an operator's request, is used in the process.

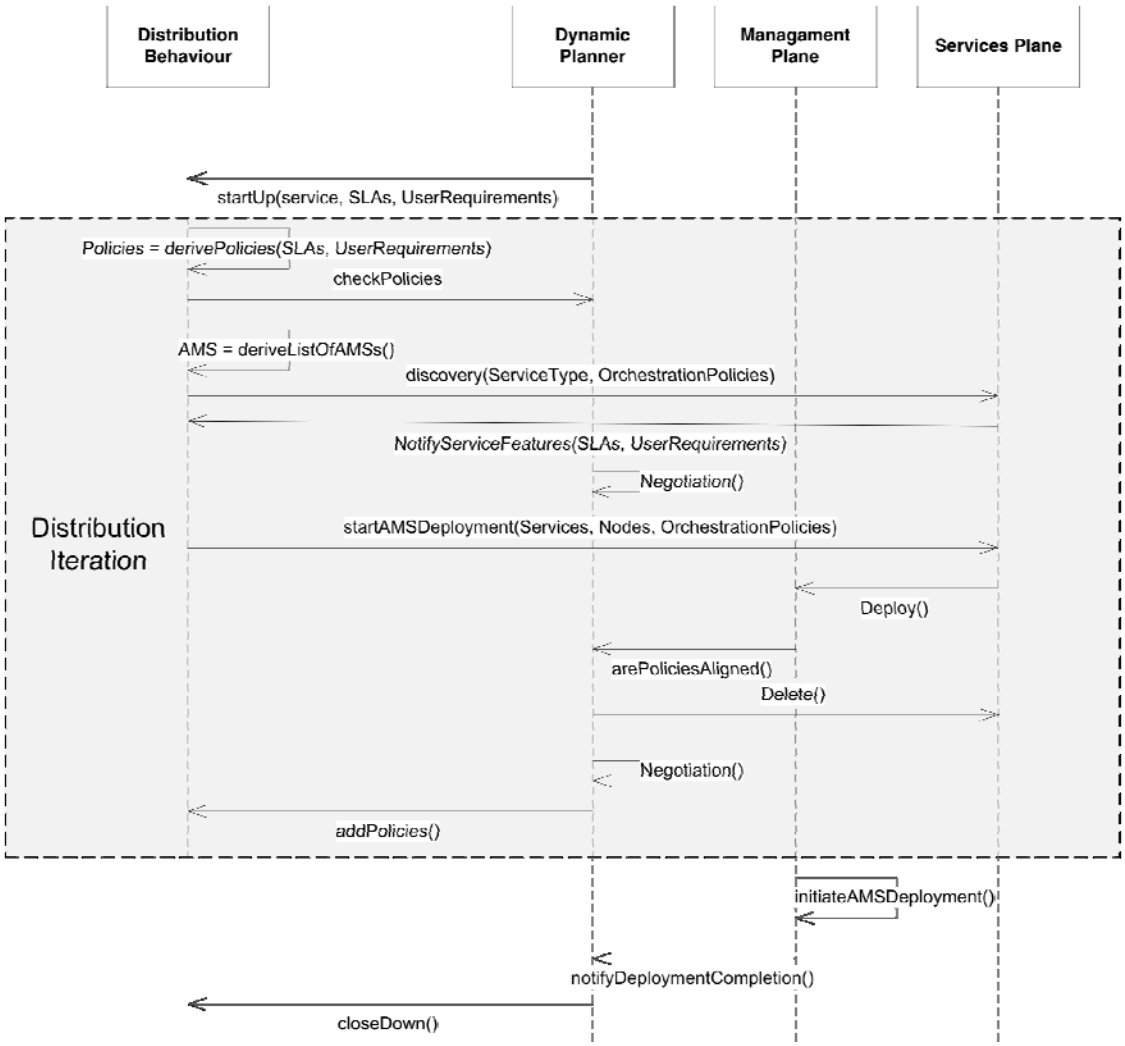

Figure 3 Sequence diagram of the distribution behaviour.

4.2.2. Distribution Policies

Policies in the distribution behaviour define the components to be deployed for a given AMS, defining the capabilities that those components must provide. Further, distribution policies define the QoS constraints and SLAs of the AMSs. Some of those parameters may be negotiable, while others may not be. This information is also coded as policies.

Capabilities policies: Those policies describe, in a somewhat abstract way, the capabilities that the deployed AMSs must have. They may define the type of autonomic management that it should perform (i.e. self-configuration, self-optimization), as well as the service that it will manage (multimedia, Web, telephony). Those requirements will usually be forwarded to the ANPI interface, so the SP will return a list of the registered AMSs that may provide the requested capabilities. Note that, due to the self-governance of the AMS, the distribution behaviour only defines in a high level the type of service that must be deployed. It is up to the AMS to specify the version, protocols and components that must be deployed to realize the service.

QoS and SLA-related policies: Those policies will guide the AMSs in the deployment of the specific components required for the service being distributed. The QoS and SLA policies are derived from the high level goals defined by the operators of the orchestration

domains and of the federation that the DOC belongs to. The policies define ranges of values for different QoS parameters, and if those parameters are mutable or not. This allows the AMSs to redefine the QoS constraints of specific components of the service as well as the entire service. Other parameters cannot be changed, since values outside this range would go against the high level goals of the orchestration domain. If any of those static QoS guarantees cannot be provided by the AMS being deployed, the AMS warns the DOC. The DOC then starts up the negotiation behaviour in order to derive a new set of high-level goals.

4.2.3. Stability and Scalability

Scalability: since the distribution behaviour is not in charge of the deployment *per se*, instead defining the parameters of the deployment, its scalability will depend on the amount of high-level goals and policies that must be taken into consideration for each new AMS deployment. For example, the number of QoS and SLA constraints that must be satisfied by the new deployment will increase with the number of AMS domains under the orchestration domain. Thus, the distribution behaviour is in charge of finding viable SLAs and policies ensuring that the service fits the requirements of each of the different operators in the orchestration domain.
Stability: the deployment of new AMSs depends on the existence of a valid set of configuration parameters, which respect the overall goals of the orchestration domain, as well as the goals and SLAs of each managed domain. In case such configuration exists, the deployment is straightforward. Otherwise, the distribution behaviour relies on the negotiation behaviour, which orchestrates the negotiation between the different domains in order to reach a compromise in the configuration of the AMS to be deployed.
Further, the scalability and stability of the distribution behaviour also relies on the amount of changes in the policies of each of the domains. If changes are frequent, the distribution behaviour will be more necessary in order to carry out AMS migration as well as to deploy new AMSs to replace older AMSs. Since the AMSs are self-governing, DOCs have limited power over too frequent configuration changes. However, the AMSs may be shut down if the governance behaviour (described in Section 4.4) detects that the AMSs are violating the overall orchestration policies defined for the orchestration domain.

4.3. Negotiation Behaviour

In Future Internet Networks (FINs) approaches [1][2][3][4][5][6][7], it is mandatory for network and service providers to offer and publish their services so that more complex services can be provided. For flexibility purposes the creation and establishment of complex services must be done in an autonomic manner. The deployment of complex services usually involves the collaboration of multiple entities having different capabilities, controlling the resources under its domain, and in summary, providing a number of network and application services. An important component that allows this requirement to be met in AutoI is the negotiation component of the Orchestration Plane. This component acts as a virtual service broker that mediates between different AMSs, taking care of service requests and providing support so that the underlying service providers can negotiate SLAs, responsibilities, tasks, high-level goals, etc. In order to support such functionality, the negotiation behaviour takes into account the nature of the underlying service providers, the services they provide, their interests, their service qualities and other key aspects. All in all, this functionality should be fully automated with well-defined negotiation-oriented

decision-making processes.
In AutoI each AMS advertises a set of capabilities (i.e., services and/or resources) that it offers for use in the Orchestration Plane. The negotiation functionality enables AMSs to reach agreements and to form SLAs for selected, well-described services. DOCs mediate during the negotiation process acting as trusted third parties or service brokers for the AMSs since the DOCs have the advantage of a more holistic view of the network. Examples include using a particular capability from a range of capabilities (e.g., a particular encryption strength when multiple strengths are offered), being granted exclusive use of a particular service when multiple AMSs are competing for the sole use of that service, and agreeing on a particular protocol, resource, and/or service to use.

The negotiation functionality orchestrated between AMSs and DOCs has inherent business and technical concerns. The following elaborates on these two critical aspects.

4.3.1. Business and Technical Concerns of the Negotiation Behaviour

When the AMSs negotiate high-level goals under a DOC, or when different DOCs negotiate high-level goals with other DOCs, successful negotiation finishes with participants aligning their internal business objectives. An agreement is reached when the participants converge to negotiated high-level goals for which virtual and non-virtual resources must be allocated, managed and controlled autonomously in their respective domains (AMS and/or DOC). During negotiation procedures, the involved participants also consider the responsibilities, benefits and penalties.
The negotiation of business objectives is influenced by technical concerns in the sense that AMSs compromise resources to fulfil the negotiated high-level goals. The governance capability of an AMS (and that of a DOC) defines with whom, why, and when these actors negotiate.
As AMSs and DOCs can have active agreements with several parties, there is a potential need for AMSs and DOCs to implement re-negotiation capabilities in response to statistical changes in the resources committed to these goals when they cannot be fulfilled, or due to some internal decisions. The DOCs may also trigger re-negotiation when the common business objectives are not fulfilled. Re-negotiation can be driven by utility functions, cost/benefit optimisations, etc. Again, during the re-negotiation of high-level goals, the responsibilities, benefits and penalties are also considered.
The negotiation behaviour provides the means for the mediation between AMSs and DOCs so that they can negotiate high-level goals as a result of a complex service request. As the AMSs and DOCs may belong to different administrative domains, they may exchange information using different languages, they may express their high-level goals in different terms and so forth. Ontology translation and mapping techniques can be used to create the common language with which participating entities can negotiate, federate, etc. The Orchestration Plane must provide the mechanisms for the negotiation to occur regardless of any of these technical aspects.
Another important technical concern of the OP negotiation functionality is that of convergence to optimal solutions. By definition, negotiation is needed when there is a complex service that demands the participation of more than one service provider. In multi-service provider environments, there may be several providers offering the same service with different qualities and can be in the middle of competitive environments. Under these circumstances of dynamic negotiation, a service request may involve several negotiation runs, and stability and convergence to optimal solutions are aspects that must be addressed by the DOCs during the negotiation activity.

4.3.2. Architecture of the Negotiation Behaviour

The operation of the negotiation behaviour mainly involves DOCs and AMSs. It spans mainly across both the orchestration plane and the management plane in the AutoI architecture, with indirect interactions with the knowledge, virtualization and service enabler's plane. Figure 4 shows its basic conceptual view in the AutoI framework.

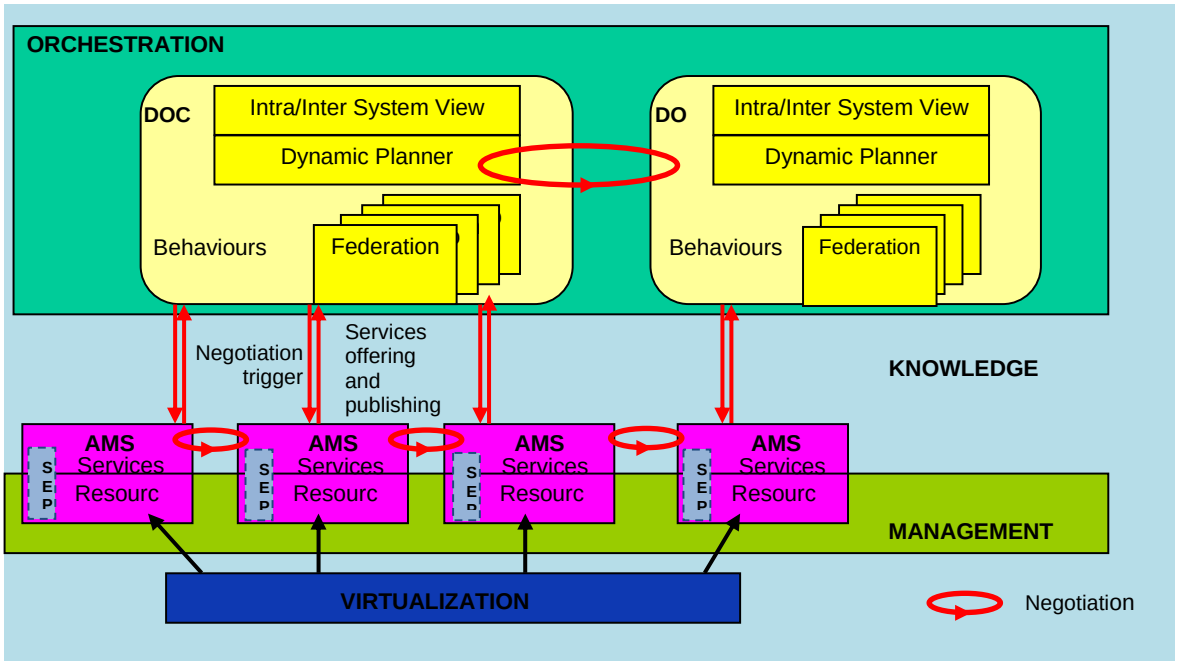

Figure 4: Conceptual view of the strategic-negotiation framework

The OP draws information on the system view from the knowledge plane. The OP works along with the management plane to complete the operation of the negotiation behaviour. AMSs are aware of their available resources through the knowledge plane. The services running on the resources managed by an AMS are in turn the responsibility of such AMS. These services are subject of negotiation. The AMSs offer and publish the services they are willing to compromise for the provision of complex services to the DOC(s).
Negotiation-wise, the service enablers plane assists the management plane in the discovery and programmability of the services residing in the virtual resources under responsibility of each AMS. The service enablers plane keeps an updated status of the services in the knowledge plane, and supports the deployment of both complex services (those orchestrated by the OP, resulting from negotiation and federation functions), and AMS services. During the negotiation phase (before a complex service is deployed), the relevance of the knowledge plane relies on provision of accurate metrics of the capacities of resources, and the availability and quality of the services that an AMS can potentially advertise for negotiation purposes. This is a joint responsibility with the virtualization plane, the management plane and the service enablers plane, which are responsible to update the above sensible information.
The DOCs trigger the negotiation process to the underlying AMSs. The following is the

general negotiation operation.

4.3.3. Operation of the Negotiation Behaviour

The negotiation behaviour is the component responsible in the OP architecture for establishing collaboration amongst AMSs and conflict resolution. A collaboration may be required when an AMS does not posses the required capability or resources for the establishment of some services. In this case, the negotiation behaviour will be invoked to enable this particular AMS to find a willing AMS that has the missing capability and then establish a collaboration agreement with it. Additionally, in the OP, a conflict may consist of AMSs having an incompatible set of policies, SLAs or high-level goals, which hinder the seamless interoperation of both. Also, since AMSs are self-governing entities, a negotiation may arise when the AMSs change their policies or SLAs autonomously, into creating an incompatibility with the overall orchestration policies and SLAs.
The negotiation behaviour does not define by itself how a conflict may be solved. Instead, it acts as a mediator for the parties involved in a conflict. Those parties may be AMSs or the DP itself. Hence, the negotiation behaviour controls the workflow of the negotiation process. First, it may define a time limit and/or number of negotiation rounds (or iterations) for a negotiation. In each of those rounds, the conflicting parties compromise, proposing a new set of system level policies and SLAs. Second, this behaviour checks if the proposed compromise is compatible with the orchestration policies.
In order to describe the general operation of the negotiation operation, consider a set of AMS modelled by a set $A = \{A_1, A_2 \ldots, A_N\}$ where N is the total number of AMS attached to a specific DOC, D. A negotiation behaviour is required when a new service request is received, when a distribution of tasks between several AMSs is needed, or when a reconfiguration for optimisation purposes is called for. In general terms a negotiation process is started by a requesting entity R. This requesting entity can be either an AMS that cannot satisfy a service request by itself, a DOC that needs to figure out which of the underlying AMSs can provide a service or both, when a reconfiguration of a service is needed, either for optimisation or due to any of the self-* management activities in the AutoI framework. It should be noted that R may not be a member of set A if the negotiation is an inter-DOC negotiation.
The service request can generically be modelled as a set $T = \{(T_1,\sigma_1), (T_2,\sigma_2), , \ldots, (T_M,\sigma_M)\}$where T is the type of service required, σ is a numerical quantification of the quality of the requested service T, and M is the total number of the type of services required. Depending on the approach used, the request may contain additional information such as the effective period of the request. If a negotiation of the service parameters is needed, the OP will base on the request received to identify the specific AMSs for negotiation and then initiates the negotiation behaviour. The behaviour first selects the AMSs that will participate in the negotiation. AMSs that do not have (i.e. advertise) the required services will be eliminated from the selection. The remaining candidates participate in the negotiation, and are chosen based on different criteria. The simplest one is a random selection where a candidate is randomly selected to take part in the negotiation. A more sensible method is a greedy selection criterion whereby all the possible candidates are ranked in decreasing order based on the type and quality of services available. In this selection approach the top ranked candidate(s) are selected to take part in the negotiation process.
There may be cases where a completed negotiation is revised due to detected conflicts. The OP may also parameterise the negotiation (e.g. numerical classification and/or range of services, time frame etc) based on the nature or objective of the scenario. However, it should be stressed that the OP does not negotiate on behalf of the AMSs but only

facilitates the negotiation process. It acts as a mediator and the negotiation behaviour itself is the vessel to achieve harmonized negotiation.
For the OP to function seamlessly, AMSs are required to provide adequate and up-to-date information with regards to their state to the DOC they are attached to. As such, each AMS shall periodically advertise information on its current services offerings (or services they are willing to share / negotiate) so that *D* always has enough information to select the candidates for negotiation. The advertisement sent by A_n at k^{th} period is denoted by $Ad_n(k)$. Once identified, *D* will inform the selected AMS(s) with a start message. The negotiation process starts after all involved parties receive *S*. It is assumed that the agreed terms from the negotiation must be autonomously enforced via local actions under each corresponding AMS.
The negotiation Behaviour employs its basic interfaces and the representative methods listed below in order to implement negotiation:

- **setOtherPolicies(SetOfPoliciesList)**: This function, implemented by the Behaviours which may need to negotiate (i.e. two or more AMSs), allows the negotiation behaviour to pass to each of the interested parties the system level policies coming from the other parties involved in the negotiation. This function has as parameters several sets of system level policies. Each set is composed of the system level policies being used for one of the negotiating parties. It is expected that, after a certain time period, the parties will call the function **checkNegotiationResult()** to indicate a new set of policies from their part, which will drive the next negotiation round, if necessary.
- **triggerNegotiation()**: This function triggers the negotiation phase on the AMSs. It indicates that all parties are ready for starting the negotiation process.
- **checkNegotiationResult()**: This function, called by the negotiating parties, passes the new set of system level policies of each of the entities for the negotiation behaviour, so that those can be analysed and verified if they are consistent with the orchestration policies. After all the involved entities have called this function, then the behaviour enters an analysis phase.
- **analyseProposedPolicies():** This function starts the process of policy analysis. The objective of this process is to check if the system level policies of each of the negotiating entities are compatible among each other, and if those are aligned with the orchestration policies. Although the AMSs are aware of the orchestration policies, they may intentionally define a conflicting set of system level policies due to their self-governing characteristics.
- **abortNegotiation()**: This function is used by the negotiation behaviour to notify all the entities that the negotiation has been aborted. This may happen due to several reasons: *(i)* the dynamic planner has aborted the operation; *(ii)* the negotiation behaviour detects that one of the negotiating parties is not willing to compromise; *(iii)* the negotiation process has achieved a certain limit of iterations or a time limit; *(iv)* the negotiation result does not satisfy the requirements of the orchestration objectives. If any of these situations happens, the negotiation behaviour will contact the Dynamic Planner to inform its failure.
- **negotiationFailed()**: Informs the DP that the negotiation process has failed. Following this notification, the DP may trigger a new negotiation, redeploy the behaviours or AMSs that are not willing to compromise, or simply inform the operator that the new high level orchestration objectives are not achievable in the orchestrated domain.
- **checkPolicies():** Used by the negotiation behaviour to check if the system level policies of an AMS are aligned with the overall orchestration policies.

Figure 5 shows the operation of the negotiation behaviour as a finite state diagram (FSM).

In essence, this behaviour operates in clear states, or negotiation iterations. In each iteration the behaviour contacts the negotiating entities, sending them the system level policies and SLAs that have been proposed by the other conflicting entities. Each of those entities must produce a new set of policies that they are willing to deploy as their new configuration. It is up to those entities to propose a configuration. During this processes, those entities may communicate among them if needed, using a set of specific protocols. Once they reach a decision, the new set of system level policies is forwarded to the negotiation behaviour. This step is needed because the proposed policies may not be aligned to the overall orchestration policies of the OP. In this case, the negotiation behaviour rejects the proposed policies, and starts a new negotiation round.
To avoid an endless negotiation, the negotiation behaviour should implement algorithms to detect if one or more entities are willing to compromise, or if convergence to optimal results is not feasible in a reasonable time limit. For example, if one AMS always returns the same set of system level policies, this may indicate that it is not willing to change its configuration for the sake of the others. The negotiation may also be limited in the number of possible iterations, the amount of time spent on iterations, and an amount of time spent negotiating. If the process fails, the negotiation behaviour communicates this to the DP. The DP may, in turn, propose a set of measures to solve the conflict: *(i)* shut down some of the components involved in the negotiation and start another version of those components; *(ii)* change the orchestration policies or *(iii)* declare failure, leaving the configuration in its last stable state, and notifying the operator of this failure.

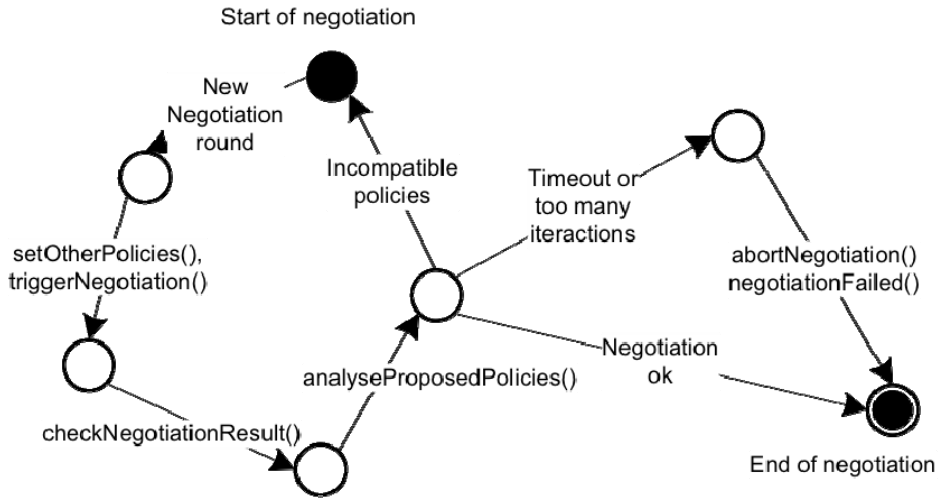

Figure 5 FSM representation of the negotiation behaviour.

Essentially, the negotiation problem is a distributed rational decision making problem. In AutoI, game theory is applied to solve this problem. Two approaches within the game theory are advocated in this project, namely a bargaining and a coalition formation approach, which are detailed separately in the next sections.

4.3.4. Bargaining-based Negotiation Approach

In this approach, negotiation parties attempt to reach a mutually beneficial (win-win) solution in a conflict of interest scenario. There are two major branches under this theory [8].

- The *Axiomatic Bargaining Theory* – it is a cooperative bargaining theory whereby

desirable properties for a solution, called axioms of the bargaining solution are postulated and then the solution that satisfies these axioms is sought. Nash bargaining solution is one such solution.

- The *Strategic Negotiation Theory* – this theory usually analyses sequential bargaining where negotiation parties alternate in making offers to each other.

The *strategic-negotiation model* based on Rubinstein's model of alternating offers [9] is advocated. The model consists of three main components: the negotiation protocol, the utility functions of the negotiation parties (in our case, the negotiation parties are AMSs) and the parties' negotiation strategies. The negotiation protocol defines how the involved parties interact among themselves. It specifies what and when each party must respond. The negotiation ends when all involved parties agree to an offer or one of them decided to quit from the process without agreement. Each negotiating party has its own utility function that evaluates different possible terminations of the negotiation. The utility functions are based on various factors such as the costs (e.g. fixed and variable cost), high-level objectives (e.g. maximise revenue or maximise utilisation etc.). Finally, the parties' negotiation strategy specifies the parties' next move in various situations.

The framework operates under several assumptions. Firstly, it is assumed that a negotiation party is not informed of other parties' responses when making his own reply. This ensures that no party may gain unfair advantage during the negotiation process. Also, all parties are assumed to be rational. This implies that the involved parties will attempt to maximise their utilities and behave consistently according to their preferences. To direct the negotiation towards common agreement, the involved parties avoid quitting or opting out of the negotiation process. By this, effectively, the model requires that a party accepts the offer if the utility derived from accepting the offer and opting out are the same. As briefly mentioned above, the involved parties are also expected to honour the agreements (if reached). However, there are no long-term commitments i.e. each negotiation is independent, thus eradicating the case where an involved party may consider any prospective future activity amongst the other parties. Finally, all parties are assumed to know and follow the above assumptions.

We need to consider the problem of how a rational party will choose its negotiation strategy. Usually the Nash Equilibrium is used [10]. However, Nash equilibrium is inadequate as it may be unstable in intermediate stages of the negotiation process and may in any case be far from optimal. Moreover, this approach ignores the whole process of offer and counter-offer that characterises bargaining and the possibility of breakdown. For our case, we use the concept of Sub-game Perfect Equilibrium (SPE) [11]. A strategy is a SPE of a model for alternating offers if the strategy profile induced in every sub-game is a Nash Equilibrium of that sub-game. For incomplete information game, we use the concept of Sequential Equilibrium (SE), which takes into account the beliefs of the involved parties.

In general, negotiation is needed in our framework when an AMS lacks resources in order to achieve a goal (e.g. when it cannot provision a new service with the quality that is requested, when a service needs to be provisioned across domains, etc). This AMS has then the need to collaborate with another AMS. In our framework, it will send a request to the OP in order to find a candidate AMS to collaborate. Based on the received request, the OP will first identify the AMSs that have the requested resources. The remaining candidates can then be chosen based on different criteria. The simplest one is a random selection where a candidate is randomly selected to take part in the negotiation. A more sensible method is a greedy selection criterion whereby all the possible candidates are ranked in decreasing order based on the type and quality of services available. In this selection approach, the top ranked candidate is selected to take part in the negotiation process.

We consider a set of AMSs, $\mathbf{A} = \{A_1, A_2, \mathrm{K}, A_N\}$, governed by an OP. A negotiation

process is started by a requesting AMS by initiating a request.

Definition 1 – Request: A request is defined by a 5-tuple $REQ = \langle id,\ m,\ s,\ dl,\ d \rangle$ where id, m, s, dl and d are the unique request identification number, the type of service / resource required, the numerical quantification of the requested item, the time period for which this negotiation applies to and the numerical quantification of the benefit the AMSs receive per time period, respectively.

For the OP to function seamlessly, AMSs are required to provide up-to-date information with regards to their state. As such, each AMS shall periodically advertise information on its current service offerings (or services they are willing to share / negotiate) so that the OP always has enough information to select the candidates for negotiation. The advertisement sent by A_n where $n = \{1,2,\mathrm{K},N\}$ at t^{th} period is denoted by $Ad_n(t)$. Once identified, OP will inform the selected AMS(s) with a start message, $START = \langle A_x, A_y, REQ \rangle$ where A_x is the AMS which initiated the negotiation and A_y is the AMS chosen to negotiate with A_x. The negotiation process starts after all involved parties receive the $START$ message. It is assumed that the agreed terms from the negotiation must be autonomously enforced via local actions under each corresponding AMS. Figure 6 shows an example timeline of the negotiation operation involving two AMSs, where A_1 initiated a negotiation (i.e. $A_1 = A_x$) and A_2 is the selected AMS to negotiate by the OP (i.e. $A_2 = A_y$). For this example, an agreement is achieved after two iterations (exchanges of information). Once the terms have been agreed, both AMSs will enforce them until the expiry time (if such a term is included in the negotiation) or when either a request to terminate the agreement is received, or when the service for which the negotiation was triggered is deleted or migrated to other AMSs. In this case, another round of negotiation will be triggered. The agreement is sent to the OP in order to update its current overall view.

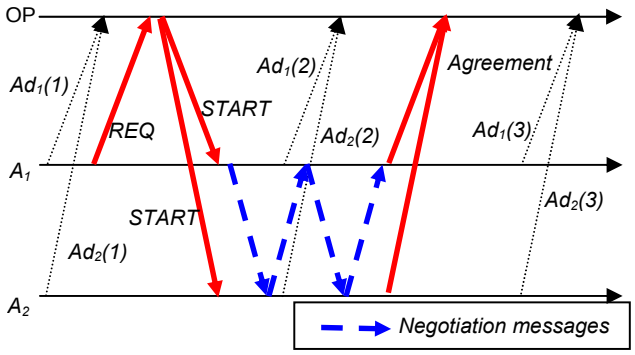

Figure 6 Timeline of OP arranging the negotiation of two agents.

The negotiation takes place between two AMSs. We denote the AMS which initiates the negotiation as R and the AMS chosen by the OP as C where $R,C \in \mathbf{A}$. Both parties negotiate over the share of a commonly desirable entity (e.g. revenue), which decays over time, i.e. $P(t \mid dl, s, d)$. The negotiation is modelled as a turn-based process that is taken over the time set $\mathrm{T} = \{0,1,2,\mathrm{K}\}$. The AMSs have the option of leaving the negotiation when it is their turn to respond. In practice, this option will be taken if the AMS realises that it cannot gain from this collaboration. For example, in the middle of a negotiation on the issue of sharing a batch of virtual machines, an AMS may decide not to negotiate further if it

realizes that if the negotiation continues, there is no possibility in gaining at least as much as what it would get if the resources under negotiation were used elsewhere. Hence, there is a threshold dictating the point where it prefers to reserve the resources rather than sharing them.

All AMSs are assumed to be rational. This implies that the AMSs attempt to maximise their utilities and behave consistently according to their preferences. To direct the negotiation towards an agreement, AMSs avoid quitting the negotiation process. By this, it effectively means that they accept an offer if the utility derived from accepting that offer and opting out are equivalent. The AMSs are also expected to honour the agreements (if reached). However, there are no long-term commitments i.e. each negotiation is independent, thus eradicating the case where an AMS may consider any prospective future activity amongst the AMSs. Finally, all AMSs are assumed to be aware of the above assumptions.

4.3.5. Coalition Formation-based Negotiation Approach

This negotiation approach advocate a framework with well-defined negotiation procedures that can provide support to systematically create service coalitions, linked to the nature of the populations of service providers like the AutoI AMSs. Contrary to the bargaining approach that targets a win-to-win environment of the involved negotiating parties, the coalition formation approach deals with the critical nature of proposing a negotiation protocol for self-interested service providers. The intention of this negotiation approach is to provide the required support to service providers (AMSs) so that they can develop coalition formation processes in competitive and collaborative environments [12]. A coalition framework based model, captures the fact that a group of entities are better off being together than on by their own. Moreover, this organizational paradigm implies that relationships established between coalitional partners are goal driven and short life [13]. This negotiation protocol enables the DOCs, or any requesting entity of a negotiation process, to work as a service broker that can virtually provide complex (i.e. composed) services relying on coalition formation processes between third-party service providers. The proposed protocol elevates the behaviour of the Orchestration Plane to a service marketplace that exploits the benefits of social networking principles in favour of manageability and scalability. Several aspects like scalability, coalition strategies of the service providers aligned to AMS self-governance policies, social network characteristics, heterogeneity of service requests, stability and convergence to optimal negotiation results are subject of analysis in this protocol. The overall picture of the service marketplace specialised by the Distributed Orchestration Component (DOC), which is the component that implements the OP concept, is depicted in Figure 7.

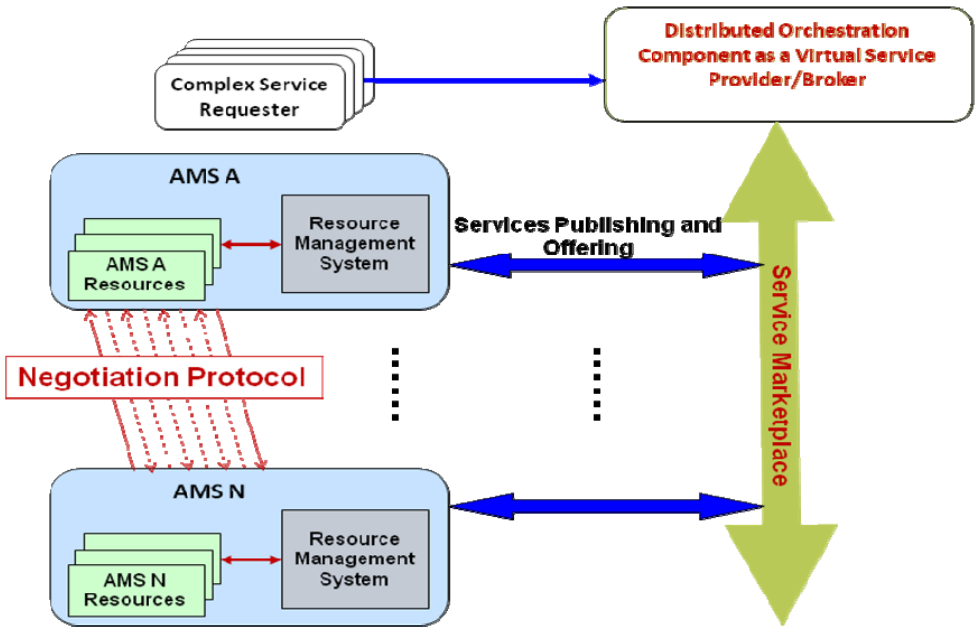

Figure 7 The Distributed Orchestration Component as a Service Marketplace.

The following are the general properties addressed by this protocol and its characteristics.
Skill grading properties: different service aggregates required in a complex service may have different degrees of importance for the overall performance of a requested service. This protocol addresses this requirement in that a required complex service request is identified with an array of service skill requirements – service skill refers to capabilities, functionalities, services or other support provided by an AMS –. For the negotiation parties (AMS service providers) the protocol considers performance levels of the service skill requirements. This approach allows; i) modelling the presence or absence of service skills to perform a certain action; ii) defining a degree of performance in case the provider can provide a service skill.
Dynamic service composition: several AMS service providers may be available at the same time, and they may also join and leave the service marketplace environment at any time. They may also change their offered services due to lack of resources, resource re-configuration or other reasons. This highlights the need for a dynamic mechanism that provides support to form services coalitions in the most optimal manner; i) keeping control of the requested services, the available service providers and their service offers at any time and ii) defining appropriate combinations of service providers meeting the requirements of the requested services.
The proposed protocol provides support for the above requirement as follows: every time a service is needed, a set of necessary service skills that satisfy the original service request is created. Service providers are characterized with a set of service skills, and it is potentially a norm that a single entity cannot fulfil a complex service. Providers are able to aggregate their service skills in order to create coalitions that fulfil the requirements of such complex service. The requirements of this service are communicated to the service providers. These providers form coalitions iteratively, autonomously and self-interestedly to satisfy the requirements of the service. During the above iterations, service providers decide the action to take towards coalition formation by using information on partial

coalition results. Service coalitions compete amongst them to be the option that best fits the demand until reaching a local-optimal state. At a certain point of the coalition formation process, the different combinations of provider entities are evaluated and the best service coalition(s) are selected.

Scalability: AMS service providers in this protocol have to take coalitional actions over time and hence evaluate the potential outcome in joining different coalitions. A large-scale system allowing service providers being aware of every other service provider in the system can become prohibitive, i.e. non-scalable. In order to ensure scalability, the protocol proposed here limits the service provider's sight to a fixed number of service providers driven by social network links. This approach relies on limiting the social awareness of providers, by letting them have a reduced and fixed set of connections that limits the coalitions each provider can evaluate at any time.

Distribution: The coalitional process is distributed as coalitional decisions are taken at individual service provider level. Providers are considered self-interested and their prime objective is to be in a competitive coalition in order to be elected in the negotiation. For this, they follow a negotiation strategy according to their self-governing policy. Negotiation strategies enable each provider to evaluate its contribution to other existing coalition and enforce the most appropriate negotiation action.

Transparency: the protocol supports transparency in that the coalition formation process is completely autonomous, and once the service requirements are acknowledged by the requester entity, no more interaction is needed with such entity. The AMS service providers are the actors involved actively in the negotiation process.

4.3.6. Negotiation Policies

The negotiation behaviour uses policies to represent the parameters of the negotiation process, as well as the ranges of AMS and service parameters being negotiated. Policies provide a higher amount of flexibility to the negotiation behaviour, allowing the AMSs as well as the DOCs to adapt the negotiation to each particular scenario.

Negotiable-set policies: those policies are produced by the AMSs as well as DOCs, and define two different sets of policies. One set defines the non-negotiable terms of the SLA, that is, the requirements that one of the AMSs or the DOC are not willing to compromise on. The second set defines the negotiable policies, which define the negotiable service parameters. In the coalition formation approach described earlier the result of the enforcement of these policies are the negotiable service parameters, namely the service skills which each AMS negotiates in the service marketplace.

Range negotiation policies: for the negotiable policies, AMSs and DOCs may define a set of ranges or allowed values. Those policies can be used to describe the capabilities of the AMSs and DOCs, as well as what resources and service levels each of the components are willing to provide. For example, in the coalition formation approach described earlier the enforcement of these policies result in specific performance levels for each service skill, with which each AMS negotiates in the service marketplace.

Orchestration-related policies: those policies act over the negotiation process, setting values for the parameters of the negotiation approaches. Such policies could define the maximum amount of negotiation iterations, the maximum time per iteration, among others. The most important use of those policies is to define limits on the negotiation process, i.e. to improve the scalability and stability of the OP. Those policies may also define goals that must be achieved at each negotiation round, and actions if those goals are not met. For example, an AMS should provide a different, more flexible set of constraints at each round. Another policy could state that, if an AMS is inflexible after a certain number of rounds, the negotiation process is stopped.

4.3.7. Stability and Scalability

Scalability: the scalability of the negotiation behaviour depends on the AMSs and DOCs. On the part of the DOCs, policies can define the amount of negotiation rounds, their duration and goals. Thus, the DOC may influence the time required to complete a negotiation attempt by defining an acceptable number of rounds as well as a set of goals for each round. The AMSs, on the other hand, can reduce the time and resources required for a negotiation by being more flexible and simplifying their requirements. A greater degree of flexibility could be obtained by a wider range of accepted parameter values, and by reducing the amount of non-negotiable parameters.
Stability: the most important actor in the stability of the negotiation behaviour is the AMS. Since the DOCs cannot force AMSs to accept a certain configuration, the AMSs must be willing to compromise. If most of the service parameters are non-negotiable or their value ranges are too strict, the negotiation process would probably take a larger number of negotiation rounds to succeed. Further, a too restrictive set of negotiable parameters may impede or even prevent the success of the negotiation.

A detailed analysis of the Orchestration Plane's negotiation behaviour, namely dealing with stability, scalability and convergence to optimal negotiation results is provided in Deliverable 6.3 "Final Results of the Autonomic Internet Approach" [24].

4.4. Governance Behaviour

In the AutoI architecture, AMSs are self-governing elements, having each their own control loops over the virtual resources under their management scope. Hence, they may decide to change their policies, SLAs and user requirements on the fly, without cooperation or interventions of the DOCs. The DOCs, being responsible to oversee the operation of the AMSs in order to allow a smooth intra-domain and inter-domain operation, must hence monitor the actions of the AMSs. The Governance Behaviour performs this monitoring. This behaviour ensures that the AMSs are aligned with the high-level orchestration goals set by each orchestration domain. In a nutshell, the functions of the governance behaviour are as follows:

- **Monitoring of the actions of the AMSs**: monitoring is performed to check if the actions of the AMSs are consistent with what the DOC has requested. Hence, the DOCs watch the configuration and policies of the AMSs, always verifying if this configuration is still aligned with the goals set at the orchestration level.
- **Enforcement of policies and SLAs defined by the DOCs:** the governance behaviour checks for misbehaviours, caused by AMSs changing their system level policies. Once they happen, the DOCs will take measures to indicate to the AMS of its non-compliance. Those requests take the form of function calls requiring the AMS to use a certain set of orchestration policies or a configuration that has previously been agreed.
- **Trigger for federation, negotiation and distribution tasks upon non-compliance**: Once the governance behaviour detects that an action from the part of an AMS that seems to be conflicting with the objectives of the orchestration plane, it will start up the proper counter-measures. This trigger occurs after the DOC tries to enforce a certain configuration over the AMSs, however the AMS rejects the proposed configuration. Hence, the DOC must find new ways to ensure the smooth operation of the network. Those may be achieved by the renegotiation

of the system level policies of one or more AMSs, the replacement of certain AMSs by another implementation, or the need to merge/split the network.

Each instance of the governance behaviour monitors one AMS. The governance behaviour assumes that the set of virtual resources managed by each AMS are non-overlapping. As a consequence, the responsible AMS will forcibly implement all self-* functions for its managed resources. This assumption does not imply that each resource will be used for only one service or network. A physical resource, being divided into several virtual resources, can be used for different services or networks. Each of those virtual resources, however, will be dedicated to a single service, and as such will be managed by only one AMS.

4.4.1. Operation of the Governance Behaviour

One instance of the governance behaviour supervises one AMS performing a certain task. Primarily, the negotiation behaviour communicates only with the DP and the AMS that it orchestrates. It uses monitoring and policy setting primitives with the AMS, while it communicates with the DP only to report violations that need further attention, requiring the triggering of one or more other Behaviours (distribution, federation, negotiation). The functions used are as follows. Figure 7 shows a sequence diagram of the lifecycle of the governance behaviour.

- **startUp(Policies, SLAs, ListOfAMS)**: Upon start up by the DP, the behaviour receives the set of high level orchestration policies as well as the SLAs that must be enforced into the list of AMSs. This function triggers internal processing on the behaviour that defines what are the parameters to be monitored.
- **deriveMonitoringParameters()**: This internal function defines the AMS parameters that must be monitored, as well as the set of allowed values. Based on this list, the governance behaviour will issue **subscribe()** calls to the AMSs. Those will allow the behaviour to receive notifications if the AMS changes its key parameters. Besides AMSs, the behaviour may decide to monitor services, links or routers if deemed necessary. Once this task is done, the behaviour enters a monitoring phase.
- **checkAlignment()**: Whenever any of the monitored resources or AMSs notify the behaviour of a relevant change in their state, the behaviour checks if this change is aligned wit the orchestration policies. If they are, then the behaviour does nothing. If not, the behaviour will make a certain amount of attempts to enforce the orchestration policies.
- **setGoal()/setPolicy()**: The governance behaviour makes use of the basic behaviour functions **setGoal()** and **setPolicy()** in order to enforce the orchestration policies. By calling those functions, the behaviour attempts to provoke an adaptation from the part of the AMS. If the AMS refuses to accept those goals, then the behaviour identifies a violation to the overall orchestration objectives.
  The functions **setGoal()** and **setPolicy()** can also be used by the DP to indicate to the governance behaviour that a new set of orchestration policies and SLAs must be enforced. Such a call from the DP may be the response of the DP for the violation of the orchestration goals an AMS. It may also indicate that the operator or another entity requested a change in the orchestration policies that must be enforced.
- **notifyViolation()**: This function is used by the governance behaviour to notify the DP of a violation of the orchestration objective. Upon receiving this function call, the DP must provide a solution to the violation. Either the DP alone solves the violation, or it requests for the help of other behaviours. For example, the DP may decide that

a change on its orchestration policies or the system level of other AMSs are needed, which requires the bootstrapping of one negotiation behaviour. In Figure 7, the resolution of a conflict of this kind is portrayed as the violation resolution phase, in order to simplify the diagram.

Hence, the operation of the governance behaviour can be roughly divided into three distinct phases. The start-up phase, where the behaviour receives the set of goals and policies that it must enforce, and subscribes to the correct events. The monitoring phase, which occurs when the negotiation behaviour watches for changes in the monitored AMSs and resources. Finally, the enforcement/adaptation phase is triggered when a violation occurs. It involves the detection of a violation and the application of a certain action to correct the violation.

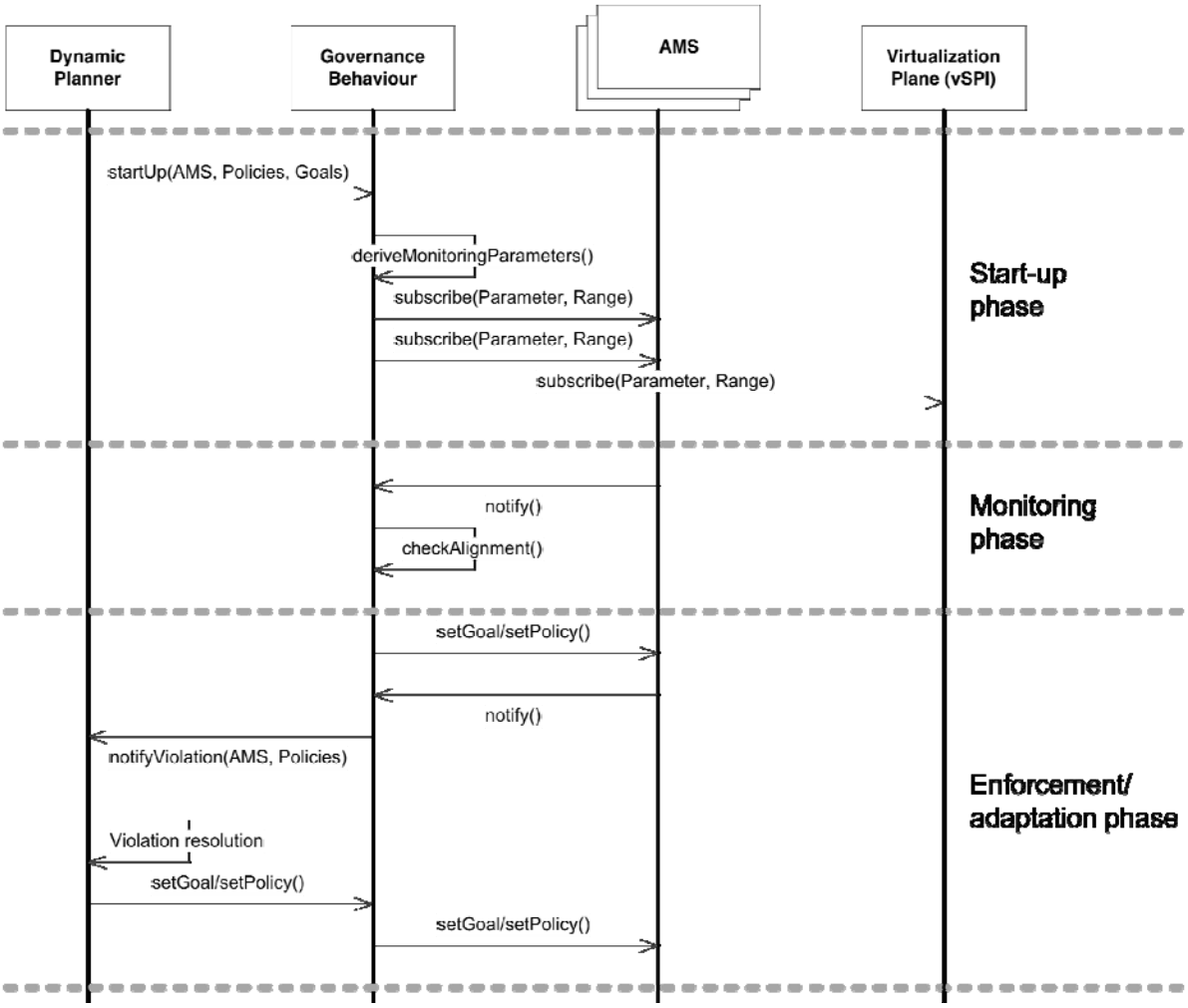

Figure 7 Sequence diagram of the governance behaviour.

4.4.2. Governance Policies

Governance policies define a set of operational parameters for the DOCs, as well as the actions that should be taken for certain violations. Thus, governance policies define the monitoring patterns that the governance behaviour must enforce. Different kinds of parameter violations will trigger different reactions. As a result, certain conditions may

trigger the redefinition of policies, while other conditions may require the redeployment of the AMS. Governance behaviours are derived from the orchestration policies, high-level goals and user requirements of the services.
Due to the characteristics defined above, governance policies resemble events, as they are defined in the following form: *if one of a set of parameters lies outside the defined ranges, then do the specified actions*. The monitored parameters may comprise properties of the AMS, user-facing services or virtual links. For example, if a certain service creates a significant degradation in the QoS, the governance behaviour may decide to renegotiate the SLAs of the self-configuration AMS managing this service. The actions that are defined in the policies may involve the following operations: triggering a negotiation among AMSs, sending a message for the AMS that it is not compliant to the set of defined orchestration goals, or shutting down the AMS and creating another one.
Since the governance behaviour monitors only one AMS, there are no "default governance policies", as in the dynamic planner policies. The DOC, however, may install the same set of standard governance policies in all governance behaviours.

4.4.3. Stability and Scalability

Since one instance of the governance behaviour monitors only one AMS, the scalability of the Behaviour will depend on the complexity of the monitoring capabilities that it implements. As a consequence, the scalability of the governance behaviour is mainly dictated by the amount of parameters that are monitored at each AMS. The governance behaviour will use the events and other facilities of the KP to receive notifications only when certain thresholds are passed in the monitored parameters. Hence, in our point of view, the governance behaviour will scale quite easily.
The stability of the governance behaviour will depend on the governance policies employed. The main concern for the stability is conflicts among the general governance policies, which define the overall operation of all the AMSs, and the governance policies specific to each class or implementation of AMSs. Thus, standard governance policies should override specific governance policies when there is overlap, in order to guarantee that all the AMSs abide to the same general rules. Thus, we assume that the governance behaviour would internally parse the policies to check for conflicts.

4.5. Federation Behaviour

The federation behaviour of the AutoI orchestration plane is responsible for the federation of networks under different orchestration domains. Since each domain may have different SLAs and policies, a federation attempt may trigger a negotiation for new policies, or even the re-deployment of services in the case that the new policies and high level goals of the network are not compatible with some of the deployed services.

4.5.1. Operation of the Federation Behaviour

A federation in AutoI is born when two or more AMSs from different orchestration domains need to come together under a common objective. The Federation Behaviour follows a set of steps to verify whether the two domains could be combined and how this task could be accomplished:
Notification for a federation request: this task is triggered when a new service is created, requiring two domains to come together, allowing the service to be accomplished in an end-to-end manner. In this step, the Dynamic Planner of domain A launches its Federation

Behaviour. The latter sends a **Federation Request message** to the Dynamic Planner of domain B containing the policies over which domain A desires to federate with the domain B. This set contains negotiable as well as non-negotiable policies. When the Dynamic Planner of domain B receives the Federation Request, the negotiation step of the Federation begins.
Policy Negotiation: the second step of Federation consists of verifying if the AMSs of domain B can accept policies proposed by domain A. If the AMSs of domain A agree on those policies, the **Federation Confirm** step begins. If not, the Dynamic Planner of domain B launches the **Negotiation Behaviour** in order to find an agreement. This is based on the policies proposed by domain A. This set is comprised of negotiable and non-negotiable policies. If the negotiation is rejected for any reason, the **Federation Reject** step is done.
Federation Confirm: If the negotiation Behaviour confirms that the negotiation can take place, the Dynamic Planner of Domain B sends a Federation Confirm message to the Federation Behaviour of domain A. Next, after receiving this confirmation, the latter confirms the federation, and its turn indicates that the Federation begins effectively.
Federation Reject: If the negotiation Behaviour rejects the negotiation, the Dynamic Planner of Domain B sends a Reject message to Federation Behaviour of Domain A.
Federation Beginning: After receiving the Federation message, the Federation Behaviour begins effectively the Federation.
These steps are performed over some primitives as follows:

- **FederationRequest(Orchestration Policies - negotiable policies & non-negotiable policies):** This function is called when an orchestration domain A want to federate to an orchestration domain B. It communicates its orchestration policies to domain B in order to let it decide if the federation is aligned to its high level goals.
- **AMSFederationRequest(Management policies):** This function is called when a DOC receives a Federation request and it attempts to ask AMSs if they would agree with a federation regarding their policies. So, the DOC of the domain should translate the orchestration policies of the other domains into system level policies in order to see if a federation could take place and if it would be aligned to their own policies.
- **AMSFederationConfirm():** This message is sent by AMSs in order to inform the DOCs about their agreement to federate. After receiving the AMS Federation Confirm of all its AMSs, the corresponding DOC analyses the possibility of federation.
- **AMSFederationReject():** Relating to the previous function, this message is also sent by AMSs in order to inform their DOC that they cannot federate given the proposed orchestration policies. A DOC attempts a negotiation process after receiving an AMS Federation reject if the threshold of federation attempts is not exceeded.
- **FederationReject():** This function is called when a DOC identifies the impossibility of federation. It sends a Federation reject to the other DOCs in order to inform it that they cannot federate.
- **FederationEstablish():** When a federation agreement is achieved, the DOC initiates the Federation by sending a Federation Establish message.

Figure 8 shows a simple case where federation is attempted. The DP of domain A sends a Federation Request to the DP of domain B, which informs its corresponding AMSs about the request given the orchestration policies of domain A. If all AMS of domain A accept to federate given the policies proposed by domain A, they send an AMS Federation Confirm

to their corresponding DOC. When that is confirmed, the DOC of domain A sends a Federation Establish to domain B and the federation begins.
Figure 9illustrates the case where a negotiation is needed to compromise policies. It shows the process taken when the maximum number of federation attempts is achieved and a domain concludes that the federation is not possible.

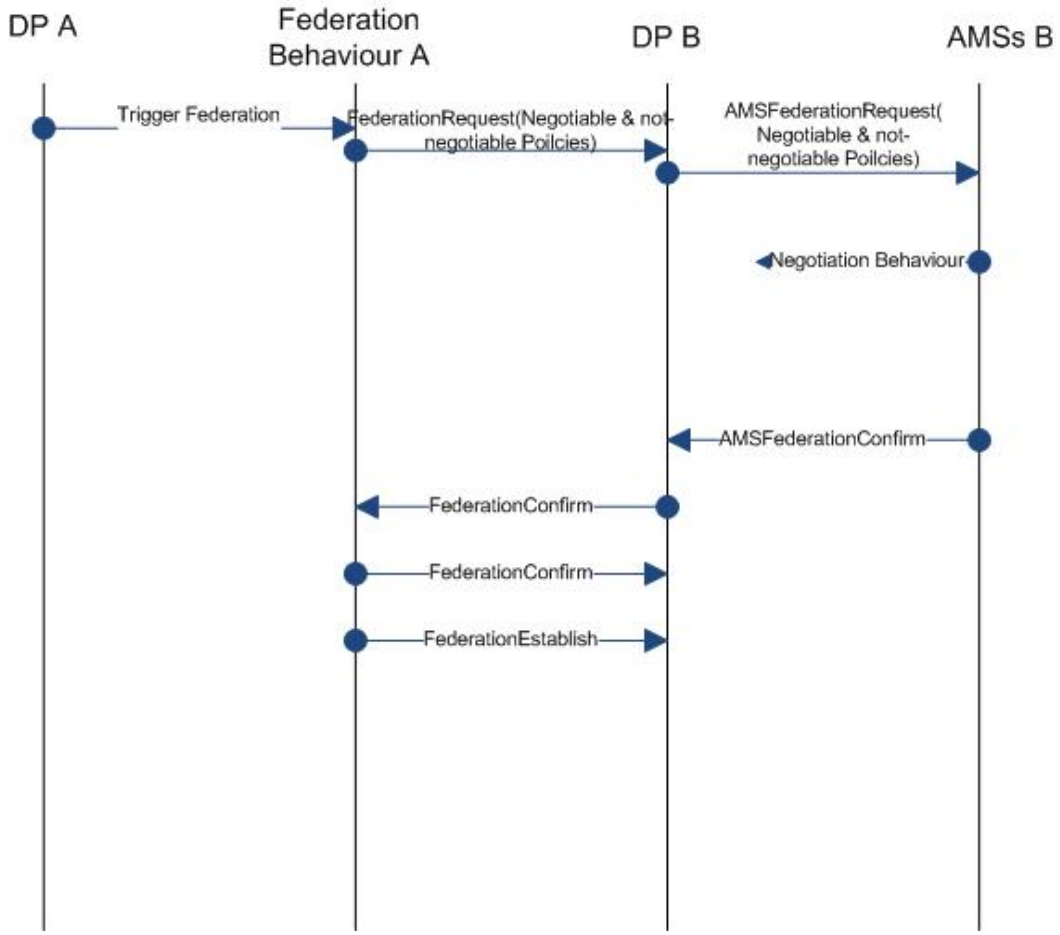

Figure 8 Operation of the federation behaviour for a failed federation.

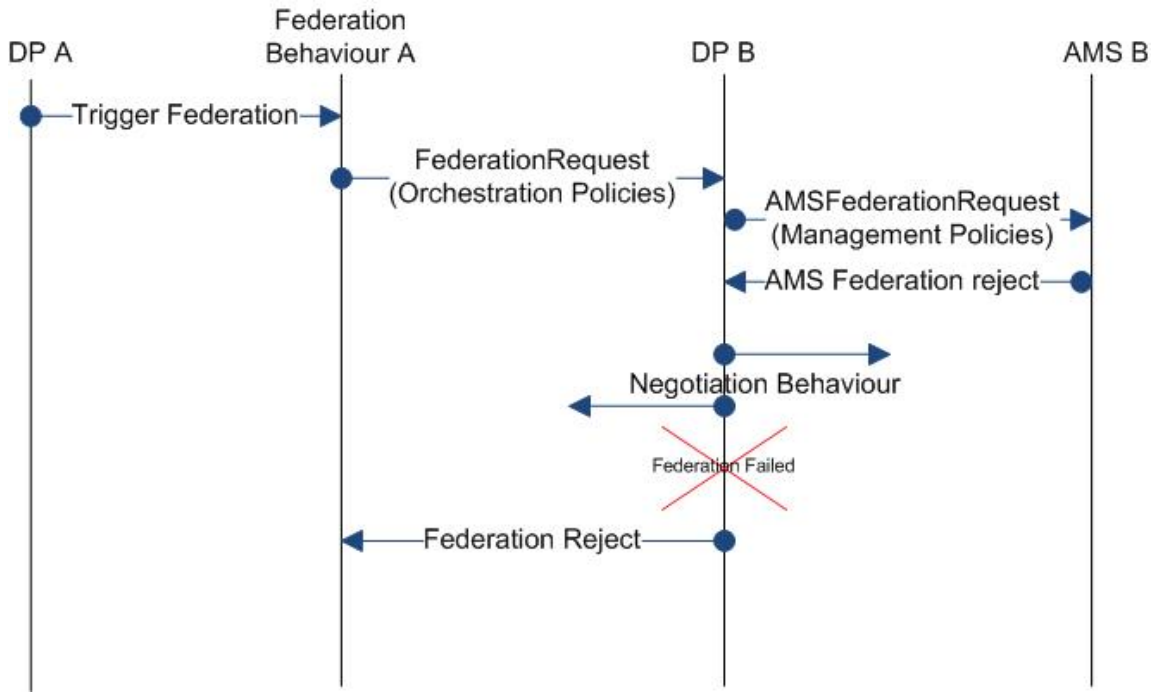

Figure 9 Federation Behaviour - federation failure.

4.5.2. Federation Policies

Policies in the federation Behaviour define the AMSs that are involved in the federation and also policies to communicate and negotiate with federation partners for the purpose of federation:

List of Involved AMSs: a part of policies in the federation Behaviour are responsible for defining which AMSs should be involved in the federation process, and then how to communicate and negotiate with them. The list of involved AMSs is based on federation purpose and the specific requirements of what has triggered the federation. This is an important type of policies in federation, as it avoids overloading network with unnecessary control traffic from non-concerned AMSs.

Negotiable and non-negotiable policies: this part of federation Behaviour policies defines the applicable policies in the federation process, outlining what are hard and non-negotiable from the point of view of orchestration, and policies which could be negotiated regarding the agreement of the involved AMSs. Based on these policies, self-governed AMSs check if the high-level federation policies are not in contradiction with their own non-negotiable policies. If there is a conflict, the AMS rejects the federation, as there is no possibility of negotiation. In contrast, if there is a disagreement in negotiable policies, the federation process continues after one or more negotiation rounds. If, after a limited number of negotiation attempts, DOCs and AMSs find an agreement on a given set of policies, the federation continues. If not, it will be rejected.

4.5.3. Stability and Scalability

The scalability of the federation Behaviour is influenced by the scalability of the negotiation Behaviour and the number of involved AMSs. As the AutoI architecture foresees a set of distributed DOC components, each DOC takes a limited number of AMSs in charge. So,

the negotiation Behaviour should deal with a limited number of AMSs and scales well.
In the federation Behaviour, a counter is defined in order to limit the number of negotiation attempts to conclude an agreement. If within this counter times, the negotiation succeeds, the federation goes on. If not, it will be rejected.
The Federation behaviour also defines policies that detect if the state has changed (for example, an AMSs cannot participate anymore in the federation or the AMSs' policies changed, etc) and renegotiate in order to define, if possible, a new agreement.

5. Orchestration Plane Interfaces

This section describes the interactions of the DOC with other elements of the AutoI architecture, as shown in Figure 10. This text refines the description of the OP interfaces found in the deliverable D2.1 [19]. We first describe the interface of the DOCs with the Knowledge Plane. The interfaces with virtual resources, the AMSs and the service enablers plane are just introduced here, since they will be defined in the deliverables respective to each of those functions.

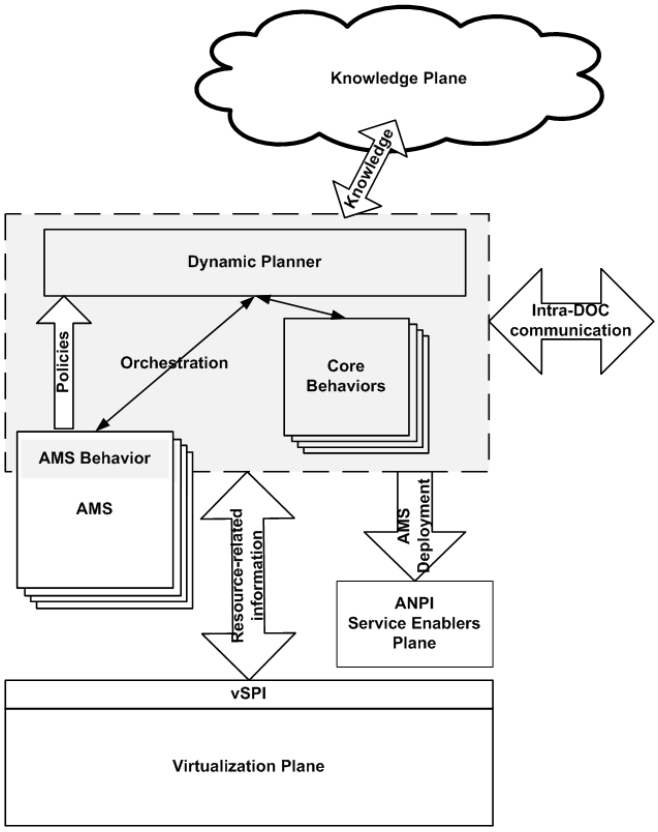

Figure 10 Interfaces of the DOC

As described before, the ultimate task of the orchestration plane is to deploy AMSs within the network. The deployment of AMSs should be done considering the management requirements of services. Consequently, the orchestration plane performs this task of AMSs deployment, using the Service Enabler Plane (SP). However, the OP uses the SP to deploy AMSs, it does not deal with the deployment of services. The SP interface is defined in the deliverable D5.2 [22].

In one hand, the orchestration plane needs to interact with virtual resources in order to provide the OP with physical management and control information regarding the state of the network and resources, which will be used for deploying AMSs Behaviours. This interface is useful since it provides the necessary information helping to deploy AMSs in the adequate places regarding the state of the network and resources. The interaction of the DOC with the virtualization plane will be described in the deliverable D1.2 [21].

The OP/AMS interface: the OP/AMS interface is defined in the deliverable D4.2 [20]. DOCs use this interface to set and the orchestration-related goals, policies and SLAs to be

used by the AMSs. Further, during a negotiation, the DOCs use this interface to obtain the system level policies and SLAs of the AMSs, in order to orchestrate the negotiation among two or more AMSs. Finally, this interface allows the DOC to notify the AMS of important orchestration events, such as the federation of two domains/networks, the creation of new AMSs, and to assign virtual resources to a certain AMS in the event of a service migration.
Apart from the shutdown and deploy commands, most of the interactions of the DOC with the AMS are in the sense of informing the AMS of events in the orchestration domain. This is the case because the AMSs are self-governing, and as such the AMS may or may not accept a new goal or SLA proposed by the DOCs. If the AMSs do not accept the new configuration, it is up to the DOC to propose an alternative configuration or to propose a negotiation among the AMSs.
The OP/KP interface: the OP uses the Knowledge Plane (KP) to obtain relevant orchestration information from the domains and AMSs (the situated view of the DOCs, as described in Deliverable D4.1 [16]). DOCs also employ this interface to monitor the AMSs. Since this interface is described in details in the deliverable D2.1 [19], this deliverable simply shows an overview of the interface.
The OP/KP interface is event-based, however it also provides a lookup primitive for synchronous access. The clients of the KP use a **subscribe** call to watch events, while the **watch** command performs range-based monitoring in certain information. It is also possible to create events, via the **push** command.
In essence, the OP subscribes to important AMS events, and watches certain parameters of the AMSs for changes using a range-based mechanism. This allows for a better scalability and lower resource usage, since the DOCs do not need to periodically re-evaluate the information stored in the KP. When changes are detected, the KP notifies the DOCs, which take the appropriate actions. All the communication among DOCs is performed using the KP as well.
The OP/SP interface: the ANPI (Autonomic Networking Programming Interface) interface allows the management of network services over a virtualised autonomic network. This interface allows the creation, destruction, migration, pause, and restart virtual services, which is performed by the SP.
This interface is used by the DOC to deploy new AMSs, as described in Section 4.2. Essentially, the DOCs create, destroy and migrate AMSs. The best deployment of the AMSs (their location, configuration parameters) is left to the ANPI and to the AMSs. The DOC may also use the ANPI to obtain a list of services that can be deployed by the SP, in order to select which components it will request to achieve the creation of a new AMS.
The OP/VP interface: the OP uses the vSPI (Virtualisation System Programmability Interface) to obtain information related to the virtual resources. This information may be needed in the deployment of new AMSs, as well as in the negotiation Behaviour to solve conflicts.
This interface is mostly associated with the AMSs, which use it to control and program virtual resources. Since the DOC is not responsible of managing the AMSs, it limits its usage of the vSPI to information requests. Those requests allow the DOC to check if a certain virtual network elements (NEs) can be used in the deployment of a new service or not, how many and where the virtual NEs are deployed. Similarly, the governance Behaviour may use the vSPI to monitor the operation of the AMSs, checking if the performance of some of the NEs is within the boundaries of the contracted SLAs.

6. Conclusions

The AutoI project proposes a new system or plane, the Orchestration Plane (OP), which enables the cooperation of the various autonomic control loops in the network, ensuring that they operate within the boundaries set by the business goals defined by the operators. This document presented a refined architecture of the orchestration plane, improving the definitions presented in the deliverable D2.1. We provided an improved description of the orchestration plane and its responsibilities, as well as its components.

Orchestration arises from the need to coordinate the operation of two or more autonomic control loops, which may reside at a single domain or at different domains. In the future Internet, each domain will be run by different autonomic management systems. Although management will be decentralised, a set of common inter-domain constraints and SLAs must be agreed upon in order to provide assurable QoS for end-to-end services. In the AutoI architecture, we added a new functional plane, the Orchestration Plane, which is realized by a set of distributed components.

The Orchestration Plane is realised by the Distributed Orchestration Components (DOCs), which have two main components. The Dynamic Planner, which is a policy-based dispatcher of orchestration tasks, and the Behaviours, which model each specific orchestration task. This division allows for the extensibility of the architecture, since the planner is a generic component, taking care of the interaction of specialised components, the Behaviours. Behaviours are controlled by workflows, which are a set of steps and rules that define the interaction among behaviours, their triggering and shutting down conditions.

Behaviours in the OP implement a specialised management task. Those tasks can be grouped into the following general groups: *federation* behaviours are responsible for the federation of domains; *negotiation* behaviours define SLAs and policies that will be used for service provisioning and the management of virtual network elements; *distribution* behaviours coordinate the deployment of new AMSs and their components; *self-governance* behaviours monitor the AMSs under the supervision of a DOC, ensuring that they behave according to the high-level orchestration goals; *knowledge update* behaviours ensure that each component of the OP as well as the AMSs have their input information, setting the correct update frequency for the information.

Due to the novelty of the orchestration concept, several aspects of orchestration must be improved. Orchestration is highly dependent on the identification and resolution of conflicts among sets of policies. Thus, algorithms that infer the conflicting policies and to propose a set of non-conflicting rules must be devised. Also, the monitoring of virtual and autonomic systems is a recent topic, and as such more research is needed to devise a set of important parameters that will most likely be used for their orchestration. As well, orchestration requires mechanisms for more cognitive control of its resources, i.e. system data collection, management and decision making, which enable the Internet infrastructure to learn about its own behaviour, to tune its operation, and to enforce its decision on data manageability [25][26]. This implies using some artificial intelligence techniques which provide the orchestration entities to enhance their operation taking into account experiences from previous context and tries .

7. References

[1] Bassi, A., Denazis, S., Galis, A., Fahy, C., Serrano, M., Serrat, J., "Autonomic Internet: A Perspective for Future Internet Services Based on Autonomic Principles" - 2nd IEEE International Workshop on Modelling Autonomic Communications Environments (MACE), 2007.
[2] Future Internet Design (FIND) Program - http://www.nets-find.net/.
[3] Global Environment for Network Innovation (GENI) Program - http://www.geni.net/
[4] Future Internet Assembly (FIA)/ FIRE program – http://cordis.europa.eu/fp7/ict/fire/home_en.html.
[5] Ambient Networks (ANs) Project, http://www.ambient-networks.org.
[6] Mobile Ad-hoc NETworks (MANETs), http://www.ietf.org/html.charters/manet-charter.html.
[7] Rubio-Loyola, J. and Serrat, J. and Astorga, A. and Fischer, A. and Berl, A. and de Meer, H. and Koumoutsos, G., "A Viewpoint of the Network Management Paradigm for Future Internet Networks", in the Proceedings of the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM) 2009 - 1st IFIP/IEEE International Workshop on Management of the Future Internet (ManFI), 2009.
[8] Kraus, S. "Strategic negotiation in multiagent environments", The MIT Press, 2001.
[9] Rubinstein, A. "Perfect equilibrium in a bargaining model", Econometrica 50(1):97-109, 1982.
[10] Nash, J. F. "Two-person cooperative games", Econometrica 21:128-140.
[11] Osborne, M. J. and Rubinstein, A. "A Course in Game Theory", The MIT Press, 1994.
[12] Rubio-Loyola, J. and Mérida-Campos, C. and Willmott, S. and Astorga, A. and Serrat, J. and Galis, A., "Service Coalitions for Future Internet Services", Published in the Proceedings of the 45th IEEE International Conference on Communications (ICC) 2009.
[13] Horling, B., Lesser, V. "A Survey of Multi-Agent Organizational Paradigms". The Knowledge Engineering Review 19(4):281–316, 2005.
[14] Fischer, A. and Berl, A. (editors), Autonomic Internet Project, Deliverable D1.1 – Initial Virtual Plane Design, 2009.
[15] Boudjemil, Z. (editor), Autonomic Internet Project, Deliverable D3.1 – AutoI Information Model, 2009.
[16] Galis, A. and Mamatas, L. (editors), Autonomic Internet Project, Deliverable D4.1 – Initial Management Plane Design, 2009.
[17] Cheniour, A. and Lefevre L. (editors), Autonomic Internet Project, Deliverable D5.1 – Initial Service Enablers Plane Design, 2009.
[18] Galis, A. (editor), Autonomic Internet Project, Deliverable D6.1 – Initial AutoI Framework, 2008.
[19] Pujolle, G. (editor), Autonomic Internet Project, Deliverable D2.1 – Initial Orchestration Plane Design, 2009.
[20] Galis, A. and Mamatas L. (editors), Autonomic Internet Project, Deliverable D4.2 – Management and Knowledge Plane Interfaces, 2009.
[21] Fischer, A. and Berl, A. (editors), Autonomic Internet Project, Deliverable D1.2 – Virtualisation Plane and Interfaces, 2009.
[22] Cheniour, A. and Lefevre, L. (editors), Autonomic Internet Project, Deliverable D5.2 – Service Plane and Interfaces, 2009.
[23] Ganek, A. G. and Corbi, T. A., "The dawning of the autonomic computing era", IBM Systems Journal 42(1):5-18, 2004.

[24] EU IST Autonomic Internet Project, "Final Results Of The Autonomic Internet Approach" http://ist-autoi.eu
[25] MANA Position paper – May 2009 - http://www.future-internet.eu/home/future-internet-assembly/prague-may-2009/management-and-service-aware-networking-architectures.html
[26] FIA Stockholm Orchestration Session – November 2009 http://www.future-internet.eu/home/future-internet-assembly/stockholm-november-2009/cross-topic-sessions.html - c204