Deliverable 8.3: Open-source CASCADAS toolkit

Status and Version:	Final	
Date of issue:	03.07.2007	
Distribution:	Public	
Author(s):	Name	Partner
	Antonio Manzalini, Rosario Alfano, Antonietta Mannella	TI
	Sandra Haseloff, Rico Kusber	Unikassel
	Franco Zambonelli,Marco Mamei	UNIMORE
	Peter H. Deussen	FOKUS
	Fabrice Saffre	BT
	Elisabetta Di Nitto, Rafaela Mirandola, Daniel Dubois	DEI
	Ricardo Lent, Erol Gelenbe, Antonio Di Ferdinando	ICL
	Roberto Cascella, Roberto Battiti	UNITN
Checked by:	Antonio Manzalini	Telecom Italia

Abstract

Complexity of modern networks and service solutions raises several challenges in the design and development of communication and content services. The unbearable costs in configuration and management call for autonomic approaches, in which services are able to self-configure and self-adapt their activities without (or with a limited) human intervention. The need for ubiquity of service provisioning calls for the capability of services of adapting their behaviour depending on the current situation in which they are used.

In a previous Deliverable, D8.2, the need for innovative approaches facilitating the development and execution of autonomic and situation-aware services have been deeply discussed and analysed. CASCADAS project has the main goal of identifying and developing an innovative component-ware architecture for the development of innovative situation aware and autonomic services.
The Autonomic Communication Element (ACE) is the basic component abstraction over which the CASCADAS vision is built. The project development activities aims at prototyping a toolkit based on distributed ACES characterised by autonomic features (self-configuration, self-optimization, self-healing, self-protection, etc). Services will be created and executed (in a distributed way) by the self-aggregation of ACEs. This architecture will be demonstrated in application use-case by means of a tool-kit developed by the Consortium.

This deliverable has two main objectives: the former is to provide a preliminary description of the first release of the toolkit, a first step towards the open source Cascadas toolkit; the latter is to report a preliminary description of the roadmap leading to an integrated fully-fledged released of the toolkit. Moreover the deliverable will describe (in synthesis) the application use-case that will be adopted to demonstrate the features of the toolkit

Table of Contents

1 Document Overview

During the CASCADAS build-up phase (M1-M18), WP1 has carried out the design and development of a first release of the toolkit based on distributed components, ACE, as a basic run-time environment (e.g. a distributed s/w infrastructure to support ACE life-cycle and interactions). During the same period, the other WPs (WP2 for supervision, WP3 for self-organization, WP4 for security, WP5 for knowledge networks) have developed libraries of tools and proof-of-concept elements as a collection of (documented) software modules, suitable to be used by third parties and ready to be ported (and then integrated) into the run-time environment (developed by WP1). Then full integration phase of the toolkit will take place during M18-M24.

This deliverable has two main objectives: the former is to provide a preliminary description of the first release of the toolkit, a first step towards the open source Cascadas toolkit; the latter is to report a preliminary description of the roadmap leading to an integrated fully-fledged released of the toolkit.

Specifically, this first version of the ACE-based toolkit includes basic tools for creating and deploying ACEs; this is being used by the other WPs to integrate and release specific libraries devoted to implement self-aggregation algorithms, knowledge network services, pervasive supervision services, and security services.

At the end of the second year of the project all these implemented tools will form the first release of the open source Cascadas integrated toolkit. The effectiveness of the toolkit will be demonstrated through the implementation of an application demonstrator according to the application-oriented approach adopted by our project. Therefore this deliverable will also describe the application use-case selected to demonstrate how the toolkit fulfils the Cascadas vision.

2 Basic principles of the toolkit

CASCADAS project has the main goal of identifying and developing an innovative component-ware architecture for the development of innovative situation aware and autonomic services.

The Autonomic Communication Element (ACE) is the basic component abstraction over which the CASCADAS vision is built. The project development activities aims at prototyping a toolkit based on distributed ACES characterised by autonomic features (self-configuration, self-optimization, self-healing, self-protection, etc). Services will be created and executed (in a distributed way) by the self-aggregation of ACEs.

ACEs, the central components in CASCADAS are partially expected to be light weighted. The envisaged environment will contain a variety of ACEs, whose autonomic behaviour will not be realised by large and computational expensive subsystems but will emerge as an effect of ACE aggregation and cooperation amongst different system components. Events are the sole means for communication among ACEs and between ACE subcomponents.

The toolkit will provide the basic interfaces and technique aimed to support all the key aspects of Cascadas project .Generally speaking a mechanism (i.e., Functionality Repository) is enabled so that specific functionalities get *deployed* into the ACE instance and *get accessed* via ACE events. In this way all the specific algorithms elaborated by all the WPs are part of Specific Part Functionalities managed through the Functionality Repository. Only for the pervasive supervision in the toolkit is settled a specific inner feature as the pervasive supervision deals with the creation of dynamic, online feedback and control loops over an ensemble of ACEs. Such control loops will be a fundamental mechanism to manage ACEs activities in a decentralized and autonomic way. In the first release of the toolkit, functionality hooks and interfaces are available to monitor and supervise the Aces behaviour by establishing a contractual connection between the component under supervision and the supervision system expected from WP2. This basically means that the toolkit is able to send/receive events by the supervised ACE sub-components to/from the supervision system where they are manage properly.

3 Demonstration Use case

The general scenario selected for demonstration considers a modern exhibition center, such as for instance a big museum or a stadium. We consider the presence in the exhibition center of a number of *advertising screens*, which can be used to display information about the exhibition itself as well as commercial advertisements. As of today, such screens display information cyclically, in a way independent of the situation (i.e., independent of who is actually close to that screen)In such type of contexts, it is realistic to foresee the presence of a pervasive infrastructure of embedded devices such as sensors of various types, WiFi connections, RFID tags and other location systems. Likewise, we consider realistic to assume that people in the crowd carry mobile devices, such as PDAs, smart phones and tablet PCs, which can be equipped with an RFID-based system to the extent of providing a sort of "ticketing system" capable of providing information. The typical information stored, that would aim to guarantee for instance that the user has paid the entrance fee and possibly other peculiar fees, would serve to store other information about the user itself. A "smart" service devoted to decide what information to display could exploit the availability of contextual information to adaptively decide what information to show on advertising screen the basis of the people around and of their activities and interests. This would increase the value of the displayed advertisement both for users and for advertising companies.

The demonstrator that will be put in place will consider (see Figure 1):

- One real screen devoted to actually display commercial in an adaptive way, depending on the surrounding users, properly controlled by an associated PC
- Real users, each with an RFID-equipped device, that can get close to that screen and whose profiles can be tracked and analyzed anonymously for the sake of evaluating what advertisements to show. An RFID reader on the screen PC will make it possible to access RFID data. In addition, data on users' mobile devices can be accessed by the PC via Bluetooth as well as via WiFi.
- The execution of distributed auctions to sell the screen time slots. These auctions will take place by executing a number of ACEs on the CASCADAS distributed testbed, which will be connected via VPN to the local PC.

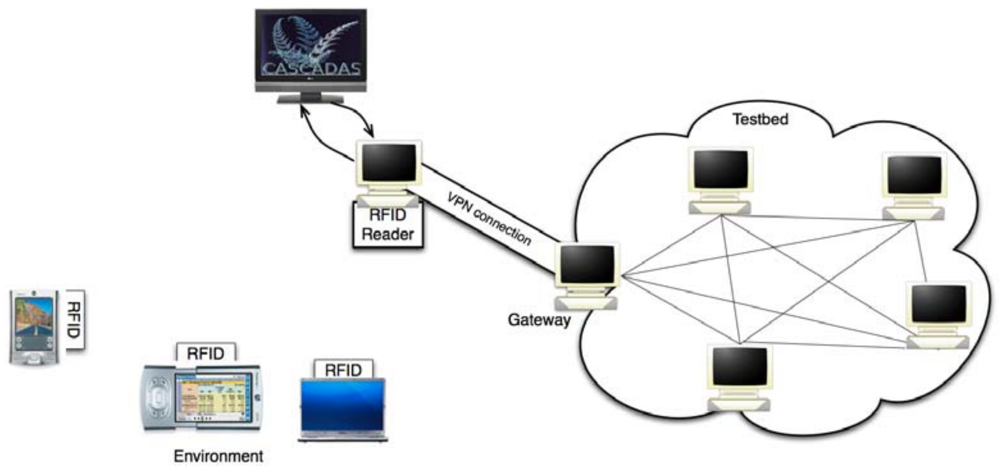

Figure 1 : Demonstrator scenario hardware configuration

The software architecture of the system employed in this scenario will be fully based on the ACE toolkit. Up-to-date information whether local or remote, to any requesting ACE will be provided by the Knowledge Network allocated on the screen PC and implemented on the ACE toolkit as well.

An ACE will be associated to the screen PC and will act as a "seller" for the time slots of the screen. Such ACE can advertise the auction in a (possibly remote) auction center, again realized via ACE. A variety of other ACEs will be instantiated on the remote CASCADAS testbed to act as bidders and to participate in auctions related to the selling of the screen time slots.

Pervasive Advertisement
The architecture for dealing with the specific part on pervasive advertisement considers the following.

- ***RFID tickets***: will include information, such as user ID and user interest, this latter in the simple and anonymous (*age range, subject of interest)* form. This data will be transmitted to the screen PC via the RFID and, translated into the XML format for Knowledge Atoms, will feed the knowledge network (see Figure 2).

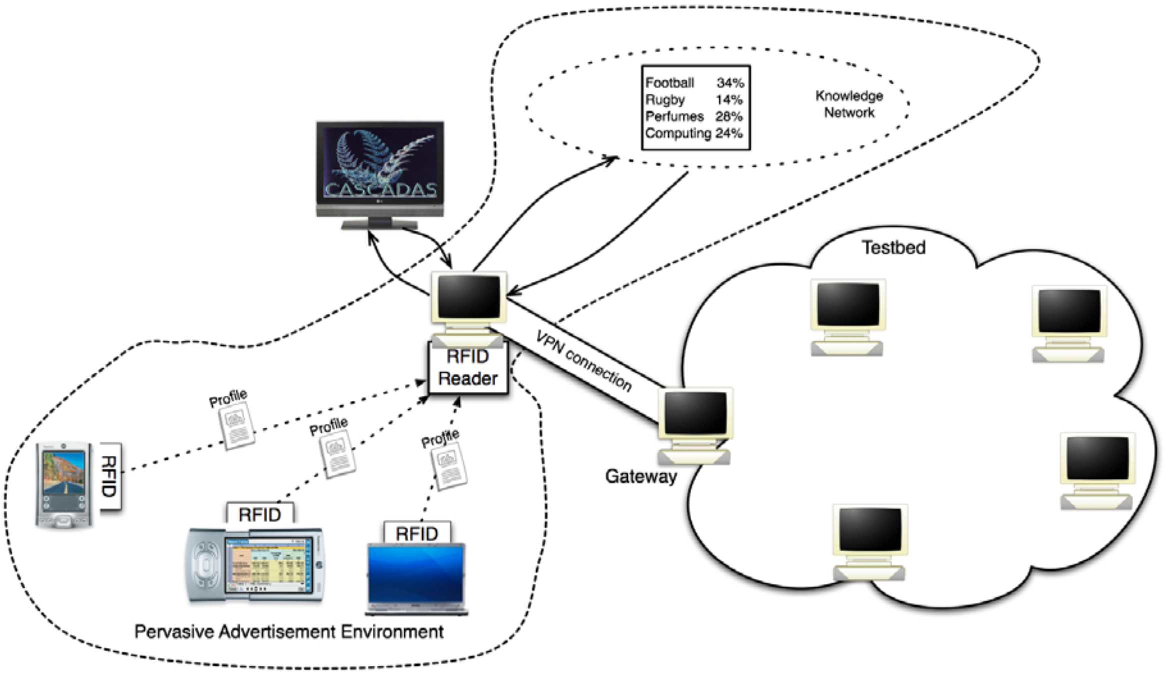

Figure 2 Acquisition and processing of profiles in the knowledge network

- ***Knowledge Network:*** Information mentioned above will be gathered by the screen PC, and will enter a knowledge network allocated on the PC itself. At any given time, the knowledge network will contain one knowledge atom for any user currently in range screen. Information contained in the knowledge network can be made available to any requesting ACE. Also, please note that access to individual knowledge atoms is always made possible for any ACE, thanks to the specific mechanisms integrated in ACE toolkit.

3.1.2 Advertisement Auction System

In the scenario, sequential time slots become available over time, each of them suitable for a particular type of audience. Such time slots become valuable to sellers interested in advertising their products and therefore, time slots are sold through an auction system. As a general rule, each time slot will need to be sold within stringent time constraints (e.g., while another advertisement is being displayed), so that real-time communications and high standard QoS networks become critical for the operation of the system.

Although, time slots could be sold ahead in time in a real system possibly allowing the creation of a complex market for such slots (e.g. reselling), we will concentrate on a single-slot selling to simplify the demonstrator. Therefore, the seller will always advertise the auction for the slot of time starting after the current one, and therefore auction duration will not have to exceed the duration of a slot.

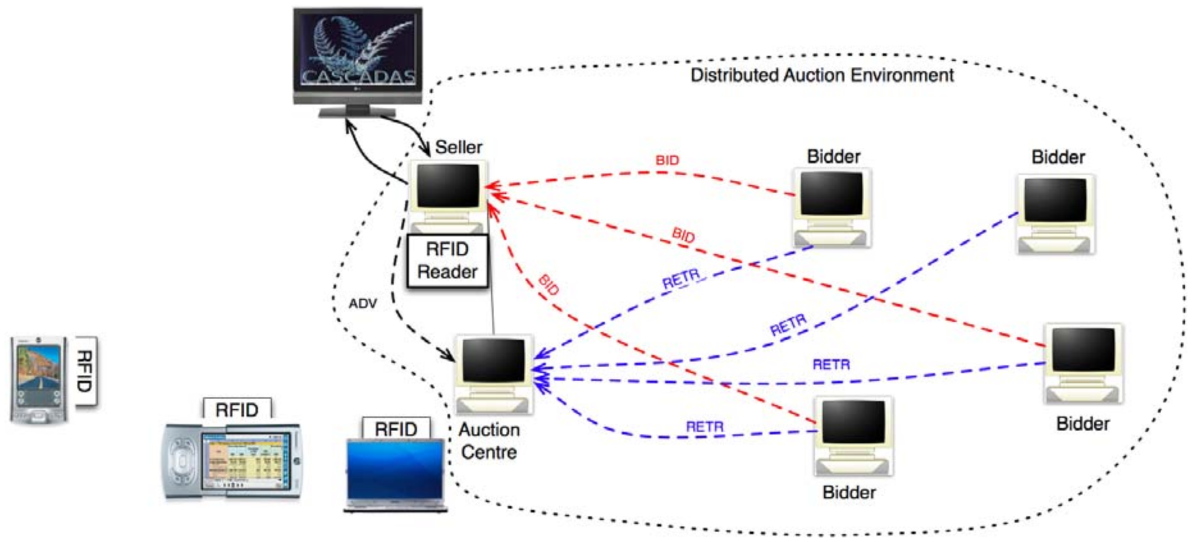

Figure 3 Auction System in the demonstrator scenario

In the scenario introduced as demonstrator, the machine communicating with the main display will act as a seller, while the item under auction is the concession to advertise, on the main screen, for a slot of time of fixed length. The actual auction will be conducted internally to a remote testbed, to whom the seller will be connected remotely through a Virtual Private Network (VPN). One of the machines on the testbed will act as Auction Centre (AC), facilitating the meeting between seller (that will contact the AC to ask the hosting of the auction advertisement) and bidders (that will ask the AC for retrieval of auctions). The rest of the machines in the testbed will act as bidders, competing for the slot of time under auction.

The auction transaction starts with the seller advertising its will to sell an advertisement slot of time through an English auction. The AC receives the message and makes the auction advertisement available. Sometimes later, one or more bidders will ask the AC for retrieval of a specific good. Bidders then come acquainted of the slot of time under auction, and decide whether to place a bid or not. Placement of a bid is realized by direct point-to-point communication with the seller. This, upon reception of a bid, notifies all competitors, and particularly the new highest bidder, of a new highest bid and contacts the AC to update the auction status.

Bidding decisions are taken, in the auction system, according to a *policy* which defines what and how parameters need to be considered when taking decisions. These are then evaluated mathematically through an aggregate function, and the policy contains facilities to map the value obtained by the function to the final decision to be taken.

The widths of the buckets, as well as the actual value of the consideration factor for each of the aspects defined, are assigned dynamically. Assignment will be based on current environmental conditions, i.e. context-awareness, and local conditions, such as for instance the financial status.

Bidders can be individual users, enterprises or a *coalition*, i.e. a number of single entities acting jointly. In this latter case, a leader acts on behalf of all the entities included in the coalition. Members democratically (for instance by means of a fair consensus protocol) take a set of decisions that will regulate the coalition behaviour. Decisions to be taken include, in the case of a coalition of bidders, leadership, maximum payable price for a good, bid increase etc. Coalitions will be able to revise own strategies based on past experiences, for instance increasing the maximum price and/or bid for a good if bids for a good of the same time have always been outbid in past.

4 Toolkit development

Cascadas Toolkit should enable autonomic, situation-aware, and self-similar development of communication services based on ACEs as core components. Moreover the toolkit will be released under an Open Source license to facilitate the wide-spread of technological achievements to the Scientific Community and industry; for this reason it is platform independent and adopted Java as programming language.

The other important aspect of the first prototype is the adoption of the DIET Agent Platform as an underlying unified execution environment. DIET Agents is a platform for developing agent-based applications. It was created as part of the EU-funded DIET project, where DIET stands for Decentralised Information Ecosystem Technologies. The reasons behind this choice are twofold: on one side we think the synergy between EU projects is an important result of the effectiveness of the project per se and on the other hand DIET tried to overcome the limitation of other agent platforms. In addition the support available by BT, the lead partner in DIET, is the other key towards the choice of DIET.

The DIET project is concerned with the development of an ecosystem-inspired approach to the design of agent systems [2]. In this context an ecosystem can be viewed as an entity composed of one or more communities of living organisms, in which organisms conduct frequent, flexible local interactions with each other and with the environment that they inhabit. Although the capability of each organism itself may be very simple, the collective behaviours and the overall functionality arising from their interactions exceed the capacities of any individual organism. These higher level processes can be adaptive, scalable and robust to changes in their environments. These elements are perfectly in line with the work carried in Cascadas as the Aces interaction is seen as an emergent behaviour where the group ability is greater than the sum of each individual acting alone.

4.1 ACE implementation

ACEs are defined to be autonomic, self-similar, light weighted, situation-aware, and semantically self-organising communication elements. The model developed for ACEs tries to satisfy all these requirements.

To realise self-similarity, though being light weighted, an ACE is separated into a Common Part, equally implemented for all ACEs, and a Specific Part that can be equipped with specific functionality differing from ACE to ACE [1]. This section shortly describes how the ACE model is implemented within the CASCADAS Toolkit. A more detailed description can be found in [6].

In order to be platform independent, the ACE model is implemented using the Java programming language. The DIET Agent Platform [8] is utilised as an underlying unified execution environment. It provides very basic features to organise the ACE components to a complete ACE as well as communication and control principles which help to manage large systems of ACEs.

In structural detail, the ACE model consists of a set of Organs each responsible for delivering a subset of the functionality of the whole component. Figure 4 gives an overview of an ACE and its Organs.

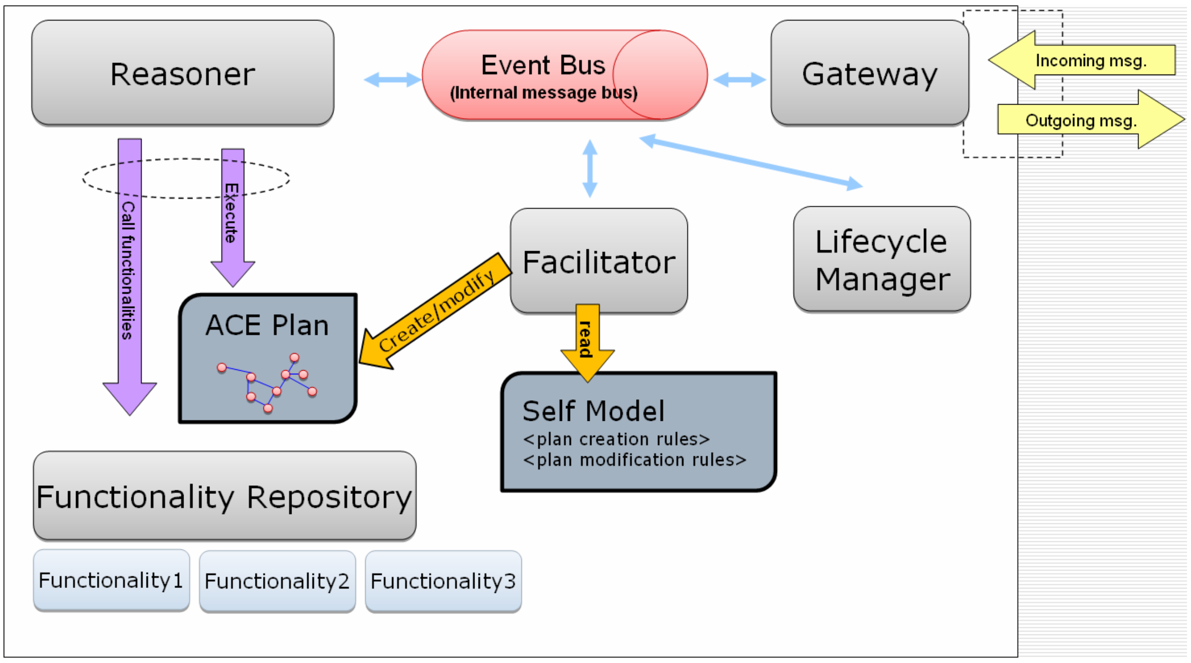

Figure 4 :Organs of an ACE

Within this model, the Specific Part is the functionality which is deployed into the Functionality Repository. An ACE developer can embed here all individual services the ACE needs to fulfil its goals. The Organs themselves as well as the features they offer form the Common Part which cannot be changed by ACE developers. It is uniform for each and every ACE and therefore builds the basis for self-similarity. Following, a list of all ACE Organs is presented explaining their functionality and the role they play within an ACE.

4.1.1 Gateway

Communication in the ACE model is based on events. The Gateway is the Organ which is responsible for managing external communication, i.e. between different ACEs. From the view of an ACE, the Gateway is the interface to everything which is not inside the ACE. External communication can occur in two different ways. First, it is possible to exchange events between ACEs in an uncoupled, session-less way. The REDS [7] middleware is used to implement this. Applying this communication mechanism, an ACE can subscribe to a set of events which it is interested in. Each time an according event occurs, REDS publishes it to all subscribed ACEs. REDS realises this by using clients, one for each ACE, that can send and handle events, and brokers which are responsible for managing the subscriptions and the routing of events.

The second external communication mechanism implemented in the CASCADAS Toolkit is point-to-point, session based communication using contracts between two ACEs. Establishing a contract with another ACE requires knowing its address. An EstablishContractEvent can then be sent in order to request a contract with the desired party. Thereafter a Mirror Agent is created using functionality of the DIET Agent Platform. This Mirror Agent then establishes a direct communication channel between the two contract parties until one of them quits the contract by sending a CancelContractEvent. For the second release of the CASCADAS Toolkit an advanced contract negotiation mechanism is planned to be implemented.

4.1.2 Bus

The Event Bus (i.e., Bus) is responsible for internal communication between the subcomponents of an ACE. Like the Gateway, it is based on events. Organs can subscribe to a set of events they are interested in. Whenever an event is sent, the Bus will dispatch it by calling appropriate handle methods that have to be implemented by each subscribed Organ.

The Bus supports both synchronous and asynchronous event handling. For asynchronous communication, events need to be sent and will then be further dispatched by the Bus. Communication in a synchronous way means that after an Organ has sent an event, it is blocked by the Bus until all subscribed components finished to handle the concerned event.

4.1.3 Facilitator and Self Model

The ACE autonomic behaviour is specified by the ACE developer within the Self Model and provided by the Facilitator. The Facilitator is the ACE Organ responsible for ACE Plan creation and modification. It processes the Self Model and creates a new or modifies an existing ACE Plan accordingly. Changing the ACE plan means changing the behaviour of an ACE.

The Self Model is defined through an XML file which has to be provided by the ACE developer. It contains all different Plans an ACE can process, as well as rules for their creation and modification. An ACE Plan is a state machine where states are connected to each other via transitions. Transitions contain actions and conditions for their execution. The Self Model basically contains the specifications for states and transitions as well as rules for building and modifying the state machine (the ACE Plan). Plan creation and modification rules are formulated in standard RuleML [10] syntax.

Facilitator actions are triggered via events. The Facilitator can be used to create a new ACE Plan via CreatePlanEvent. Plan modification is a continuous process, which depends on context information. In the current implementation the Facilitator subscribes to the Reasoner for receiving certain context data. Each time it gets notified about context data changes, it executes the Plan modification rules and modifies the ACE Plan accordingly. For introspection purposes, the Facilitator provides access to the Self Model of the ACE.

4.1.4 Reasoner and Plan

ACE Plans are generated by the Facilitator and executed by the Reasoner. They are state machines containing states and transitions. Each Plan has an initial state and a goal to achieve which is specified within the transition that contains the *goal achieved* description. Executing the ACE Plan means executing actions specified within transitions. Plan execution always starts at the initial state and ends after reaching a final state.

The Reasoner executes the Plan in two cases: first, immediately after it receives the new Plan from the Facilitator and second, when the ACE service is requested by another ACE. In case the ACE Plan contains a loop back transition between two states, the Plan will be continuously executed. In this current implementation the Reasoner also takes care of gathering the context information.

The Reasoner relies upon the OOjDREW rule engine [9]. It translates the ACE Plan into standard RuleML syntax, executes the rules in the rule engine, evaluates the outcome and performs actions accordingly.

4.1.5 Lifecycle Manager

The Lifecycle Manager is responsible for starting and stopping an ACE. It is triggered by the Reasoner which may request any lifecycle action according to the ACE Plan. The Lifecycle Manager will then prepare the requested action by informing all other ACE Organs. These have to prepare themselves for the announced action. Having done this, each Organ has to confirm to be ready by answering with a LifeCycleEventResult. The Lifecycle Manager collects and evaluates all results and finally executes the requested action when all Organs are ready.

The first version of the CASCADAS Toolkit implements the lifecycle actions *start* and *stop*. For the next release, it is planned to include cloning and moving of ACEs as well. These features will then enable ACEs to be mobile and will provide them with the ability to change their working environment if necessary.

4.1.6 Functionality Repository

The Functionality Repository is an Organ which is responsible for deploying and executing individual services of an ACE. It is used as a container and access point for the features, an ACE is equipped with. Arbitrary code, specifically programmed as well as already existing libraries, can be added to the Functionality Repository. Using the concepts of modularity and reusability optimises programming expenses for creating systems of ACEs.

To deploy a service to the Functionality Repository of an ACE, a library implementing the service and an XML descriptor describing it are necessary. The descriptor includes an identification (ID) and a name for the functionality that should be deployed. Furthermore, it lists the required input and output parameters.

Any service which is deployed to the Functionality Repository can be accessed using events. ACE internal functionality calls are dispatched by the Bus. If an Organ likes to invoke functionality from the Repository, a FunctionallityCallEvent has to be sent. Thereafter, the requested functionality is executed and the results are returned. In the case that an ACE likes to call a service provided by another ACE, the requester has to send a ServiceCallEvent to the provider. This results in the execution of the desired functionality and the delivery of the results. Of course, all service calls have to contain values for all parameters specified in the XML descriptor of the invoked service.

Looking at the Organs the ACE model is composed of, each of them contributes to fulfil the requirements listed above. A Common Part containing all Organs and being implemented equally guaranties self-similarity. The features and properties the Common Part provides are granted for each and every ACE. Keeping this common functionality restricted to what is absolutely necessary for all ACEs ensures their light weighted character. Anyway, the possibility to deploy any individual service to the Functionality Repository enables a practically unlimited diversity of functionality to be offered. Autonomicity and situation-awareness result from the ability of the Facilitator to react on the context an ACE encounters. Its behaviour, even its main goal can be adapted accordingly. The GN-GA

(Goal Needed – Goal Achievable) protocol is used to advertise own services and find those which an ACE cannot perform itself. Communicating and interacting this way enables the creation of a complex autonomic system.

4.2 Pervasive Supervision

To define the integration of supervision architecture into an ACE based service configuration we have to answer two basic questions:

1. How to exploit ACE mechanisms (protocols, interfaces) to (a) define the possible interactions between the configuration under supervision, and the supervision system, i.e. what the supervision system is allowed to monitor, and which actions it is permitted to perform if corrective measures become necessary; (b) obtain run-time information (monitoring); and (c) interact if the interference of supervision functions becomes necessary (i.e. the configuration under supervision is in a (or approaches a) non-desirable state.
2. How to define the supervision system itself (a) either as an integrated part of the ACE architecture, or (b) as a component or configuration of components that is architecturally separated from the ACE configuration under supervision.

4.2.1 Interaction with supervised ACEs

To provide a certain service, an ACE basically executes a workflow generated from its *self-model*. This workflow is called a *plan*. It comprises of states and transitions leading from one state to another. A transition is performed by calling a specific function from the function repository of the ACE (which is assumed to be stateless), or by sending/receiving external messages. Additionally, a session object is maintained that stores data necessary for the service computation. This session object may be modified by specific functions.

The self-model thus defines what is supposed to be the correct or intended function of an ACE. It explains the structure of states, and which actions are available to change from one state into another and how to update state information, Furthermore, a special attribute is has been introduced that is used to assess the desirability of a certain state, e.g. whether it is an initial, intermediate, or end state, an error state, etc. Thus the self-model can be understood not only as a functional specification of ACEs, but also as an expression of computation goals and purposes. Supervision can now be described as the task to make sure that those goals/purposes are fulfilled, by means of the functions that are made available by an ACE.

To answer question 1(a), we envision using the GA/GN protocol developed in WP1 to commit a contract between the ACEs to be supervised and the supervision system. The contract comprises the self-models of these ACEs, together with monitoring and control permissions for actions.

To answer questions 1(b) and 1(c), let us now have a closer look into the architecture of a supervision system: Figure 5 provides a summary of the elements of the supervision system (orange) and their relationship to the constituents of an ACE under supervision (gray).

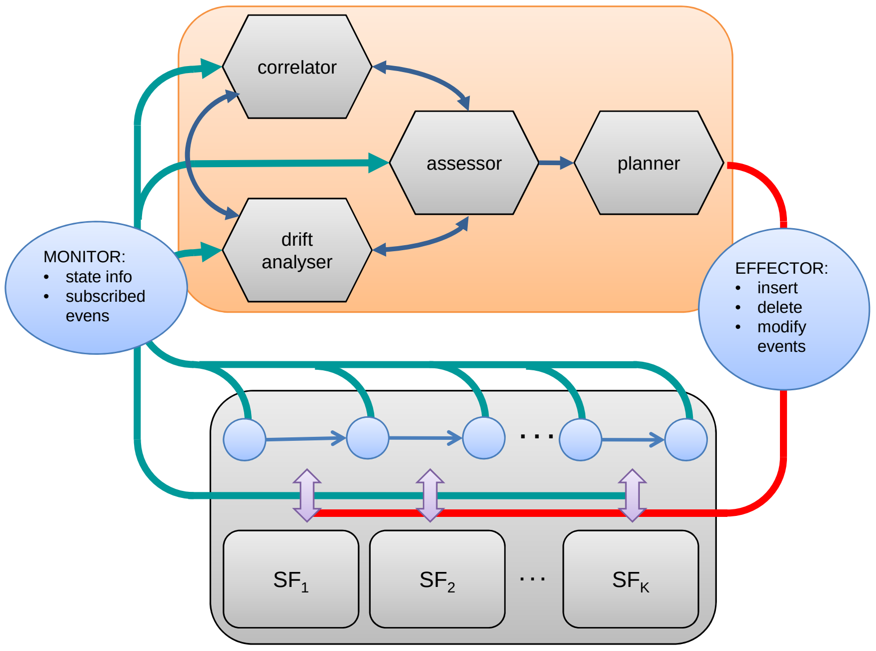

Figure 5 Architecture overview (singular ACE)

For monitoring, ACEs provide interfaces (a) to access state and session object, (b) to subscribe on information on specific function invocation or incoming/outgoing messages. Technically, this is solved by hooking up on the ACE internal publish/subscribe communication infrastructure.

We are now going to elaborate briefly on set of components that is available to build supervision systems for ACEs or ACE configurations. All these components will be implemented as ACEs.

– **Monitor** components links the supervision system with the ACE under supervision by utilizing the access mechanisms described above.

– **Correlators** are responsible to aggregate monitored data from distributed sources and to correlate them, in order to extract meaningful indicators of the current health condition of the system under supervision

– **DriftAnalysers** try to anticipate future problem situations in the system under supervision. Additionally, information from the environment may be used to supplement the analytical process.

Thus if Correlators are concerned with the current state of the system, DriftAnalysers are concerned with the possible evolution of the system. Finally,

– **Assessors** make assumptions on the current (or future) system health on the basis of raw data or the output of Correlators and DriftAnalysers, and invoke a Planner if necessary.

Monitors, Correlators, DriftAnalysers, and Assessor thus form an analysis system. The reactive part is provided by the following components:

– On the basis of the assessments generated by the Assessor, the **Planner** tries to compute a course of actions that is intended to resolve the detected problem. Planning is based on the actions described in the self-model of the ACE (or ACEs) under supervision.

- **Effectors** are responsible to execute plans. This is again done by intercepting internal events of the ACE under supervision. Events can be deleted, modified, or inserted.

4.2.2 Integration of Supervision Functions.

Let us now answer question 2. We use alternative 2.b for the following reasons.

- Separation of concerns. Integration of supervision functions into the ACE architecture itself would blow up the ACEs unnecessarily. In some cases, supervision functions may not be desired, possible, or necessary.
- Widening of scope. Some supervision functions require the coordinated interaction with a number of ACEs, which would be difficult to realize in the intrinsic scenario.
- Flexibility. In a number of cases, the described analysis/reaction cycle is obviously unnecessarily heavy-weighted. The external approach allows us to formulate the software architecture in a way that it is possible to flexibly use only the components that are needed, while others are not instantiated at all.

Thus we implement each of the components described in the previous paragraphs as a specialized ACE that runs asynchronously to the other components.This allows the flexible definition and setup of arbitrary complex control cycles comprising for instance chains of Correlators and DriftAnalysers, hierarchical Planners, and coordinated Effectors.

Now since these components are ACEs by itself, the GA/GN protocol can be used to discover available supervision components and the contracting mechanism developed in WP1 is available for the set-up of the configuration of the supervision system and the parameterization of its components (e.g. by rules, models, etc.). Therefore, supervision systems become an integral and pervasive part of service configurations, utilizing functions of the underlying platform for discovery and orchestration.

4.2.3 Implementation Roadmap

We conclude this section by elaborating on a road map comprising four phases to implement the described concepts, leading from of monolithic supervision systems to a pervasion of interacting supervisors is divided in the following phases

Phase 1 – "Basic supervision": This includes the implementation of the components described above, under the assumption that the system under supervision is a singular ACE or an unstructured set of ACEs. It comprises the basis monitoring and effection mechanisms explained above. Self-models of ACEs will be employed as supervision models for planning, i.e. we consider also the establishment of contracts between supervision system and system under supervision.

Phase 2 – "Hierarchies": In this phase, hierarchical supervision is considered. In this phase we will stipulate the assumption that service hierarchies are a-priory given, and do not change during service execution. This simplification will allow concentrating of hierarchical planning and distributed execution of hierarchical plans.

Phase 3 – "Self-configuration": The assumption that hierarchies are fixed will be dropped in this phase. We now consider ACE configuration with dynamically adapt their structure, and what this dynamism implies to the structure and function of the associated supervision pervasion.

Phase 4 – "Self-supervision": A final phase concludes addresses the question how a supervision pervasion itself can be made robust against problems.

4.2.4 Self-organization algorithms

The initial WP3 contribution for the first toolkit version is a standard interface called Clustering_Algorithm for distributed self-organization algorithms.The purpose to these algorithms [3] is to find a solution to the clustering problem, where the clustering problem, is defined as follows: given a network of interconnected nodes characterized by a type, the purpose of clustering is the creation of interconnected groups of nodes having the same type. Another related problem that is addressed by this algorithm is the reverse-clustering problem, where the aim is to constitute groups of nodes having different type. According to Ace-based toolkit structure, self-organization algorithms are part of the Specific Part Functionalities and accessed through the generic Clustering_Algorithm interface,
It provides all the methods that will govern the whole life of the algorithm, from its Initialization to its termination. The life cycle of a self-organization algorithm can be schematized as follows:

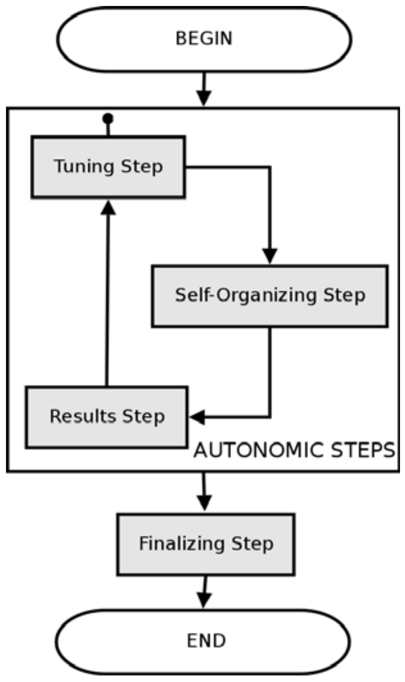

Figure 6 Algorithm steps

Initialization Step

The algorithm is instantiated and it receives information about the *local node*, (i.e., the Ace) an *ID* that identifies the current instance of the algorithm (to solve ambiguity when there are parallel executions of the same algorithm in the same ACE) and an interface used to communicate with its local ACE.

Tuning Step

In this step some algorithm proprieties can be optionally set by who is loading the algorithm.

Results Step

This is an optional step. The algorithm reports its results to the local node or on a remote node (useful to perform algorithm analysis).

Self-Organizing Step

The algorithm participates to the self-organization of the system by sending and receiving event messages.

Finalizing Step

In this step the algorithm terminates cleanly its execution, stops its threads and marks itself as completed. Only a completed algorithm can be safely unloaded from the Ace.

Each implementation of the ClusteringAlgorithm interface is parameterized with the possibility to perform the self-organized algorithms such as: clustering (aggregating nodes with the same type), reverse-clustering (aggregating nodes with different type), passive clustering and an active (or "on demand") clustering according to the work carried by Wp3 during the first year of the project. In particular the implementations that will be proposed are:

- Saffre Clustering; at the beginning of each iteration an initiator node is randomly elected, it chooses two neighbors having the same type, it creates a new link between them, and then it removes one of them from its list of neighbors.
- Fast Clustering: it's derived from the previous algorithm. In order to prevent all the useless iterations, it is possible to add the following constraint: a node cannot lose a link to another node if both nodes have the same (for clustering) or different (in reverse clustering) type
- Accurate Clustering; to link two nodes with different types unless they cause an increment in the number of links that have endpoints with different types.
- Adaptive Clustering is a self-adaptive version of the three previous algorithms with the aim to get only the best from the three algorithms above: it tries to use the most constrained algorithm (Fast algorithm) and, if it fails because the constraint is not satisfied by any neighbor, it switches to the Saffre algorithm. If even Saffre algorithm fails, it tries the least constrained algorithm (Accurate algorithm). If, for any reason, some new neighbor is added to the node, then the algorithm is changed again to the most constrained one.

The above algorithms use different strategies in choosing the nodes to cluster that give different results in terms of network type homogeneity and algorithm speed.

4.3 Security mechanisms

The contribution of WP4 to the CASCADAS toolkit consists in providing security functionalities to protect the communication within and among clusters of ACEs. These functionalities are deployed by using existing libraries implementing a set of cryptographic algorithms and hash functions.

The most used cryptographic primitives have been also deeply analyzed to understand their suitability for the communication environment envisioned in the CASCADAS project and, in particular, with respect to the heterogeneous nature of devices that embed Autonomic Communication Elements. This will help to understand what the performances of these algorithms and functions are with respect to key size, device capabilities and cryptosystem type.

In order to protect the system from traditional attacks that mine the system itself by exploiting communication vulnerabilities or by improper use of messages and resources, we focus on solving basic security problems that are common to any communication system. We refer to basic security problems as confidentiality, integrity and authentication.

With respect to the security architecture presented in [11], we focus on ACEs of type *B* for the first release of the toolkit as they provide specific cryptographic protection to the CASCADAS system.

The approach taken in WP4 consists in defining a secure framework that can be applied to the CASCADAS system if required by the communication environment. With this concept in mind, security is not deployed as a stand-alone component that is always embedded in the ACE model or implemented in the communication Gateway to provide security for any message or event communicate to external ACEs, but it is defined as service or functionality that can be either “aggregated” or exploited following the communication framework and service formation defined in the CASCADAS project. The advantages of this solution are twofold:

1) Security is used on demand when required, thus, reducing the unnecessary communication overhead introduced by message flow exchange either to agree on a shared key or to authenticate ACEs. In addition, the computational capabilities of the device running the ACE are not overloaded by extra computation if security should not be in place.
2) Security components can be re-used or better different cryptographic primitives can be used in an appropriate way by considering the data flow and the communication environment.

According to the ACE-based structure, security functionalities, implemented in the security library, are deployed in the Functionality Repository and accessed via the ACE event model. In our effort to come up with hands-on examples of typical security problems of a communicating system, we selected those cryptographic libraries that are compatible with the development environment chosen for the project. For these libraries, we consider the basic constructs that can be used in the CASCADAS toolkit to integrate and complement software components to achieve the basic security levels required by the application scenarios defined as prominent examples of the CASCADAS system.

In CASCADAS, we envision the presence of multiple ACE agents running on the same device. These ACEs implements basic functionalities in a virtual trusted environment. Complex services are deployed through aggregation or external exploitation of the single ACEs capabilities.

In this virtual community, ACEs that implement basic cryptographic functionalities have the role to provide security to other components when they require to communicate in a secure way with external ACEs outside the trusted domain. For instance, if the communication between two external entities requires data authentication and integrity, an ACE implementing HMAC will be used for the calculation of the digest. This ACE will exploits the capabilities of another ACE implementing a hash function, e.g. MD5 or SHA1, and the capabilities of a ACE implementing a key agreement protocol like Diffie-Hellman as shown in Figure 7.

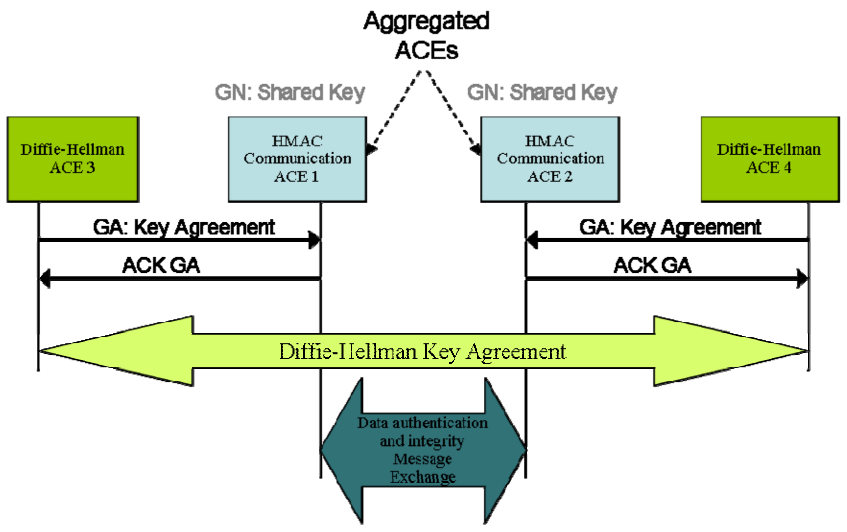

Figure 7 ACE Communication Model: ACE 1 and ACE 2 want to communicate securely. They exploit the capabilities of ACE 3 and 4 to agree on a symmetric key used to provide data authentication and integrity during the message exchange.

Due to different context of communication and to the heterogeneous nature of the devices, it is important for an ACE to choose the best option in terms of cryptographic algorithm available in the trusted domain. For this reason, we have also focused on practical considerations that need to be pondered when deciding which security service or cryptographic function needs to be selected to achieve a specific goal, both taking in consideration the application performance, storage and computational requirements.

For instance, not all basic ciphering algorithms are suitable for the communication with mobile devices, which are characterized by limited processing resources. Thus, it is important to assess the processing overhead of cryptographic algorithms independently of specific software implementations to derive the feasibility of deploying algorithms on mobile devices and networks.

4.4 Knowledge network

The overall development process of WP5 software has been so far organized along three stages. During the first year of the project relevant conceptual components have been defined and the overall structural framework has been drawn up, which has lead to the alpha release of knowledge network software. All of these are described in [4]. The research in the first half of the second year has concentrated on aspects that support not only the autonomic principles but also the utilisation of stigmergic principles for flexible query optimisation as well as the initial self reasoning mechanism and context verification. This has lead to the beta release of knowledge network software, described in [5]. For the second half of the second year of the project, we aim at refining and further developing the concepts proposed so far and integrate the results of WP5 development into the ACE framework, by implementing knowledge networks in terms of ACEs and by making knowledge network services available to ACE-based services. The plan for implementing this is described in the following section.

4.4.1 ACE Integration Process

The integration of the ACE model (WP1) and knowledge networks (WP5) will proceed in two main directions:

- An interface will be developed to retrieve access and produce information in the knowledge network that is compatible with the standard way adopted by ACEs to interact with each other (i.e., GN-GA protocol and contracting).
- The core mechanisms of knowledge networks will be integrated on top of the ACE architecture. This completes the integration in that the knowledge network components (knowledge atoms and knowledge containers) are actually ACEs themselves.

4.4.2 Knowledge Network Interface

ACEs access the knowledge network using the features of the toolkit

- The GN-GA protocol will be used by ACEs to locate the most appropriate Knowledge Container (KC) to satisfy their knowledge need. Specifically we subclass the GN and GA message classes to have knowledge network specific message format: KNM (Knowledge Needed Message), KAM (Knowledge Available Message).
    a. KNM provides a template to express interest to a class of context information (e.g., I want all the information related to a specific region, or to a specific user)
    b. KAM provides another template specifying which kind of information is produced by a given entity.

  The communication middleware using in the toolkit (i.e., REDS) takes care of routing KNM messages to matching KAM messages so that relevant information sources are discovered.
- Once suitable KCs have been discovered, they can be directly accessed via the point-to-point mechanism to actually retrieve the requested information. Such point to point interaction also allows negotiating on specific information access modalities (e.g., quality of information, sampling rate, access rights, policies etc.).

4.4.3 Realising Knowledge Network via ACEs

The following step is to realise the core mechanisms and components of the knowledge network on top of the ACE architecture describe in [1]. In order to modify knowledge network components, namely KA and KC according to this architecture (as well as any other ACE-based component) the Self Model and the required Functionalities thereof has to be specified.

A complete description of these ACE internal components can be find in [1]. Here only the basic mechanism of ACE's that are relevant to knowledge networks are recapitulated:

- The ACE Self Model describes at an abstract level what the ACE is capable of doing and how it realizes its functionalities.
- The Facilitator creates a Plan on the basis of the current system circumstances (i.e., context) by loading specific behaviour from the Self Model. The Plan is represented by means of a finite state automaton.
- The Reasoner is the component in charge of running the finite state automaton triggering its transitions. Upon the triggering of a transition, the Plan indicates which Functionality has to be called to achieve the transition.

All these components apart from the Self Model and the Functionalities are either common to all the ACEs or dynamically generated (e.g. Plan from the Self Model).

4.4.4 Implementing Knowledge Atoms

At the most basic level KAs will be implemented according to the trivial Self Model (that actually constitutes a single constant Plan) as depicted in Figure 8.

This Self Model mainly states the following actions:

- If receiving a KNM message that matches the kind of information the atom is able to produce, then reply with a KAM message describing such kind of information
- If a query point-point message arrives, then execute the getValue functionality.
- The getValue functionality actually accesses the low level sensor machinery, retrieves the result and sends it back to the requesting ACE. It is worth remarking that since point-to-point messages are really sort of RMI invocation, the sending of the result back to the client is actually a simple return statement.

4.4.5 Implementing Knowledge Containers

KC will be implemented according to the Self Model in Figure 8.The Self Model basically states:

- If receiving a KNM message that matches the kind of information the atom is able to produce, then reply with a KAM message describing such kind of information
- If a query point-point message arrives, then execute the queryAtoms functionality.
- The queryAtoms functionality actually sends a point-to-point query message to the atoms registered within this knowledge container. The replies will be combined according to a KC-specific policy – to be implemented within the queryAtoms functionality – and the final result is returned to the enquired ACE.
- Every T seconds, the knowledge container looks for knowledge atoms (or other knowledge containers) able to provide information that are suitable to its need. It performs this by sending a KNM message specifying the kind of information it is willing to aggregate (i.e., contain).
- Upon the receipt of suitable KAM messages the container calls the storeAtomAddress functionality.
- The storeAtomAddress actually registers the atom in the container by storing its address in a suitable repository. Registered atoms will be queried upon the receipt of a query message.

It is finally worth noticing that the above self models and functionalities are only a first attempt of mapping knowledge network functionalities into the ACE model. In future versions more advanced mapping taking into account more context-related information would be developed.

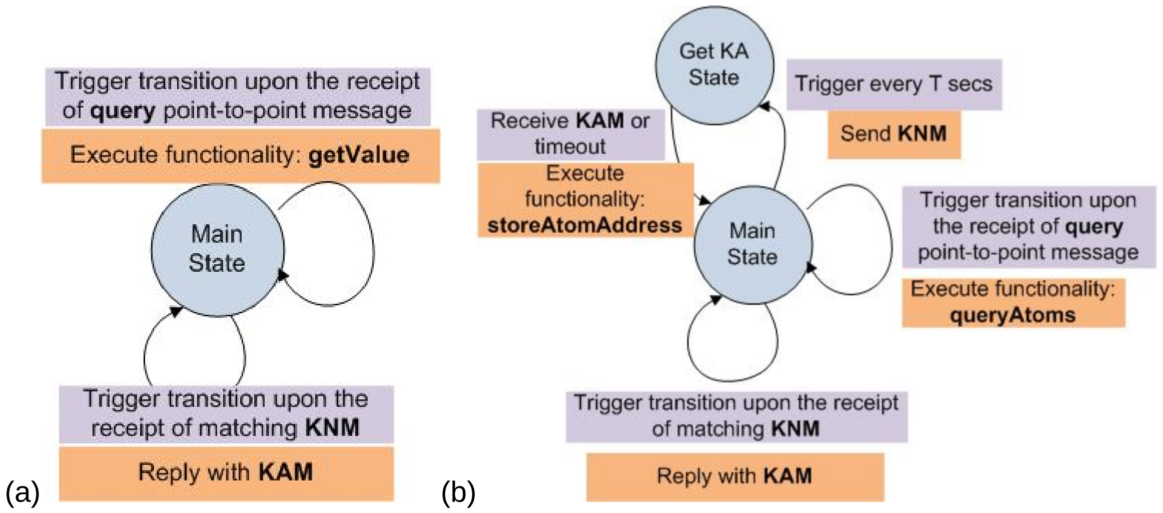
Trigger transition upon the receipt of **query** point-to-point message
Execute functionality: **getValue**
Main State
Trigger transition upon the receipt of matching **KNM**
Reply with **KAM**
(a)
(b)
Get KA State
Trigger every T secs
Send **KNM**
Receive **KAM** or timeout
Execute functionality: **storeAtomAddress**
Main State
Trigger transition upon the receipt of **query** point-to-point message
Execute functionality: **queryAtoms**
Trigger transition upon the receipt of matching **KNM**
Reply with **KAM**

Figure 8 : (a) Self-Model of a Knowledge Atom (b) Self-Model of a Knowledge Container

5 Conclusion

CASCADAS project has the main goal of identifying and developing an innovative component-ware architecture for the development of innovative situation aware and autonomic services.

The Autonomic Communication Element (ACE) is the basic component abstraction over which the CASCADAS vision is built. The project development activities aim at prototyping a toolkit based on distributed ACES characterised by autonomic features (self-configuration, self-optimization, self-healing, self-protection, etc). Services will be created and executed (in a distributed way) by the self-aggregation of ACEs. This architecture will be demonstrated in application use-case by means of a tool-kit developed by the Consortium.

This deliverable (collecting and integrating contributions from all WPs) has provided a preliminary description of: 1) the first release (at M18) of the ACE toolkit (i.e. run time environment and libraries); 2) the roadmap leading to an integrated release of the toolkit; 3) a brief description of the application use-case that will be adopted to demonstrate the features of the toolkit.

Regarding future steps, it should be noted that the first release of the toolkit (currently available) provides all basic ACE functionalities which allow developers to create ACEs providing basic services. From the lifecycle point of view, the basic lifecycle functionalities starting and stopping of ACEs are supported, whereas cloning and migration will be implemented in the future. Therefore, in the current implementation ACEs are static components. They will remain in the environment of the initial creation. ACE communication is event based. We distinguish between internal and external ACE communication. The Internal communication is provided by the ACE organ called Event Bus while the external ACE communication is handled by the Gateway. The Service lookup is implemented by the usage of GN/GA protocol. The ACE autonomic behaviour is specified by the ACE developer within the Self Model and carried out by the Facilitator.

In cooperation with other WP-s, WP1 will test and evaluate the current CASCADAS Toolkit implementation. A list of requirements will be created with the improvements that should be implemented in the next release.

From WP1 point of view, the following tasks are planned:

- The concept of context data acquisition should be redesigned in terms of moving this functionality from the Reasoner to the Functionality Repository.
- The Lifecycle Manager should be enhanced with cloning and moving functionalities in order to support mobility of ACEs
- Further enhancements of the Functionality Repository in terms of loading functionalities should be implemented.
- General improvements of most ACE Organs should be made in order to enable additional functionalities and to enhance the ACE stability.

The main task of other WPs will be to integrate coherently into ACE run-time environment the libraries of tools and software modules[1] that they have developed in the building phase.

6 References

[1] Manzalini, A. et al.: "CASCADAS Deliverable D1.1 – Report on state-of-the-art, requirements and ACE model", 2006

[2] Cefn Hoile, Fang Wang, Erwin Bonsma and Paul Marrow, *"Core Specification and Experiments in DIET: A Decentralised Ecosystem-inspired Mobile Agent System"*, Proc. 1st Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS2002), pp. 623-630, July 2002, Bologna, Italy

[3] Saffre,F. et al.; "Cascadas Deliverable D3.1 Aggregation Algorithms, Overlay Dynamics and Implications for Self-Organised Distributed Systems",2006

[4] Zambonelli, F et al.: "Cascadas Deliverable D5.1 Knowledge Networks Specifications, Mechanisms, and Alpha Software Release",2006

[5] Baumgarten,M et al.: "Cascadas Deliverable D5.2 Extended Beta Release Software for Knowledge Networks",2007

[6] Haseloff, S. et al.: "CASCADAS Deliverable D1.2 – Prototype implementation (release 1)", 2007

[7] REDS – A Reconfigurable Dispatching System, available online: http://zeus.elet.polimi.it/reds.

[8] DIET Agent Platform, available online: http://diet-agents.sourceforge.net.

[9] OOjDREW – Object Oriented Java Deductive Reasoning Enginge for the Web, available online: www.jdrew.org/oojdrew/.

[10] RuleMl - Rule Markup Language http://www.ruleml.org/0.88/

[11] R.Cascella et al: "CASCADAS Deliverable 4.1 :Security architecture", 2006

[1] For details see specific Deliverables due by M18.