IST IP CASCADAS "Component-ware for Autonomic, Situation-aware Communications, And Dynamically Adaptable Services" "

Bringing Autonomic Services to Life

Deliverable 1.1

Report on state-of-art, requirements and ACE model

Status and Version:	Final	
Date of issue:	09.01.2007	
Distribution:	Project Internal	
Author(s):	Name	Partner
	Antonio Manzalini (Editor)	TI
	Antonietta Mannella	TI
	Rosario Alfano	TI
	Edzard Höfig	FOKUS
	Marco Mamei	UNIMORE
	Borbala Katalin Benko	BUTE
	Tamas Katona	BUTE
	Rico Kusber	UNIK
	Nermin Brgulja	UNIK
Checked by:	Franco Zambonelli Ricardo Lent	UNIMORE ICL

Abstract

The overall objective of IST Project CASCADAS (http://www.cascadas-project.org/) is to develop and validate an autonomic framework for creating, executing and provisioning situation-aware and dynamically adaptable communication services. Particularly the project development activities aim at prototyping a toolkit based on distributed self-similar components characterised by autonomic features (self-configuration, self-optimization, self-healing, self-protection, etc.).

This document constitutes the Deliverable 1.1 "Report on state-of-art, requirements and ACE model".

Table of Contents

1 Document overview

The overall objective of CASCADAS is to develop and validate an autonomic framework for creating, executing and provisioning situation-aware and dynamically adaptable communication services. Particularly the project development activities aims at prototyping a toolkit based on distributed self-similar components (Autonomic Communication Elements) characterised by autonomic features (self-configuration, self-optimization, self-healing, self-protection, etc). The Autonomic Communication Element (ACE) is the basic component abstraction over which the CASCADAS vision is built. Services are being created and executed (in a distributed way) by the self-aggregation of ACEs

This document, constituting the Deliverable 1.1 "Report on state-of-art, requirements and ACE model", report the main results (achieved during the first year of the project) about the definition of the ACE model and its interactions mechanisms.

CASCADAS has adopted an application-oriented approach: starting from scenarios and related use-cases, high level requirements have been defined and are being used by WPs activities.

In particular, the document is structured as follows after the introduction to Autonomic definition; the chapter 3 describes the project vision; the chapter 4 describes the application scenario and the requirements collected through the interaction with the other WPs. Chapter 5 is devoted to a review of some existing architectural models and platforms offering autonomic features by highlighting commonalities and differences with respect to the CASCADAS objectives. Chapter 6 describes the components envisioned for the Autonomic Communication Element, the core of this deliverable while chapter 7 illustrates an overview of techniques/tools that might be adopted for developing the ACE architectural model. Chapter 8 focuses mainly on the autonomic aspects addressed by the project.

2 Introduction to Autonomic principles

If computer systems manage themselves, if networks organise themselves to establish a wide-ranging, high-quality communication, if outages are reduced to zero because of reliable error detection and correction systems, and if IT-professionals do not have to keep such systems running but just have to further improve them, then we have reached the era of autonomic computing.

The problem we are faced with is the fact, that IT systems become more and more complex. In order to permanently increase the efficiency in our everyday work, to advance comfort and to continually create new services, we develop faster, cheaper and smaller computer systems. The price for this improvement is a drastic rise in complexity affecting hardware as well as software. This, if not limited, will lead to a situation in which the installation, configuration and administration of a system cannot be efficiently realised even by a team of IT professionals.

Autonomic computing systems are considered to be a potential solution to this problem. IBM introduced the *"Autonomic Computing Initiative"* in the year 2001, with the aim of developing self-managing systems (cf. [4] , [19]). "Autonomic" is derived from the human autonomic nervous system. Its property to act self dependent and without being controlled by any other entity, but controlling itself was taken as model to be applied to future technical computer systems. In this context, "IBM defined four general properties a system

should have to constitute self-management: *self-configuring, self-healing, self-optimising and self-protecting*. These are accompanied by four enabling properties or attributes, namely *self-awareness, environment-awareness, self-monitoring and self-adjusting"* ([30], also cf. [19]).[1] The IBM vision of autonomic computing implies that implementing self-managing attributes involves an intelligent control loop which collects information from the system, makes decisions, and then adjusts the system where required.

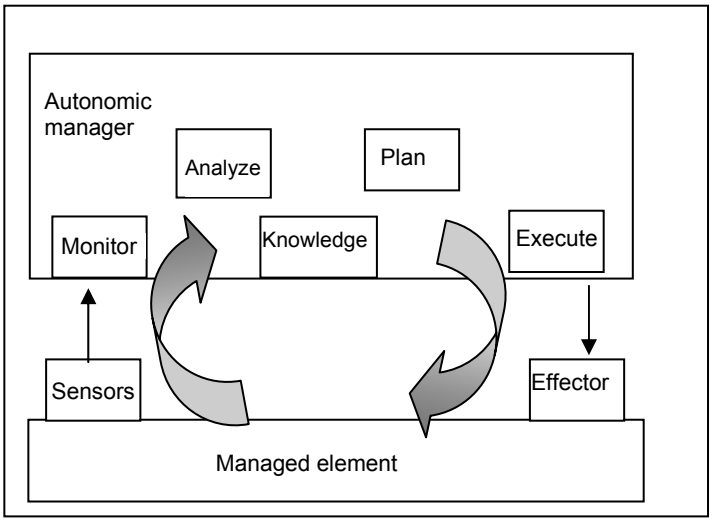


Figure 1 IBM-MAPE architecture

The MAPE-K architecture organises the control loop into two main elements: a managed element and an autonomic manager. Thereby, a managed element is what the autonomic manager is controlling and an autonomic manager is a component that implements a particular control loop. Figure 1 illustrates the MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) reference architecture proposed by IBM within the autonomic computing initiative.

The **managed element** is a controlled system resource, which can be either a single resource (e.g., a web server, database server or router) or a collection of resources (e.g., a pool of servers, cluster or business application). The managed element is monitored through its sensors, providing mechanisms to collect information about the state of an element. Effectors, which are mechanisms that change the state of an element, allow for controlling the managed element. A combination of sensors and effectors forms the management interface that is available to an autonomic manager.

The **autonomic manager** is a component that implements the control loop as consisting of four stages that share knowledge:

[1] This first set of attributes is now enriched with the inclusion of features such as self-anticipating, self-adapting, self-critical, self-defining, self-destructing, self-diagnosis, self-governing, self-organized, self-recovery, self-reflecting, and self-simulation[30]. Yet the initial set still represents the general goal.

The **monitor stage** collects, aggregates, filters, manages and reports details (metrics and topologies) collected from sensors related to a managed element, to provide both self-awareness and awareness of the external environment.

The **analyze stage** provides the mechanisms for modelling and correlating complex situations (e.g., time-series forecasting and queuing models). These mechanisms allow the autonomic manager to learn about its environment and help to predict future situations.

The **plan stage** organises the actions needed to achieve goals and objectives.

The **execute stage** controls the execution of a plan (considering also on-the-fly updates).

The plan and execute parts decide on the necessary self-management behaviour that will be executed through the effectors.

The four functions - monitor, analyze, plan, and execute - consume and generate *knowledge*. A large amount of this knowledge comes from the first step of the control loop: monitoring. It is important to consider the type of data that is necessary. If large amounts of data are stored, performance might deteriorate because even if data has no relevance for the system, it is constantly being monitored. All known information about the system is provided to the knowledge part which can grow as the autonomic manager learns more about the characteristics of the managed resources. The gathered knowledge is continuously shared among the four functions in order to improve their decision making processes. The monitor-, analyze-, plan-, and execute-parts collaborate and exploit the common knowledge to provide the control loop functionality.

The IBM MAPE-K architecture is not related to a specific technology. Instead its purpose is to work with existing computing technologies, as well as with new technologies that will emerge in the future.

The idea of autonomic computing offers interesting aspects to be further investigated in the CASCADAS project. Nevertheless, the situation we are faced with in the scope of CASCADAS differs significantly from the IBM perspective in the following points.

Autonomic computing systems, as they are considered by IBM, are seen to be complex and integrated. ACEs (Autonomic Communication Elements), the central components in CASCADAS are partially expected to be light weighted. The envisaged environment will contain a variety of ACEs, all acting autonomic themselves. For that reason, autonomic behaviour will not be realised by large and computational expensive subsystems but will emerge as an effect of ACE aggregation and cooperation among different system components, e.g., interaction with the knowledge network. This last point is the main aspect which mostly differentiates the IBM approach from the CASCADAS approach: in the latter not only each component is autonomic, but even components interactions at system level are autonomic.

The consequence of the distributed character of a network of ACEs is a very high degree of heterogeneity and dynamic. Different ACEs experience different environments and varying situations. So, they have to be aware of the situation they are in to be able to autonomically adapt themselves to occurring changes.

Heterogeneity, variability, and the light weighted and distributed character of ACEs require a high communication effort among ACEs. The ideas of autonomic computing have to be further investigated and developed. They serve as a source of inspiration and a basis to build on. Nevertheless, for the purpose of the CASCADAS project this basis has to be extended and focused on strategies of autonomic communication to manage the given

challenges. The following section describes what autonomic communication is and how it is related to CASCADAS.

2.1 Situated and Autonomic Communication

The term "situated and autonomic communication" (short AutoComm hereafter) refers to a long-term research effort that is projected in leading to a radical paradigm shift towards self-organising; self-managing and context-aware autonomous networks (see Fokus AutoComm whitepaper [18]). Networks are envisaged as being built of large amounts of structurally simple and self-similar autonomic building blocks that are able to connect in an ad-hoc manner, forming new networking primitives for service provisioning that potentially may exhibit complex behaviour. AutoComm systems are adaptive: Changes in the usage pattern, formation or any other environmental circumstances may initiate the elements to re-configure and adjust their behaviour, trying to optimise service provisioning with regard to a new situation.

The aims of AutoComm are similar to the one's of IBM's initiative on Autonomic Computing [4], because the underlying problem is the same: How to manage the complexity of future computing systems (in the case of AutoComm that is telecommunication systems).

Complexity of connected systems is constantly increasing. For example the bandwidth available on wireless, coupled with ad-hoc networking, could in perspective rival the capacity of backbones. A plethora of interwoven devices that form a ubiquitous, mobile information access layer based on a various technologies (e.g., Wi-Fi IEEE 802.11n, WiMax IEEE 802.16e, Bluetooth UWB or UMTS HSDPA / HSUPA), all bundled within the same case, is quickly emerging.

Providers are investing large amounts of management and maintenance effort to enable a smooth operation of current-day internet with its more than 1 billion users, nearly half a billion registered domains and several hundreds of protocols in use. But bandwidth is cheap and users are accustomed to flat-rate price models, making it hard for carriers to invest capital in the network infrastructure, operating on a best-effort base and making revenues disappear when provider-specific streams are converging in the internet. To stay in the market, telecommunication providers are coming up with a multitude of diverse and innovative services, which on the one hand need to be constantly maintained and on the other hand call for intra-provider settlement interfaces that need to be standardised and managed, as well (see [25]). Coping with this growing complexity will become more and more problematic, so 'Keep it simple' should be one of the basic principles of a future communication paradigm. For AutoComm this is in regard to single network elements and centred on networking Selfware. Selfware is the common name for all the "self" system properties, i.e., for a number of tightly coupled processes – sensing, data handling, decision making and communication – that are used to achieve system properties of self-awareness, self-healing, self-configuration, self-optimisation, etc. and that can be instantiated in a multitude of variations.

AutoComm studies "the individual network element as it is affected by and affects other elements and the often numerous groups to which it belongs as well as the network in general". The "goals are to understand how desired element's behaviours are learned, influenced or changed, and how, in turn, these affect other elements, groups and networks"(cf.[18]) The idea is to engineer micro-properties in order to cause a desired macro-behaviour, optimally without any human interference and in a self-organised and

evolutionary way. To accomplish this, AutoComm research has to find general principles enabling this kind of behaviour by building upon results from diverse scientific fields like control theory, cybernetics, machine learning or social and human sciences. The discovered principles have to be verified, evaluated and cast in a shape that is making them useful for the wider autonomic community. That is, new approaches to system modelling have to be devised that can cope with incomplete information and that will support evolution; engineering processes have to be re-thought along with their associated tools, languages, methods and metrics. Checking for validity of systems would need to be done in new ways, requiring not only a pervasive approach for verification or for guaranteeing certain system properties, but also requiring new test infrastructure and tools.

First prototypes are already visible (for example the FOCALE architecture from Motorola [29], but generally efforts are concentrated on research, rather than implementation as deduced from the member list of the Autonomic Communication Forum [2]).

3 Project Vision at a glance: the Open Autonomic Service Environment

Providers' Service Framework (as we know it today) is that set of platforms, functionalities, systems and data for the creation and execution of services; furthermore current solutions includes also related interfaces towards a control layer and towards systems for management and provisioning.

In order to save CAPEX/OPEX[2] and to generate new potential sources of revenues, technologies and solution for next generation service frameworks are required. Innovative proposals are expected to be characterised by distribution of resources and de-centralization of functionalities. The requirements above are likely to be met by a dynamically configurable architectures using a P2P overlay network (generally IP-based, both Internet and Service Providers' networking solutions).

On the other hand, Web 2.0 is bringing a shift of business models thus forcing players of the service arena to look for frameworks capable of following rapidly market trends (even with new approaches, e.g., including advertisers in the value-chain) for composing and providing even short-life personalized services (beta versions à la Google).

Given that context, there is the need of finding solutions capable, for example, of composing and running complex services starting from highly distributed basic components (even outside the Operators' domain, e.g. including Service Brokering capabilities). Services may be executed starting from the dynamic self-aggregation of distributed self-managing autonomic components. This approach would allow also managing the huge amount of heterogeneous data and information, making knowledge available, in the proper form, where and when it is necessary. Furthermore the same component-based environment may be capable of "handling" service-related knowledge and data.

CASCADAS is adopting an application-oriented approach: starting real application scenarios highlighting the above needs and requirements, project vision is proposing an Open Autonomic Service Environment for the future evolution of today service frameworks.

[2] CAPEX = Capital Expenditures, refers to the cost of developing or providing non-consumable parts for the product or system. OPEX = Operating Expenditures are the on-going costs for running a product, business, or system.

The Open Autonomic Service Environment is expected to be the highly distributed platform for composing, executing and providing situation-aware and dynamically adaptable communication and content services. The tool-kit developed in the project will be used to demonstrate such vision specifically referring to some use-cases of particular interest (such as pervasive communications, etc).

The essence of the innovation stands in exploiting highly distributed resources (even commodity servers at low-cost) running autonomic S/W solutions based on distributed self-aggregating, self-organising components. The overall architecture may enhance the self-similarity of Google technology [39](both pizza-box servers and clusters of servers have the same functional architecture) supporting a distributed replication of data. This will allow high levels of availability also starting from low-cost commodity H/W.

A first key characteristic of the Open Autonomic Service Environment is the distribution of resources and infrastructures at any level, introducing the distributed paradigm in traditionally monolithic field like a telecommunication company infrastructure. In principle a distributed system is a collection of independent sub-systems (linked with distributed software) that appear to the operators/users of the system as a single entity. Distributed software enables sub-systems to coordinate their activities and to share the resources of the system - hardware, software and data. Motivations for distributed systems are:

- Functional distribution
- Physical separation
- Resource Sharing
- Economics

Another aspect of the Open Autonomic Service Environment is a P2P network that allows resources communications. This is dynamic network where peers resources can act as server and client indistinctly and peers might freely join and leave the network over the time. P2P communications enable large numbers of resources to share information and resource directly without dedicated central servers. P2P characteristics are:

- De-centralization
- Ad hoc behaviour
- End-to-End communication
- Shared ownership
- Scalability/Reliability

The above two characteristics together with the deployment of autonomic distributed S/W solutions is expected to allow executing and provisioning situation-aware and dynamically adaptable communication and content services; furthermore data and information broadly distributed may be effectively handled to make it available where and when it is necessary. Autonomic S/W components, distributed over the resources, should perform also "embedded" self-management behaviours (for fault, configuration, accounting, performance and security).

The Open Autonomic Service Environment vision is expected to be capable of overcoming at least two serious bottlenecks (in current solutions):

- Software complexity and system heterogeneity: individual system are increasingly complex to maintain and operate; heterogeneous systems are becoming increasingly connected, and behaviours, execution context, interactions not known a priori
- Huge amount of data and information that should be collected, handled, replicated, correlated and made available when and where it is necessary

Regarding the former, self-management solutions as offered by autonomic behaviours of the environment may represent a sound solution. The self-managing characteristics of the distributed resources made them capable of hiding completely such complexity to operators and users: e.g., systems may make decisions on its own, using high-level policies from operators, constantly checking and optimizing status and automatically adapting to changing conditions.

Concerning the need to deploying solution capable of collecting, handling, replicating and managing the huge amount of data and information available, again an autonomic and highly-distributed environment is providing key features such us hiding complexities and high scalability (mandatory in a context such rich of data).

Concerning some of the advantages produced by the development of such vision, from the Customer viewpoint, an Open Autonomic Service Environment can offer:

- "Simpler and better approach" to service
- Customization of services (Customers' profile, context, etc.)
- Services meeting better(or even anticipating) Customers' needs
- Pervasiveness of contents and communications service

From the Service Provider viewpoint an Open Autonomic Service Environment can offer:

- Cost Optimization
- Enabling an horizontal TLC-IT integration
- Using low-cost H/W and *smart* autonomic S/W
- Adopting self-management (self-configuration, self-healing, self-optimization etc.)
- Generating New Revenues
    - Picking the opportunities offered by some ongoing trends (and the future related evolutions) of the web (e.g., Web2.0, Web3.0, etc.) with more flexible and open solution for executing services
    - Enabling new business models based on sharing resources, service enabler (TLC and IT) and data highly distributed

The CASCADAS tool-kit will be used to validate such project vision (i.e., Open Autonomic Service Framework). Use-cases have been selected to demonstrate (into the test-beds) executing and provisioning situation-aware and dynamically adaptable communication and content services.

3.1 The Autonomic Communication Element

The key ideas of the project is to identify and rely on a new model of distributed components, called ACEs—Autonomic Communication Elements, able to autonomously

self-organize with each other towards the provisioning of specific user communication services, and able to self-adapt such provisioning to social and network contexts [1].

The ACE is the main building block needed to implement the vision of Autonomic Communication outlined above. It will provide the basic interfaces and mechanism aimed to support self-similarity, self-organization, situation awareness and any other key aspect needed to implement the Open Autonomic Service Framework.

The basic structure of the Autonomic Communication Element as outlined in the following sections has been defined on common agreement among WP1 members and it is defined taking into account the requirements form CASCADAS application scenarios and the development of other CASCADAS WPs.

4 Application scenario and requirements

The purpose of this section is to describe the proposed application scenarios collecting among partners to define the requirements in order to provide an efficient support to the ACE model definition.

4.1 Motivating Examples for Autonomic Communications

In CASCADAS we have identified three main application scenarios to drive the projects activities and goals. Two scenarios come from the pervasive computing area and another one is from wide-area Internet computing.

- **Smart Environments Supporting Independent Living**.

As the population continues to grow, society is faced with the challenge of supporting those within the community who still remain within their own homes and are not fully independent.

This scenario introduces novel techniques for person-centric services in pervasive spaces. From a technical perspective, It proposes, how such services could be realised based on a distributed network of knowledge, facilitating dynamically combined and flexible service provision that engenders service continuity. This proposal assumes that individuals are equipped with some devices (e.g., a smart phone or a PDA) able to determine the user localization (using GPS or less expensive local hardware) and to interact with a wireless network that should be provided by the ambient. One of the challenges for future smart environmental infrastructures is the need to reason about "situation" and to understand the deduced behaviour. To do this they are required (both at the level of individual components and as a whole) to be introspective, and to feed back the results to improve performance. While this process provides the knowledge with which they can, eventually, manage and configure themselves it does also make them more self-aware or in short it makes them smarter.

- **Behavioural Pervasive Content Sharing**.

Several pervasive computing applications are rooted on providing personalized content to users anytime and anywhere. This scenario is based on devising suitable mechanisms to provide the user with the best available content given information such as the user profile,

his/her location, the device being used to access the content, the user current activity, etc. Two exemplary applications of this general scenario are *behavioural pervasive advertisement* and *pervasive tourist information*.

1. *Behavioural Pervasive Advertisement (BPA)* applies pervasive computing techniques and technologies to the nowadays emerging advertisement technique called behavioural advertisement (or behavioural targeting) which tries to ensure that advertisers reach the target audience in a more effective way. This application proposes to extend such technique to any communication context where user interests and needs can be grasped. Moreover, exploiting the pervasive nature of CASCADAS based applications, BPA may provide customized contents and advertisement, not only during web navigation, but different channels may be personalized to the single user or to groups of users (e.g., digital screens in the road may autonomously provide advertisement customized according the users' profiles moving by the screen position).
2. *Pervasive Tourist Information* considers that people, using handheld devices, can connect to a tourist guide system of a city/museum/exhibition and receive useful information such as nearby objects, 'what-to-visit- next' or "how-to-get-to" suggestions, hints about public transport etc. The location of the tourist, together with his/her profile and current activity are used to differentiate the content being presented (e.g., tourist information, transportation facilities, etc.) In addition, if transportation means (buses, trams) do have electronic location tracking (which predicts the arrival of the next vehicle) then tourist ACEs can choose the hopefully optimal path to the destination. Using similar techniques, one could also use the system for searching friends or profile-matching people in overcrowded places: mobile devices owned by each person could connect to a network (possibly ad-hoc) and to inject signals there to find persons and things.

- **Distributed Auctions**

Auctions are a class of negotiation protocols for allocating goods based upon competition among the interested parties.

Autonomic components are well-suited for dynamic, constrained and real-time environments such as electronic marketplaces. In such environments, components representing their customers negotiate for goods and services following negotiation protocols.

The idea at bottom line of the above scenarios is that the wide spectrum of application they contain is a reasonable guarantee that the models and abstractions we are going to develop in the project will be general enough and still application-driven.

4.2 Requirements

In this section we are going to derive requirements from the scenarios described above. First, general –foundational – requirements are listed and explained: they are the basic characteristics to consider in the ACE model. These requirements are the ones allowing ACEs to "live" in the selected scenarios. Then, we present those requirements involving advanced and autonomic ACEs functionalities that are the core of the other WPs. These requirements ensure that ACEs will be ready to integrate such functionalities in a coherent model.

The ACE model described in the next sections takes these requirements in considerations to build an effective model that will be suitable to address the challenges of the presented scenarios.

General (Foundational) Requirements

In the following table we identify the main general requirements for ACEs. Such requirements derive basically from the open, dynamic and heterogeneous scenarios that CASCADAS tries to address.

Scenario [3]	*Requirement*	*Motivation*
SESIL, BPCS	ACEs must be very lightweight with lightweight interfaces, suitable for lightweight devices	In pervasive scenarios the implementations of distributed communication services involve a plethora of devices (e.g., wireless sensors, PDAs laptops,etc.) The ACE architecture should fit for different extent to all of them.
SESIL, BPCS, DA	ACEs must support interoperability between the different levels of the ISO-OSI stack (network-level, the service-level, as well as the user level). ACEs should be capable of handling both users-level events as well as network-level and device-level events.	The goal of CASCADAS is to build autonomic communication services spanning different layers of the network stack. All the above scenarios require performing activities both at the network level (e.g., routing in sensor network in SESIL) and at the application level (e.g., supporting high-level policy in DA). ACE should be able to deal seamlessly with all of them.
SESIL, BPCS	ACEs must tolerate execution over unreliable devices and unreliable network links. ACEs should count with a dynamically, unpredictably and frequently changing network structure	All considered scenarios, but mainly SESIL and BPCS, consider possibly faulty devices interconnected with low bandwidth, unreliable networks links.
BPCS, DA	ACEs must support for dynamic and spontaneous aggregation and composition, even in absence of centralised control.	Mostly in BPCS and DA scenarios ACE's networks will be diffused over a large spatial scale in a highly decentralized fashion. Because of this and considering the fact that a central control is not feasible adequate tools to control, self configure and make secure ACEs ensemble are required (this is mainly a WP3 concern).

[3] **SESIL** = Smart Environment Supporting Independent Living, **BPCS** = Behavioural Pervasive Content Sharing, **DA** = Distributed Auctions.

Scenario [3]	*Requirement*	*Motivation*
SESIL, BPCS, DA	ACEs' aggregation model should support self-similarity, in which a group of ACEs can be accessed as a single entity. Support for transparent aggregation from the outside	ACEs should contemplate very large scale systems composed by a huge number of different devices in which services could be composed by a large number of components. In addition, software reuse and component-oriented design are fundamental principles to build complex, large-scale system such as those required by the presented scenarios. ACEs should be able to provide advanced composite services without changing ACE internal behaviour.
SESIL, BPCS, DA	ACEs should be able to communicate with each other in various means (point-to-point, any cast and multi cast, local multi cast, probability multi cast). ACEs should be able to implement both simple and stateless communication protocols (e.g., value queries), and complex and stateful protocols (e.g., negotiations).	The variety of the identified application scenarios and the possible applications within them requires that ACEs should be able to create and use a number of communication services. Such services should sustain and support the dynamic distributed scenario by providing high-level, powerful communication channels that are also robust and scalable.
BPCS, DA	ACEs do not necessarily have a clearly identifiable name/identifier, or a specific stakeholder, and must be able to interact in an anonymous way.	Mostly in BPCS and DA scenarios ACE's networks will be diffused over a large spatial scale in a high decentralized fashion. Because of this and considering that a central control/registry is not feasible due to the dynamism of the scenarios novel tools to control and enable interaction will be required.
SESIL, BPCS, DA	ACEs should support dynamic interfaces (i.e., should be able to dynamically adapt the provided functionalities). ACE should share a common ontology or be able to access services translating from one ontology to another.	A big challenge in open scenarios like the ones being considered relates to interoperability between heterogeneous components. This challenge involves both the syntactic level (interface definition) and the semantic one (meaning of an interface)
BPCS, DA	ACE should be able to include mechanisms to support the building and maintenance of various structures of overlay networks. ACE overlays must be	In large scale scenarios, interaction mechanisms must be tuned for specific application needs. Overlay networks are flexible and powerful mechanisms to tie

Scenario [3]	*Requirement*	*Motivation*
	lightweight and scalable.	together related components, while avoiding information overflow.
BPCS, DA	ACE should support mechanisms for dynamic life cycle management (grow, swarm, & shrink...)	In BPCS and DA scenarios, developers cannot easily foresee the amount of resources needed to achieve a task, or the load conditions upon which their components will have to operate. Thus, the implementation of any distributed services must consider to handle available resources in an autonomic and situation-aware way, by replicating its components on need, and have them start autonomously to increase the quality of services by properly exploiting the enlarged resource availability as a collective (i.e., as a swarm). On the opposite, whenever such resources appear excessive, the distributed swarm of ACEs should properly shrink itself to accommodate the new need.
BPCS, DA	ACEs should be able to provide services with various and tuneable QoS, and also should support for QoS evaluation	In dynamic scenarios, QoS must be constantly monitored and pre-emptive actions must be enforced to guarantee suitable levels of QoS. This would allow both to better comply with contracts, and also to optimize resource usage.

4.3 Requirements Involving ACE and WP2 (Pervasive Supervision)

The *pervasive supervision* work-package 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. To this end, ACEs have to provide suitable hooks to let controller ACEs to supervise their operations.

Scenario	*Requirement*	*Motivation*
SESIL, BPCS, DA	ACEs must provide hooks to let other components to monitor and supervise their behaviour. This can be achieved either by some kind of *reflective* operation, or by *aspects programming*, or by suitable *connector* components.	Pervasive supervision requires monitoring the ACEs behaviour and to possibly change it. This kind of supervision should be as transparent as possible from the ACE point of view, in order to advoid complexity and foster separation on concerns. In addition, a given supervision mechanism (e.g., controlling resource consumption) should be general and should be applicable to different ACE ensembles with minimal changes.
SESIL, BPCS, DA	ACEs can be asked to perform some operations to ease the supervision task. For example, ACEs can be asked to aggregate, filter and report the operations they are undertaking to the supervisors. This can require aspects, and/or mobile code.	Supervising large ACE ensembles with fine-grained logs (e.g., the trace of operation of every ACE) would be unfeasible because too complex. Aggregated and filtered information are required to enforce an effective high-level and robust supervision.

4.4 Requirements Involving ACE and WP3 (Self-organized Component Aggregation and Emergent System properties)

The *self-organized component aggregation and emergent system properties* work-package aims at identifying a repertoire of self-organized algorithm useful for a number of tasks. The inherent decentralized and large-scale nature of ACE applications will take advantage of these algorithms. To enable a fruitful integration of WP3 activities, it is important that the ACE model supports the implementation of self-organized and emergent algorithms.

Scenario	*Requirement*	*Motivation*
SESIL, BPCS, DA	ACEs have to be possibly lightweight as ant-based components.	A great number of self-organized/emergent algorithms is based on a huge number of tiny components doing something smart together. The ACE framework has to enable scenario with a huge number of tiny components without

Scenario	*Requirement*	*Motivation*
		incurring in scalability problems.
SESIL, BPCS, DA	ACEs have to interact in various ways: direct communication, epidemic communication,stigmergic[4] communication, and also simply by observing each other behaviour (i.e., BIC Behavioural Implicit Interaction). It is worth noticing that this last form of interaction presents strong analogies to the "observable" requirement required by WP2.	The key point in a number of swarm-intelligent and emergent algorithms is in mimic the way in which components (e.g., insects) interact with one another. To support the development of such kind of algorithms, it is thus fundamental that ACEs are able to interact in various kinds of ways.
SESIL, BPCS,DA	ACEs should have a concept of location ACEs should be able to organize in a overly network	A number of swarm and emergent algorithms are based on the location of the components according to some metric space ACEs should be able to link together to create cluster and groups as requested by swarm algorithms.
SESIL, BPCS, DA	Mechanisms to assess the similarity between ACEs need to be provided.	Mechanisms exploiting the similarity between components are involved in many swarm algorithms. In particular, these are needed to support the creation of WP3 calls clusters (i.e., groups of ACEs offering the same functionality) and reverse clustering (i.e., groups of ACEs offering different functionalities).
SESIL, BPCS, DA	ACEs should have proper places where to plug-in reconfiguration algorithms.	It is important to plug swarm algorithms in ACEs to enforce autonomic and self-organizing functionalities.

4.5 Requirements Involving ACE and WP4 (Security, Survivability and Self-Preservation)

The *security, survivability and self-preservation* work-package deals with security and self-healing mechanisms. This WP has many points in common with WP2, in that the mechanisms proposed by WP2 can naturally be applied for security purposes. However, the fact of controlling and modifying a running system cause privacy and security problems in turn. ACE should provide mechanisms to effectively enforce security constraints.

[4] stigmergy refers to communication by modification of the environment. It is an often observed strategy in emergent systems.

Scenario	*Requirement*	*Motivation*
SESIL, BPCS, DA	ACE communication has to be encrypted, and not all ACEs can interact freely with each other.	The goal of WP4 is to develop high-level and flexible protocols to enforce security constraints at the application level. The ACE architecture must support the creation of such services. To achieve this ACE has to create a secure foundation on which to enforce higher level policies. With this regard, the fundamental requirement is that ACEs communication cannot be eavesdropped.
SESIL, BPCS, DA	ACEs need some kind of unique identification.	A number of basic security issues require concepts such as reputation, trust and authorization. All of them require identifying the individual being involved.
SESIL, BPCS, DA	ACEs must provide secure hooks to let other components monitor and supervise their behaviour. This is fundamental to find security breaches and enact security policies. However, this must not create privacy issues (some ACE can contain sensible information) and must not open the way to malicious control threats.	Security, survivability and self-preservation issues implies an activity and a constraint the clashes with each other. On the one hand, it is important to be able to inspect the behaviour of ACEs looking for misbehaving components, and possibly update their functionalities. On the other hand, it is important to avoid that misbehaving controller tamper ACEs that are working correctly.

4.6 Requirements Involving ACE and WP5 (Knowledge Networks)

The *knowledge network* work-package aims at supporting ACEs with suitable knowledge to ease their application tasks. ACEs and knowledge network are deeply intertwined. On the one hand, ACE should be able to access knowledge networks efficiently for the sake of acquiring contextual information. On the other hand, knowledge networks will be constituted by ACEs in turn, thus ACEs must provide the proper hooks to support the development of knowledge networks.

Scenario	*Requirement*	*Motivation*
SESIL, BPCS, DA	ACEs are required to be lightweight. ACEs should be able to create an overlay networks linking to one another	The knowledge network will be made of ACEs each of them representing specific information. For this reason, it is very likely that the knowledge network will be constituted of a large number of these elements that, for scalability reasons, should be lightweight. Moreover, the

Scenario	*Requirement*	*Motivation*
		elements of the knowledge network are expected to be able to organize themselves in suitable knowledge networks to link related concepts together.
SESIL, BPCS, DA	Suitable mechanism to access the knowledge network should be defined (to be used by application ACE). Suitable interfaces to export knowledge information should be defined (to be used by knowledge network ACE).	ACE will access the knowledge network to achieve context awareness. Thus knowledge network-ACEs have to provide suitable interfaces to be accessed. In addition, application-ACEs must be able to access the knowledge network.
SESIL, BPCS, DA	Knowledge network-ACEs should be able to create mechanism enabling them to combine and aggregate higher-level knowledge. It is worth noticing that this requirement is similar to the filtering requirement asked by WP2.	A fundamental activity performed by knowledge networks will be data aggregation. A possibly large number of data sources can be aggregated to create a distilled summary. Such summary would be much more easily understandable by application ACE.
SESIL, BPCS, DA	Knowledge network-ACEs will have to propagate changing their content across the network to enable field-like and pheromone-like type of knowledge. It is worth noticing that this requirement is relevant in a number of self-organized algorithms (e.g., WP3). This may require code mobility.	Knowledge network ACE should support autonomic and swarm-intelligent algorithms. Since a number of these algorithms require "diffusing" messages (like in physical/chemical fields and pheromones), It could be important to have similar kind of mechanisms in knowledge networks.

5 State-of-Art

The investigation and development of a model for situation-aware communication and dynamically adaptable services is the main objective of the CASCADAS project. For that reason we will describe the idea of autonomic computing as a basic principle for self-managing computer systems. As an extension of autonomic computing we then present the state of the art in autonomic communication. Thereafter, current activities and results of projects thematically related to CASCADAS will be summarised to give an overview of ongoing research in relevant or adjacent scientific areas. This chapter closes with a brief description of component models related to the research performed in the CASCADAS project.

5.1 Related Projects

5.1.1 BIONETS

The goal of the BIONETS project is to provide a biologically-inspired open networking paradigm for the creation, dissemination, execution, and evolution of autonomic self-evolving services able to adapt to localised needs and conditions while ensuring the maintenance of a purposeful system. The project addresses problems in pervasive communication/computing environments characterised by an extremely large number of embedded devices. *Heterogeneity, Scalability and Complexity* have been identified as the three main challenges of such environments to the conventional networking approaches.

Heterogeneity

Heterogeneity results from the observation that there will be a huge differentiation in the devices of future ubiquitous networks. The BIONETS project distinguishes between two main device categories. On the one side there are complex portable devices with a large amount of processing power (e.g., laptops, PDAs, smart phones etc.), and on the other side miniaturised devices with sensing, identifying, and basic communication capabilities, surrounding us in everyday lives.

The heterogeneity issue has been addressed by introducing two-tier SOCS (Service Oriented Communication Systems) network architecture [Figure 2].The upper layer consist of so called U-Nodes (User Nodes) which are basically devices running services. U-Nodes may communicate among themselves and can communicate with T-Nodes. As depicted in Figure 2.The resulting network topology is an "archipelago of connected islands of nodes" ([5]). The lower layer consists of T-Nodes (tiny sensor nodes) which represent cheap tiny devices such as sensors, tags and RFIDs. T-Nodes do not communicate among themselves. They simply answer to poll messages sent by U-Nodes which are interested in getting the actual value of the random field they are sensing.

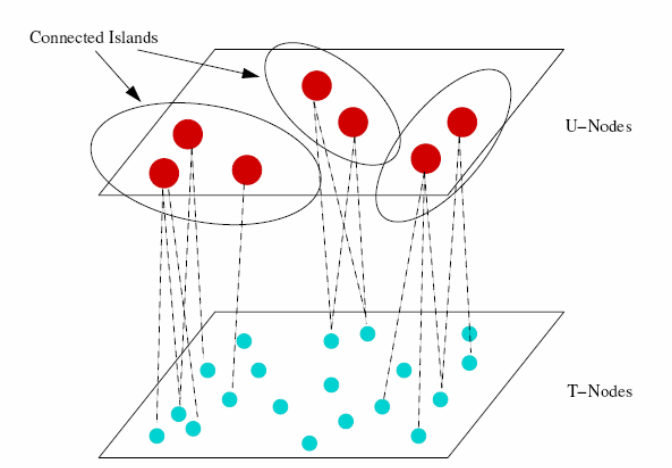


Figure 2 Two-tier SOCS network architecture [5]

Scalability

When applied to large scale wireless environments, the end-to-end communication paradigm which is typical for internet-based communication, suffers from huge scalability problems. The BIONETS project improves network scalability using two methods. The first method works at the U-Node level and exploits the mobility of devices to convey information. Information is exchanged either in a peer-to-peer manner through single hop broadcasting, and is diffused by local relaying on packets [6] or opportunistic exchange when mobile devices come into mutual communication range[7]. The second method exploits the locality of information coming from the environment, where data originated from sensors looses its usefulness as soon as they spread in both time and space domain. BIONETS introduces information filtering principles and mechanisms which target the scalability issues.

Complexity

Because of the dynamic nature of BIONETS network operations, the complexity issue which is related to the need of controlling and maintaining the network functionalities cannot be solved using conventional centralised solutions. Distributed mechanisms need to be introduced which are able to predict and control the behaviour of large scale complex heterogeneous systems. The BIONETS project follows the *adaptation by evolution* approach where a one-to-one mapping between biological entities and their technological counterparts is built, and introduces a framework for service evolution able to imitate what happens in the living world.

BIONETS is "a network that looks like a living ecosystem, where services play the role of organisms, evolving and combining them to successfully adapt to the environmental characteristics" [5].

The Information filtering approach might be very interesting for the CASCADAS project to improve network scalability and validity of information.

5.1.2 Autonomic Network Architecture (ANA)

The main goal of the ANA project is to explore novel way of organising and using networks beyond today's internet technologies. ANA aims at designing and developing an autonomic network architecture capable of autonomously arranging network nodes as well as whole networks. This novel network architecture should scale in time and functional way; that is, the network can extend both horizontally (i.e., add more functionality) as well as vertically (i.e., explore different ways of integrating abundant functionality) and change over time.

The target of the ANA project is to develop a functionally scaling self-aware network that builds up the basis for the evolving network which includes self-* features and functionalities such as self-management, self-monitoring, self-repair and self-protection.

One of the aspects analysed within this project, which could be of interest for CASCADAS, is the service migration problem.[26] In the miniaturised networks where traditionally heavy network elements (routers) are increasingly being supplemented by lighter network elements that are contributed by traditional network users, the problem of placing the

service to the proper network elements is seen as a significant problem. Considering the mobile ad hoc environment consisting of resource-limited mobile devices this problem becomes very clear.

The optimal service placement problem in ad hoc networks consisting of p nodes can be determined by formulating and solving the *p-median problem* [24].Unfortunately the p-median problem has been shown to be NP-Hard for general graphs [22], and therefore inappropriate for solving the service migration issue. In the ANA project a simplified migration policy is proposed for unidirectional tree topologies where a service can migrate only to the next neighbours. It is shown that the information available at the current node is sufficient for determining the direction towards nodes with monotonically decreasing cost. To decide on service movement, a service node simply needs to monitor and aggregate the data exchanged among its neighbour nodes associated with the concerned service. Hence, the movement decision is based exclusively on the information gathered through the monitoring process (cf. [26]). The service moves from node to node until it reaches the optimal service position.

5.1.3 Haggle

Haggle is an additional "Situated and Autonomic Communications" project which aims to solve the connectivity and networking problems in mobile ad hoc environments while introducing a new application-driven message forwarding approach. The project defines an innovative system that uses best-effort, context aware message forwarding between ubiquitous mobile devices, to provide services even when connectivity is local and intermittent. The Haggle approach is more oriented to the human way of communicating, rather than to other technological aspect of communication. It introduces a new autonomic communication paradigm, based on advanced local message forwarding and sensitive to realistic human mobility. It relies on a communication architecture that uses opportunistic message relaying, and is based on privacy, authentication, trust and advanced data handling [Figure 3**Errore. L'origine riferimento non è stata trovata.**].

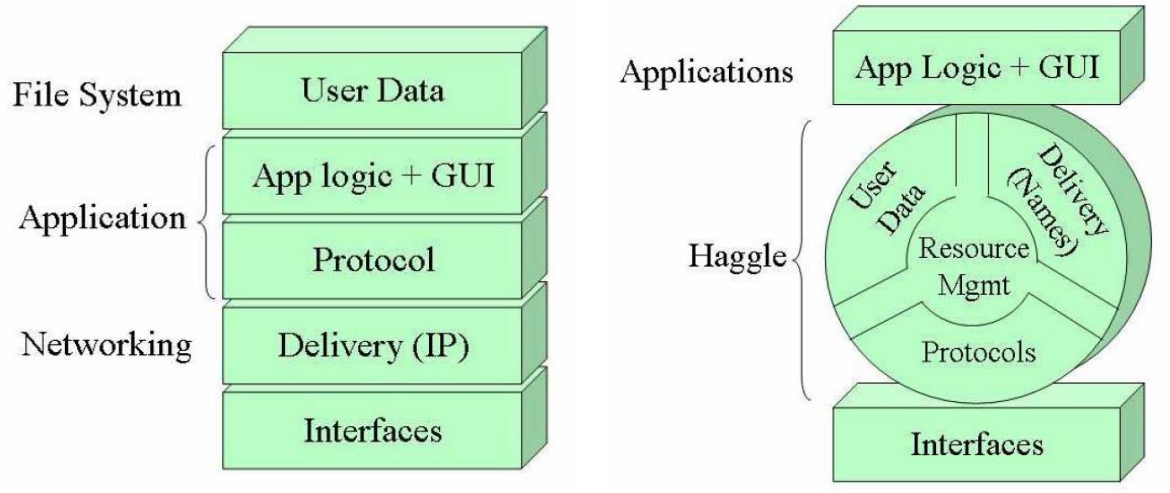


Figure 3 Current networking architecture vs. Haggle networking architecture [28]

The Haggle project introduces a *Pocket Switched Networking* (PSN) approach where two types of applications are defined: (a) *known-sender* where one node needs to transfer data to a user defined destination and (b) *known-recipient* where an application requires particular data.

The Haggle project focuses on new autonomic networking architectures, whereas CASCADAS focuses on situated services. Nevertheless there are some approaches introduced in Haggles mobile networking principles that can be used in CASCADAS. For example the *intermediate nodes approach*, where intermediate nodes keep the forwarded data when exchanging information, can be applied to our messaging approach as well.

5.1.4 AutoMate

"The overall objective of the AutoMate project is to investigate key technologies to enable development of autonomic Grid applications that are context aware as well as self-configuring, self-composing, self-optimising, and self-adapting" [3] .AutoMate focuses on autonomic components, the development of autonomic applications as dynamic composition of autonomic components, and the design of runtime services to support these applications. The AutoMate framework architecture is depicted in Figure 4

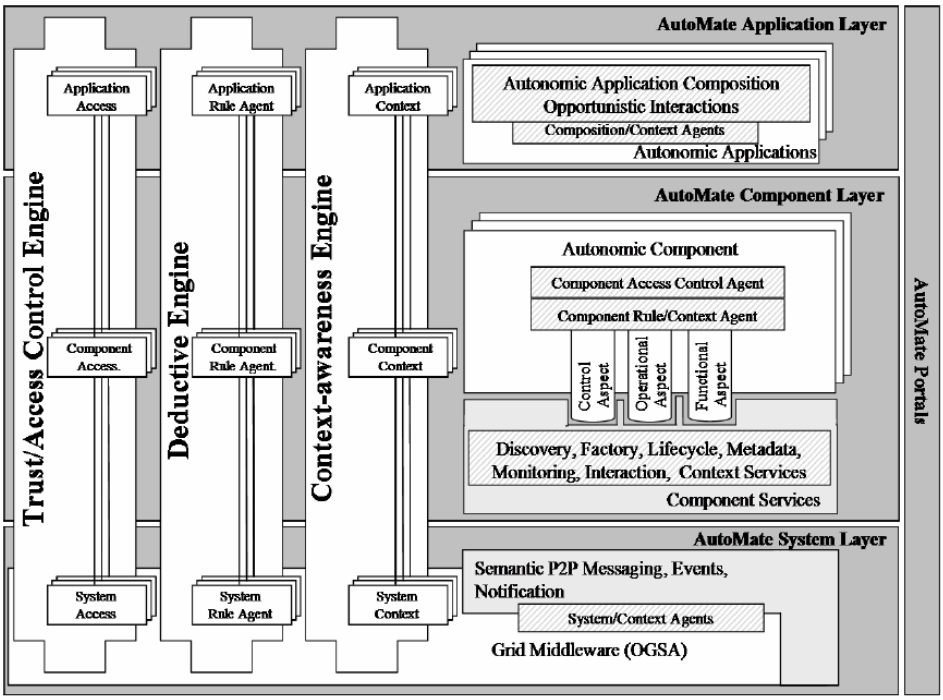


Figure 4 AutoMate Architecture Diagram [3]

As depicted in the figure above, the AutoMate framework builds on the Open Grid Service Architecture (OGSA) [31] and is composed of the following components:

- **Automate System Layer:** The AutoMate system layer builds on Grid middleware and OGSA. It extends core grid services to support autonomic behaviour. The system layer also provides specialised services like peer-to-peer semantic messaging, events and notification.
- **AutoMate Component Layer:** The AutoMate component layer defines autonomic components and comprises functionalities required for their execution and runtime management. Such functionalities are for example discovery, lifecycle management, context awareness etc. (which are built on core OGSA services).
- **AutoMate Application Layer:** The AutoMate application layer implements functionalities to support autonomic composition and dynamic interactions between components.
- **AutoMate Engines:** In order to support certain features like for example access control, inference and context awareness, AutoMate defines functionalities provided by so called engines. Engines are realised as decentralised networks of agents.
- **AutoMate Portals:** The AutoMate portals provide users with secure, pervasive access to the different entities of the AutoMate framework.

All AutoMate components exhibit information and policies about their behaviour, resource requirements, performance, interactivity and adaptability, so that this information can be used by other involved parties. A conceptual overview of an AutoMate component is presented in figure 5.

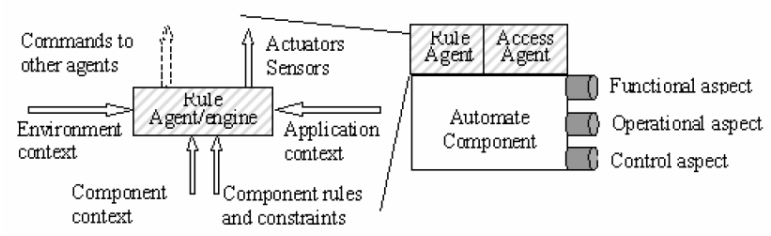


Figure 5 AutoMate Autonomic Component[3]

Each component is described though it's functional, operational and control aspects. *Functional aspects* describe component functionality and can be used by the compositional engine to select appropriate components based on application requirements. *Operational aspects* describe component operational properties including computational complexity and resource requirements, and can be used by the configuration and runtime engines to optimise component selection and adaptation. *Control aspects* describe the adaptability of a component and define sensors/actuators. Autonomic components encapsulate management-, interaction-, control-, and access-policies as well as rules, a rule agent, and an access agent.

AutoMate defines a dynamic service composition model that allows applications to adapt to dynamic system and environment changes. The Service composition model is context aware. It is based on policies and constraints that are defined as simple rules at runtime. These rules are executed on the distributed deductive engine. There is no central authority that manages the composition process.

Looking from the perspective of the CASCADAS project two aspects of the AutoMate framework are very interesting. On the one side the concept of the AutoMate components and component description covers aspects that are relevant for our ACE design. On the other side the autonomic service composition and the way this functionality is implemented in the AutoMate project, might be of interest in CASCADAS as well.

5.1.5 Cortex

The overall objective of the CORTEX project [20] [27] is to investigate the theoretical and engineering issues necessary to support the use of sentient objects in order to build large-scale proactive applications. A sentient object [27] is a mobile, intelligent software component that is able to sense its environment via sensors and react to sensed information via actuators.

The goal is to develop a programming model able to support the development of proactive applications constructed from mobile *sentient* objects. The programming model has to address any issues arising in environments built of networked components that will act autonomously in response to a myriad of events and which have to affect and control the surrounding environment in order to operate independently from the human control.

The key elements of the model are the following:

- *sentient object model*: providing the internal structure of the component built by its sensory capture, context awareness and intelligent interface.
- *event-based framework*: allowing the sentient objects to communicate each other and controlling messages propagation by proximity rule and content filtering. The main target of this specification is the need to address the requirements of applications running in mobile environments.
- *specification of QoS parameters*: which may be mapped to the system level.

Sentient objects are the basic building blocks of applications developed following the CORTEX programming model. This make such applications consisting of a very large number of mobile software components accepting input from the environment via a variety of sensors and autonomously acting upon the environment via a variety of actuators and cooperating using different network technologies.

The following picture shows a basic view of the sentient object model:

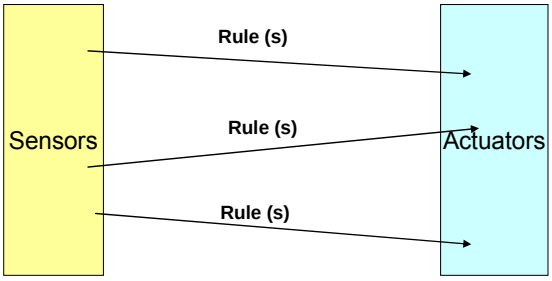


Figure 6 Basic view of the sentient object model

Sensors and actuators are the sentient object interface. Actuators are controlled by rules based on inference engine.

The following picture gives a more detailed view of the sentient objects organisation in a CORTEX model application.

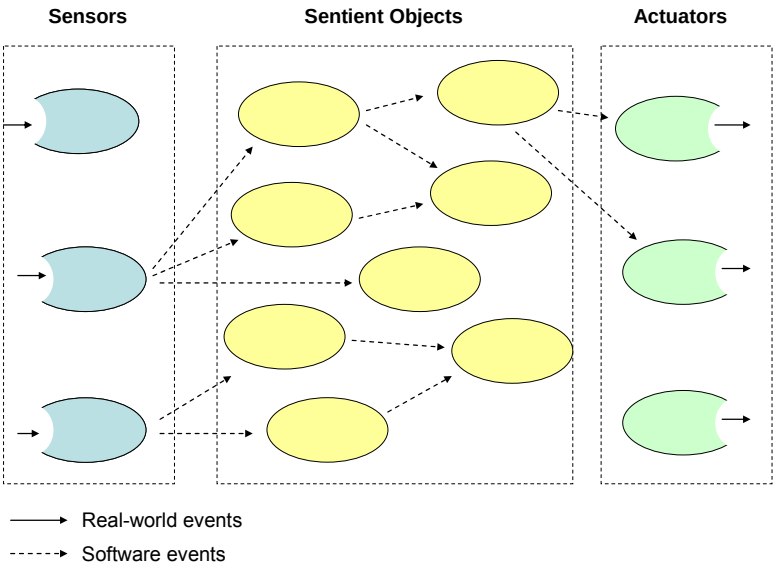


Figure 7 Sentient objects organization structure

We distinguish three major entities in the sentient object model:

- *Sensor*: entity that produces software events in reaction to a real-world stimulus detected by some world hardware device;
- *Actuator*: entity that consumes software events, and reacts by attempting to change the state of the real world in some way via some hardware device;
- *Sentient object*: entity that can both consume and produce software events, and lies in some control path between at least one sensor and one actuator.

The most important feature of a sentient object is that it implements the control logic. This control logic works on stimulus coming from the external environments. The importance of the external environment events and states make context-awareness a key factor for the sentient object.

CORTEX model defines context-awareness as:

"The use of context to provide information, to a sentient object, which may be used in its interactions with other sentient objects and/or the fulfilment of its goals." [33]

Three main components implement context-awareness in the sentient object:

- *Sensory Capture*: integrates the different events coming from sensors and filters them to limit noise and errors coming from the environment.
- *Context Representation*: transform raw data in a format that is useful for the sentient object.

- *Inference engine*: implements the reasoning capability of the sentient object which has to be able to take the appropriate decision based on the incoming inputs. Such engine is based on a knowledge-base built by a set of production rule (CORTEX adopts CLIPS [16] as declarative language to specify rules and integrates it with Context Based Reasoning mechanisms).

Coordination mechanisms implemented by sentient objects are based on interaction through the environment (stigmergy), inspired to the behaviour of colonies of insects.

CORTEX ad hoc network target imposes limitations to the design of the event service, considering the lack of a network infrastructure. The event model adopted by CORTEX is STEAM [23] which addresses a number of core issues for publish/subscribe framework. The key characteristic of the model is that it doesn't require any event broker: brokering functions are implemented both at consumer and producer side.

The key hypothesis which has driven STEAM model is that in a pervasive environment with high mobility, entities are most likely to interact if they are in close proximity. So the rule is: closer consumers are located to a producer the more likely they are to be interested in the events propagated by the producer. This rule limits the forwarding of the event messages, reducing the usage of the communication resources.

Event filters are the main tools to control the propagation of the messages. The novelty of the approach is that subject and proximity filters are applied at the producer side, whilst content filter are applied at consumer side. The significant advantage of this approach is that consumers haven't to forward content filter to producers when they change their geographical area. This simplifies the dynamic reconfiguration requirements as far as subscription and content filter is regarded.

The Sentient object, which represents the CORTEX main building block, seems to be very close to the ACE concept. Indeed, CORTEX sentient object faces some key issues for the ACE architecture success: the distribution of reasoning capability across the components and the adoption of a communication paradigm which both properly scales and ensures loosely coupled communication. On the other hand, ACE has to address wider objectives then the Sentient Object in order to meet CASCADAS goals. The sentient object is specifically conceived to provide the development environment to build proactive application in order to control the environment. We think it is only a part of the CASCADAS objectives; relevant aspects like the need for self-similarity, emergent collective behaviour, self-organisation and the like seems not be so pertinent in CORTEX.

5.1.6 Runes

RUNES [17] is a middleware supporting the development and the execution of component-based applications. The RUNES middleware leverages a small component-based infrastructure able to provide at runtime modularised and customisable services to be applied in the context of specific applications. The core entity in the RUNES model is the *component* that is defined as an encapsulated unit of functionality and deployment. Components foster a cross-layer approach to software development, in that each component can be in charge of activities belonging to different abstraction layers of a distributed application, ranging from operating-system layer to high-level user interfaces.

The basic architecture of RUNES is divided in two parts:

- A *Middleware Kernel* which is a run time reification of a simple well defined software component model.

- A set of *Component Framework* which provide a configurable and extendible set of application services.

The Middleware Kernel

The aim of middleware kernel is to provide the underlying methods for the managing of application components. All the operations provided by the middleware are accessed through an interface of a particular component present in every node, the *Capsule.* The main goal of the middleware and Capsule is to allow: *(i)* the dynamic loading/unloading of components into the system; *(ii)* the instantiation of components and (later on) the destruction of the instances created; *(iii)* the binding and unbinding of components, so that a component can access the methods of another component.

The Component Framework

RUNES defines a *Component Framework* (hereafter CF) to support the development of component-based applications. The CF aims to: *(i)* provide an intermediate abstraction between components and the distributed system; *(ii)* increase the understandability and maintainability of the system; *(iii)* support the developer during the creation and assembly of components; *(iv)* enable the use of lightweight components (plug-ins).

In more detail, RUNES provides a framework of components to support the following important services:

- **Reflection Services.** These services enable the representation (or meta-representation) of the system. Such a representation, expressed as a tree of objects, is machine-readable so that components can understand the system and perform useful operations, such as adding or removing components, or intercepting method calls to add behaviour to an existing system (e.g., logging and enforcing security policies).
- **Local OS Services.** These services aim to provide the set of services to realise an abstraction layer over operating system functionalities. Thanks to these services RUNES components have a unified abstraction on top of which it is possible to operate with OS-mechanisms ranging from the MAC layer up to the application layer.
- **Overlay Services.** In order to enable flexible communication patterns, an overlay network is often imposed on the underlying physical network. This overlay network may span each device in the system to support routing and communication activities. Overlay services support the creation and the maintenance of such networks.
- **Context and Location Sensing Services.** These services, based on monitoring the behaviour of suitable sensors, are used to provide high-level context information to the components.
- **Advertising and Discovery Services.** These services allow components to discover the functionalities provided by other components and advertise their own, in order to efficiently bind and interact with each other.
- **Logical Mobility Services:** In dynamic scenarios, it is very difficult to have all the application functionalities installed in each and every device since the beginning of the application. For these reasons, services are needed to disseminate new functionalities (code) in the system. This is the goal of Logical Mobility Services.

- **Coordination Services.** Coordination of activities is needed between components for a number of reasons: interaction, synchronisation, etc. The goal of Coordination Services is to provide an efficient coordination mechanism to support components activities.

On the basis of this brief presentation, it is rather easy to see that RUNES provides a lot of functionalities that are amenable for ACEs' definition. However, a number of important aspects defined in the CASCADAS project (e.g., ACEs' self-* and autonomic properties) are not properly addressed by RUNES and thus require further investigation.

5.2 Component Models

Component models define solution directions for dynamic handling of components. Component models consist of the abstract definition of the platform (the common high-level functionalities that components access, inter-component communication model, registration, etc.) and of the rules and regulations that components must comply with. The following section describes relevant existing component models from the viewpoint of the CASCADAS project.

5.2.1 JavaBeans and Enterprise JavaBeans (EJB)

JavaBeans technology is a component architecture, and JavaBeans are reusable software components [34] [35]. The vision of the JavaBeans technology is that independent software vendors (ISVs) offer their software components as standard JavaBeans and these beans can easily be integrated with other beans and into a new software.

JavaBeans are simple. Most elements of the JavaBeans specification are optional to use, only a few regulations are obligatory.

JavaBeans components are recommended to be of small or medium sized granularity (very big components are not recommended because of the configuration difficulties). JavaBeans components are portable – it is one of the main goals to provide platform neutral architecture. Individual Java Beans vary in the functionality they support, but the typical unifying features that distinguish a Java Bean are:

- Support for events as a simple communication metaphor than can be used to connect up beans.
- Support for properties, both for customisation and for programmatic use.
- Support for persistence, so that a customised bean stores it settings and state in a uniform way.
- When a bean is used in a builder application, it is recommended to
    - Support introspection so that a builder tool can analyse how a bean works.
    - Support customisation so that the user can easily customise its appearance and behaviour.

In the life-cycle of a JavaBean we distinguish between design-time and run-time. Design-time is when the component gets customised and gets integrated into a program. Run-time is when the program is executed.

The JavaBeans component platform is called container. Containers provide uniform, platform-independent (operating system independent), high-level API, including the "usual" Java Standard Edition API elements. On each OS platform, the high-level API is mapped into platform-specific calls, for example, on a Microsoft platform, the JavaBeans API is bridged through into COM (Component Object Model) and ActiveX. All JavaBeans containers must support: JDBC (Java Database Connectivity) for database access, CORBA IIOP (Internet Inter ORB Protocol) for CORBA object interoperability, and RMI (Remote Method Invocation) for Java remote method calls. Multithreading and internationalisation are also obligatory to support. Containers may include additional network access mechanisms and services.

JavaBeans are subject to the standard Java security model. A JavaBeans component is recommended to have "minimal assumptions".

Enterprise JavaBeans is an extension of the JavaBeans component model, where–besides other things–the containers support web technologies and transaction handling, too.

Enterprise JavaBeans (EJB) add additional functionality, relevant in enterprise computing, to JavaBeans. The EJB component platform (EJB container) includes automated persistence handling, transaction management, failure-safe operations, etc.[36]

Compared to the CASCADAS approach, EJB as well as JavaBeans do not provide strategies to tackle issues like e.g., optimal resource use (it's the responsibility of the container only), self-reflection or autonomic communication. EJBs do not offer any specific support for autonomic communication, e.g., adaptation, self-sustainability, self-healing (fault-tolerance is a property of the container) etc. Though, those issues are covered by the ACE model as it is researched in the CASCADAS project. Nevertheless, JavaBeans and EJB are successful component models widely used within the Java Community. Concepts that CASCADAS can borrow from EJBs are more related to aspects of enterprise computing, e.g., reliability, load balancing, security and persistency.

5.2.2 CORBA Component Model

The Common Object Request Broker Architecture (CORBA) Component Model, short CCM, is a general purpose component model for distributed computing systems, based on the CORBA middleware [37][38] . CORBA provides platform independent communication in distributed computing systems [38]. This is achieved by the standardised description of software (application) programming interfaces (API) by the Interface Description Language (IDL) and the transport protocol General Inter-Operation Protocol (GIOP).

The CORBA Component Model advances the concept of software in CORBA to the concept of components [37]. For this purpose, CCM introduces the terms Basic Component and Extended Component. Basic components are made of attributes and the equivalent interface. Attributes are used to write and read the component configuration. The equivalent interface represents the functionality of the component, i.e., its API. In addition to basic components, extended components are made of facets, receptacles and event source and sink. A facet is a single aspect of the component's API, i.e., a particular functionality implemented in the component. The sum of all facets results in the equivalent interface. Receptacles allow other components to "plug into" the component to be notified on events. The event source publishes defined events, whereas the event sink consumes them.

CCM supports platform independent, loosely-coupled communication between components.

CASCADAS goes beyond CCM by introducing an ACE model capable of supporting autonomic features like situation awareness, self-organisation, aggregation etc. Concepts that CASCADAS can borrow from CCM are the communication principles of components, i.e., attributes, facets, receptacles and event sources and sinks, and the concepts envisioned for platform independent communication.

6 ACE component model

This section describes how the goals of the ACE component definition, the requirements collected and the partners' agreement has generated a conceptual and functional design of the ACE.

6.1 The ACE conceptual model

The conceptual model describes the results of the WP1 partners' discussions about the main concepts that an ACE has to cover and support in order to implement the autonomic communication principles as described in section 2. In Figure 8 it is depicted a conceptual model of the ACE to fix all the architectural aspects the model of the ACE has to deal with.

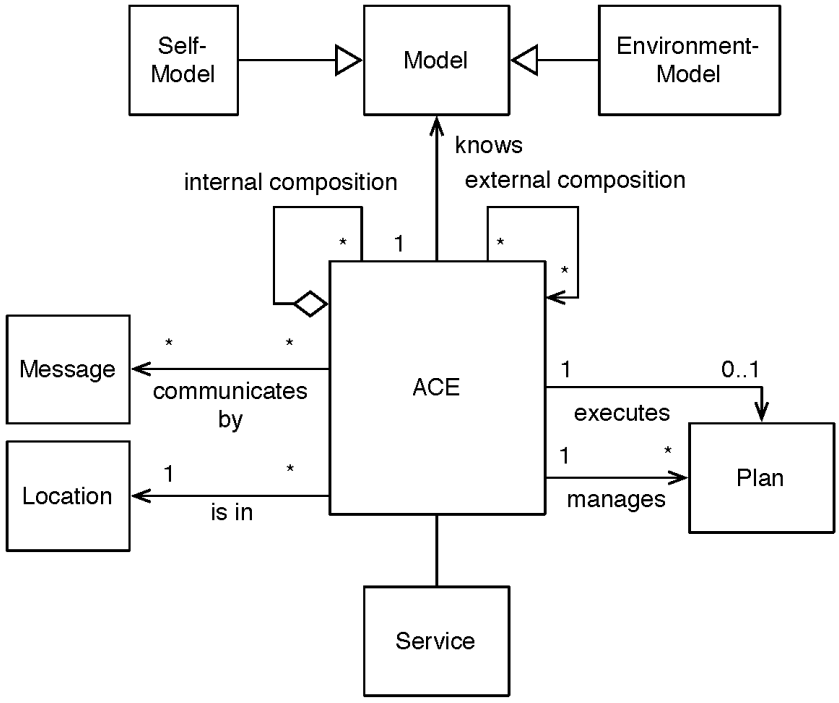


Figure 8 Conceptual UML diagram of the ACE base model

As seen in this diagram, ACEs may be composed in two different ways: externally and internally. Externally composed ACEs collaborate loosely to provide a service, while every single ACE remains visible as an independent entity. Internal composition aims at providing a service by mutual collaboration of ACEs through binding them more strictly: a new ACE emerges that is representing the whole ensemble and ACEs which can be found to be "within" this emerging ACE (in this section the emerging ACE is referenced as *combined* ACE and all aggregated ACEs as *contained* ACEs).

There are at least two models an ACE knows. On the one hand a self-model, representing its own context-dependent behaviour and caring for self-awareness; on the other hand an environment model, used for situation-awareness. The environment model may be the set of available self-models from other ACEs in combination with any other information communicated by the knowledge network. If during the course of the CASCADAS project, it turns out that other ACEs solely generate context information, these two models may collapse into a single one. This seems to be happening in the case of internal composition: the self-model of a combined ACE seems to be the environment model of all contained ACEs.

Communication between ACEs is message based, facilitating a time and spatial decoupling of the communication partners, for example by following mechanisms as surveyed by the authors of [11]. Messages may be buffered when their receivers are unavailable, they could be multiplexed for point-to-multipoint communication, they may be routed for indirect communication between ACEs, and they can be delivered asynchronously avoiding the requirement for synchronisation among ACEs. Messaging may include semantic routing which means that a recipient may be addressed via its semantic properties instead of a logical or physical address.

Every ACE resides in a location, which marks a position within the knowledge network, giving it access to the stigmergic information within that area. ACEs are supposed to be able to move between locations and are free to migrate to any place where at least one ACE is already in existence. The purpose of moving ACEs between locations is to optimise resource use. Examples are that ACEs may move closer to their clients to reduce communication costs, or to locations where they might be executed more efficiently to increase service quality. ACEs decide autonomously when and where to move, consequently they might have other reasons to move other than the ones given above.

Furthermore, we have identified the notion of a *Plan*, which is an explicit formulation of the way an ACE is supposed to act. ACEs are not only executing plans (which in turn dictates their behaviour), but also managing plans by creating, choosing, changing, rearranging, and removing them and thereby effectively changing their potential behaviour. The planning concept promises to enable adaptation and self-organisation capabilities.

6.2 The ACE functional model

This section points out how the concepts described in the previous section are mapped on to the main functional blocks contained in an ACE.

ACEs are structured in two parts. A common part is exposed through a so-called "common interface", which needs to be offered by every ACE and serves as the basic mechanism for enabling self-similarity and a specific part, available through a "specific interface" that contains the specific functionality of an ACE. This concept is influenced by previous work of some partners as published in [12]. As we do not expect every ACE to implement the common functionality by itself, a mechanism has to be researched that is able to relay

access of an interface transparently to another interface (e.g., by propagating functionality binding information like an updated service access point location from one ACE to another or by offering a proxy role and relaying messages). This will be of importance when migrating ACEs, using internal composition or creating ACEs based on other ones. A convenient way of creating ACEs would be the cloning of existing ACEs, resulting in inheritance of all common bindings from the parent ACE.

The Figure 9 shows the proposal for an ACE structure: every element will be described and detailed in the following sections.

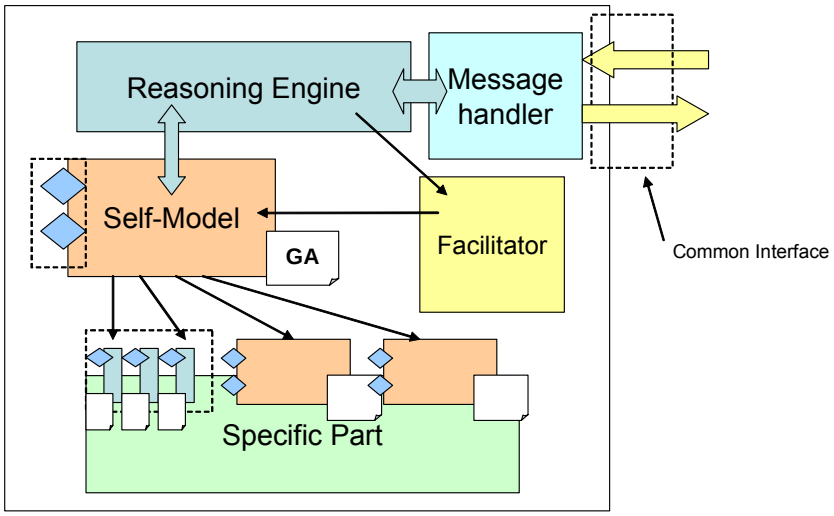


Figure 9 ACE structure

For a better understanding, Figure 10 lists the meaning of the graphical symbols used in Figure 9

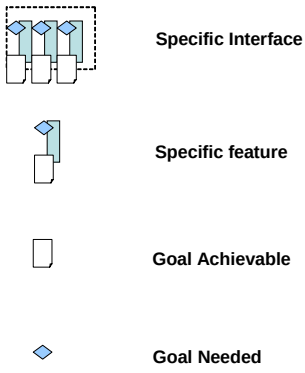


Figure 10 ACE structure legenda

The following table summarizes the correlation of the functional blocks defined in Figure 9 with the concepts described in the ACE conceptual model:

Concept	Architectural element	Description
ACE	ACE	The ACE concept is implemented as a basic component which contains all the elements needed to

Concept	Architectural element	Description
		implement the ACE features.
Message	Message Handler	Messages exchanged in a given format (e.g. XML) which carry the information needed to implement self-organization and other group features among ACEs. The MessageHandler is the ACE element capable to parse the incoming messages/outcoming messages, forwarding them to the proper ACE element.
Behaviour	GN/GA	A formal semantic description (e.g. xml) of the goal the ACE is able to achieve and of the goal the ACE need in order to achieve its own goals.
Model - Self-Model	Self-Model	A representation of the ACE behaviour that comprises the way in which features of the specific part have to be called to achieve the goal.
Model - Environment	Reasoning Engine/Facilitator	The environment is represented in the internal state of ACE contained in the Reasoning Engine component. The Facilitator, defines alternatives to the basic plan in order to face changing conditions.
Plan	Self-Model	The Self-Model describes the plan to achieve a given goal by the ACE.

6.3 The Common Interface

The Common Interface is basically the way ACEs communicate and interact with the world outside (i.e., other ACEs or the environment). The communication is message-based and then the Common Interface is implemented by a Message Handler which is able to understand a fixed set of messages. The ACE collaborations and aggregations are exclusively carried out by the exchange of these messages.

The basic set of messages addressed by the Common interface is:

- Goal needed (GN): a sort of request, with a semantic description attached, which specifies what kind of functionalities the ACE needs from other ACEs, to achieve its goals.

- Goal Achievable (GA): this message is used by an ACE to state what kind of task it is able to provide. It has also a semantic description and typically will be used to communicate to other ACEs what an ACE is able to do.

The figure 11 below is a simple example of a call-flow with a proposal sent by ACE1 (i.e., GA) to a certain number of ACEs and the dialog between the ACE1 and an ACE receiver (e.g., ACE2) which needs the advertised features of the ACE1.The ACE2 discovers a semantic matching of a received goal-achievable with its goal-needed, it sends back a kind of acknowledgment to the ACE1.

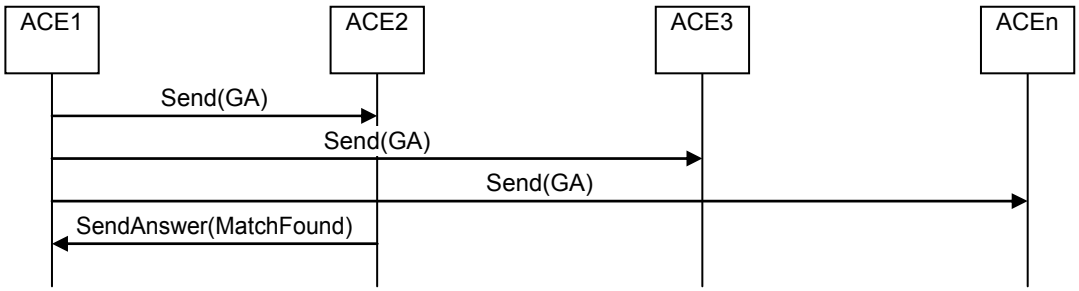


Figure 11 Call-flow

The GN-GA protocol does not generate a flood of GN request to all the ACEs. The GN-GA protocol is a semantic advertisement protocol by which ACEs advertise their capabilities through the GA message. One solution to limit the flooding overhead is the implementation of a sort of semantic time-to-live: if the incoming GA (i.e., the features described) belongs to the same semantic domain of the receiver ACE then the GA is propagated otherwise it is discarded as it is possible that ACE's neighbours are not interested in that GA as well.

The mechanism described above is based on the following main assumption: if the GA received doesn't pertain to the semantic domain of the receiver ACE, it would be high probable that neither the receiver's neighbours are interested in that GA, so in most cases it is better not to forward the message.

The semantic domain is defined by all the ACES where the GA/GN matching is satisfied.

6.4 The Specific Part

The Specific Part contains the ACE specific functions. It exposes these functions through the Specific Interface.

The Specific Interface contains a description of each function of the ACE Specific Part: the set of features which characterize the ACE behaviour. For each feature a semantic description of the job the ACE is able to do (GA) and the indispensable and essential actions and conditions to accomplish it (GN) are specified.

For example, given a specific function which executes a query on a DB containing personal data information, the semantic description of the GA could be something like: "get people profile" and the GN could be something like: "a connection with a database is necessary".

Besides the two main component of an ACE as explained above we envision some additional elements necessary to support the ACE as described in the following sections.

6.5 The Self Model

The Self-Model describes the possible states for the ACE and the possible transitions between pairs of states. In other terms, it could be defined as a state machine. Therefore the Self-Model is a description of the steps the ACE will execute to achieve its goals.

Any state is described by a semantic description used by the ACE to reason about its current state with the help of the Reasoning Engine.

The transition functions are the specific features which must be invoked during a state transition.

The ACE Self-Model is published to the outside world using the GN – GA protocol i.e., a semantic advertisement protocol by which ACEs advertise their capabilities.

6.6 The Reasoning Engine

The reasoning engine executes (the implementation of) the self-model and its main role is to keep trace of:

- The state reached in the Self-Model execution. Eventually, it may take trace of the history, storing the previous states.
- The environment: any GA coming from other ACEs.

Mainly, it has to be able to run the state machine representing the self-model. It should check if a transition may take place invoking the proper specific features if specified, and it has to properly represent the semantic description of the new state reached.

Briefly the reasoning engine:

- Tracks the Environment Model
- Runs the Self-Model.

6.7 The Facilitator

The Facilitator will be the core autonomic part of the ACE, adapting its behaviour to the changing conditions, situations or faults.

The adaptation of the behaviour means changing the self-model state machine when a specific feature exhibits different behaviour depending on its state.

In order to achieve this, the self-model "developer", or any autonomic mechanism should insert in the original self-model some additional transition to adapt the ACE behaviour. We call such additional transition *Join Point* (JP) added on the fly to modify dynamically the behaviour of the self-model.

The activity of the Facilitator could be summarized as follows:

- The Reasoning Engine will trigger any changes perceived in the environment or in the ACE's internal state (self-model) to the Facilitator.
- The Facilitator is able to identify the proper action i.e., JPs in order to adapt the self-model to the sensed environment modification.

The following picture shows a simple template based example implementing the previous schema:

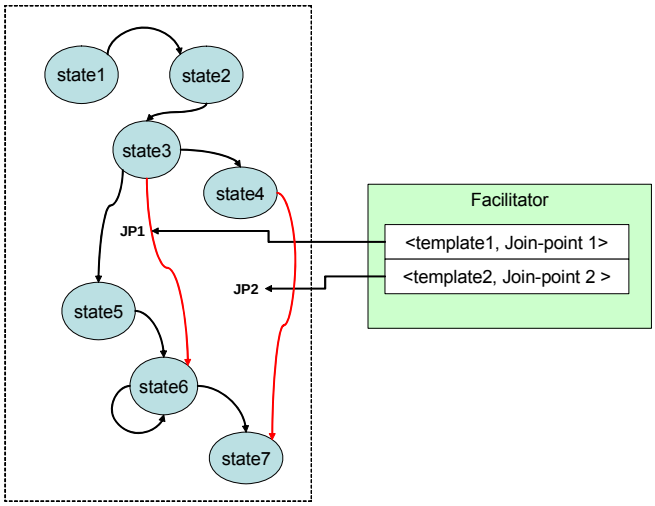


Figure 12 Schema of the facilitator interaction mechanisms

For example, let us have a self-model state machine with 7 states (figure 12) and a facilitator configuration with a change condition defined when state 3 is reached. If template 1 is matched, the JP1 is activated and the self-model is modified (from state 3 to state 6 skipping state 5).

The complex situation what we have in mind is quite close an inference engine to face any problem arisen during the ACE activity.

6.8 Example1: ACE Personal

This section is devoted to an example to explain the ideas behind the ACE components.

Let us suppose that we have an ACE, called ACE Personal, running on a mobile device (e.g., PDA) whose main task is to collect data from the user's mobile when she/he comes in a specific geographical area. The ACE Personal starts:

- Collecting data when a specific SMS is received by the mobile device.
- Advertising other ACEs interested in personal data when the user comes into a specific area.

The GA/GN messages for the ACE personal data are:

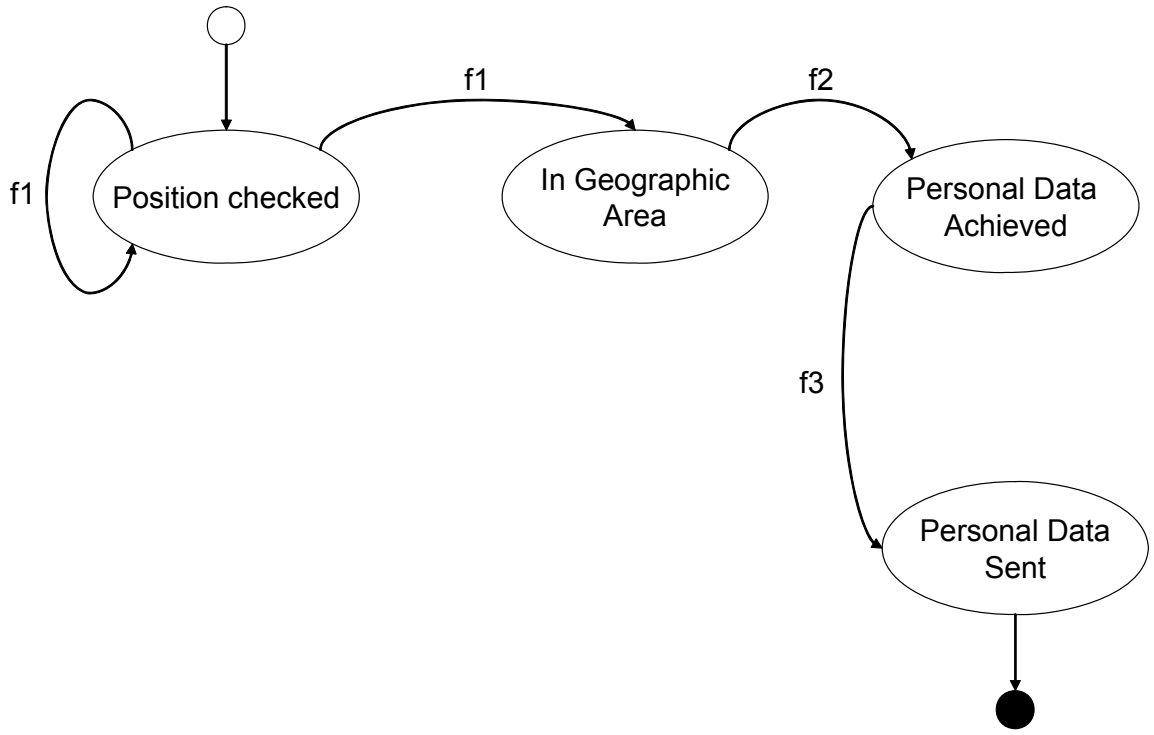
GN: SMS received from a Service Provider Localization System

GA: get Personal Data.

We do not cover the case when the user goes out from the interested area. In that case, an additional message should be sent by the ACE, advertising that the personal data of the user is no longer valid. Obviously, in a real world such a situation should be monitored and managed in a proper way.

6.8.1 Self-Model

The following picture shows a representation of the Self-Model for the ACE Personal:



This example doesn't contain any implementation details. The semantic description is given in a natural language and the self-model states are described with simple tuples

State	State desc.	Description
Position Checked	<Position, Lon, Lat, Reached>	The ACE has checked its position (location)
In Geographic Area	<Area, X,Y,Z,W, In>	When the ACE is in the geographic area covered by the Ads Screen service it receives a star-up sms message.
Personal Data Achieved	<Data, Personal, Age, Achieved> <Data, Personal, Gender, Achieved>	The ACE has read data from the handset device.
Personal Data Sent	<Data, Personal, Age, Sent> <Data, Personal, Gender, Sent>	The ACE has sent the personal data to interested ACEs (ACE population aggregator).

6.8.2 Specific Part and Specific Interface

The following table describes all the specific functions contained in the specific part of the ACE and referenced in the Self-Model above:

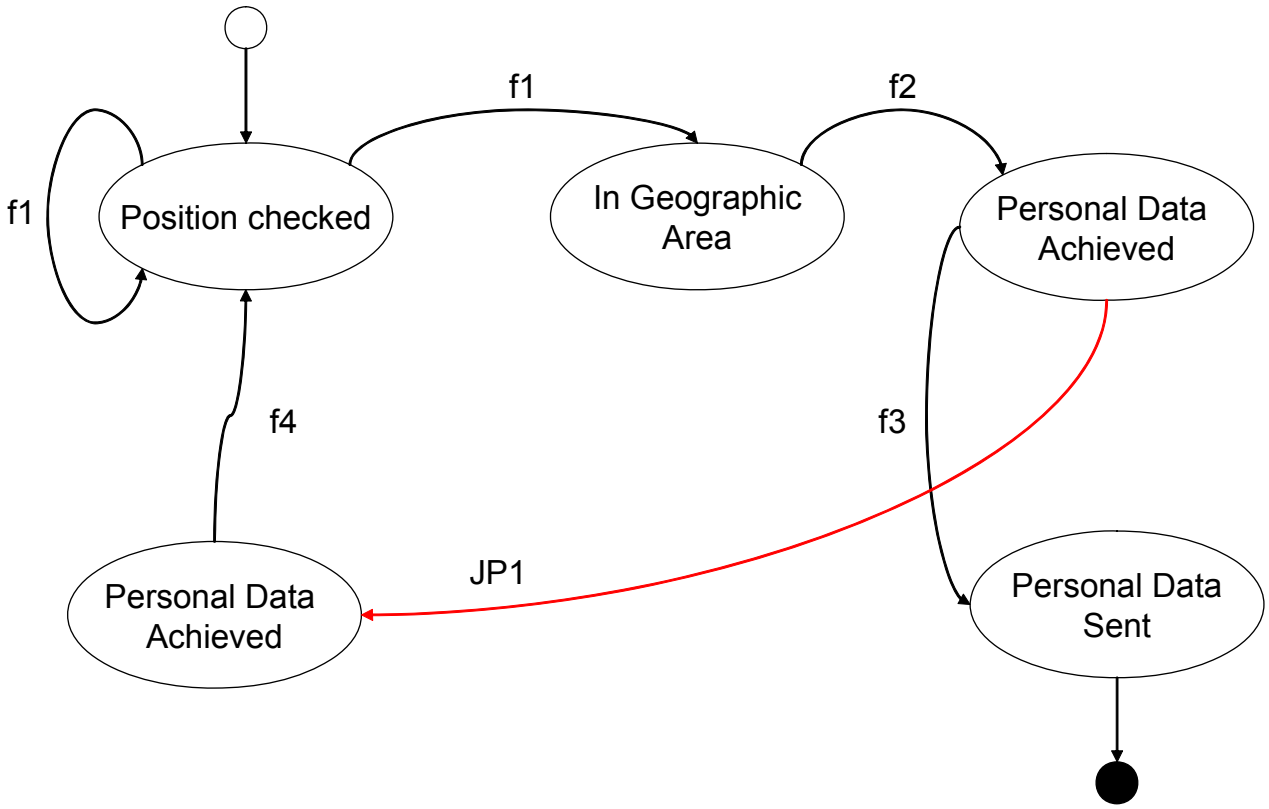
ID	Name	Description	GN	GA
F1	checkPosition	Poll the location system through the handset APIs in order to understand its geographic position.	F1.GN1: Location-based APIs installed on the handset F1.GN2: UMTS coverage	Position X,Y is Reached
F2	getPersonalData	Retrieve the personal data from the handset device.	F2.GN1: Personal data management APIs installed on the handset F2.GN2: personal data store enabled;	Personal Data retrieved
F3	sendPersonalData	Send the personal data to the interested ACEs.	None	Personal Data Sent
F4	resetData	Reset the data collected in the previous execution	None	Data reset

The Facilitator should be equipped to face any exceptional case arisen during the execution: for example when the user turns off the device before the ACE sends the data or the user goes out from the specific area. The Facilitator should recognize the situation described and it has to modify the self-model to avoid, in this case, any data inconsistency. In order to recognize the special situation described, the facilitator should monitor the states held by the reasoning engine and has to do a matching of such states on its templates. The states of the Self Model are recorded in the reasoning engine. So, conditions as:

- the handset is turned off and the user is still in the specific geographic area,
- the handset is turned on and the user is out of the geographic area

should be recognized and managed by the facilitator.

The following picture shows the modified Self-Model needed to adapt to the situation above:



The Facilitator should be configured in the following way:

Template	Self-Model Join Point
<Handset, turned off> <Data, Personal, *, Achieved> <Position, Lon, Lat, Not Reached>	JP1

7 Supportive Technologies

The objective of this section is to elaborate about the technologies that might be adopted for developing the ACE architectural model that is described in chapter 5. This section has been structured in sub-sections per each block of the ACE architectural model where the potential implementing technologies are described.

7.1 Inter-ACE communication

As CASCADAS is focusing on highly distributed and dynamically changing networks, inter-ACE communication is built exclusively on *message-based communication*. The basic building blocks of communication flow are Messages, which can be transmitted using various types of low-level protocols, transparently to the ACE. On the technical level, the Message Handler implementation realizes and carries out concerning low-level tasks.

The Message Handler provides high-level interfaces for sending and receiving messages, supporting several addressing schemes and reliability models.

7.1.1 Message Format

Messages are XML documents in order to support interoperability and independent implementation. Messages are defined in the name space "**CASCADAS/ACE**" and have to adhere to a specific message format as detailed in the following XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:ace="CASCADAS/ACE" xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="CASCADAS/ACE">

   <!-- Base type for addresses -->
   <xs:complexType name="Address">
<xs:complexContent>
        <xs:extension base="xs:anyType"/>
      </xs:complexContent>
   </xs:complexType>

   <!-- Base type for messages -->
   <xs:complexType name="Message">
      <xs:sequence>
         <xs:element name="Destination" type="ace:Address" minOccurs="0"
maxOccurs="unbounded"/>
         <xs:element name="Source" type="ace:Address" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="reliable" type="xs:boolean" default="false"/>
   </xs:complexType>

   <!-- Example Text  Message -->
   <xs:element name="TextMessage">
      <xs:complexType>
         <xs:complexContent>
            <xs:extension base="ace:Message">
               <xs:sequence>
                  <xs:element name="Text" type="xs:string"/>
               </xs:sequence>
            </xs:extension>
         </xs:complexContent>
      </xs:complexType>
   </xs:element>

   <!-- Further message types follow here -->

</xs:schema>
```

As the XML Schema syntax is rather lengthy and counter-intuitive, please consider the picture 13 for an explanation of the structure's specification.

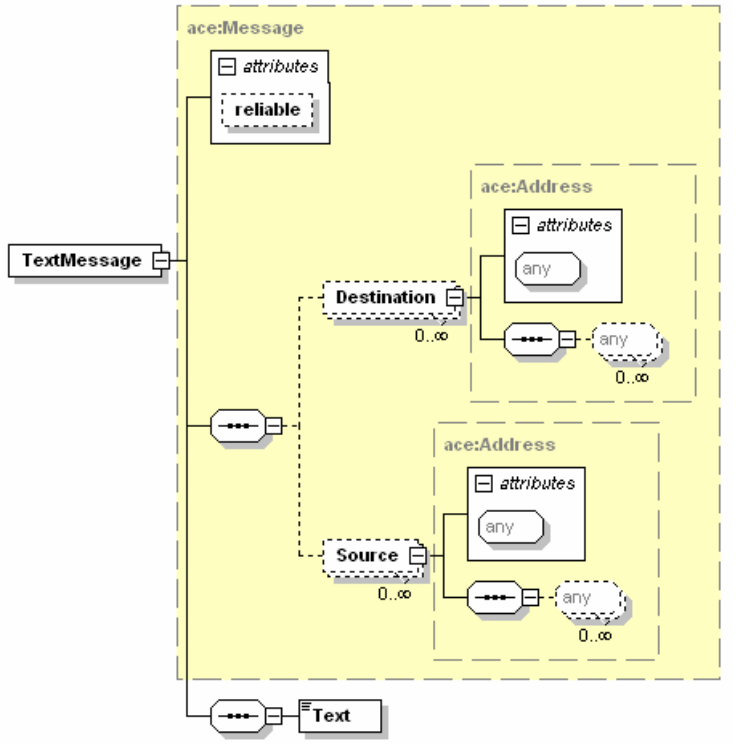


Figure 13 Structure of an `ace:TextMessage`

Any message processed by a Message Handler extends the basic `ace:Message` type. The same is valid for Addresses; they are deriving from the `ace:Address` type. A fictitious `ace:TextMessage` type has also been declared to demonstrate how the basic messages types are supposed to be extended.

Minimally an XML Message consists of the following:

- A mandatory message root element. It is fully qualified name defines the unique type of the message.
- An optional "**`reliable`**" attribute which requires the Message Handler to use a reliable transport mechanism.
- 0..* "**`Destination`**" tags referring to addresses where the message should be sent. Message Handlers may use optimisations when transmitting (e.g., if the given list of destinations is identical to a set of participants in a multicast communication group, the Message Handler might send the message only once using multicast). If no "**`Destination`**" tags are found in the XML message the Message Handler tries to reach as many other ACEs as possible (e.g., broadcast). Different addressing schemes may be used.
- 0..* "**`Source`**" tags referring to addresses the message originated from. As a single sender might have multiple addresses (aliases) or a group of senders might cooperate to send a certain message, we allow for several source addresses. It is also possible that no source address is given; in this case the sender might want to remain anonymous or has no identity of its own.

- Any number of other tags depending on the type of message. The exact tag/attribute names must be specified in the XML Schema for a message type.

The **`Message`** **`root`**, **`reliable`**, **`Destination`** and **`Source`** tags/attributes of the Message are often referred to as the "header fields" even if they are not explicitly wrapped by a "`header`" tag. Using an explicit "`header`" tag would not be of any advantage, as data in structured XML does not need to use explicit delimiters (as in e.g., frame or packet based protocols) but would only increase the size of the message and slow down processing.

Following is an exemplary text message that would be sending to two destinations, on the one hand a given IPv4 address and on the other a single destination out of a group of aliases. The source of the message would be known as the alias "`Home`" and the payload is a string.

```
<?xml version="1.0" encoding="UTF-8"?>
<ace:TextMessage xmlns:ace="CASCADAS/ACE" reliable="false">
   <Destination>
      <IP version="4" host="example.com" port="4711"/>
   </Destination>
   <Destination>
      <OneOf>
         <Alias>foo</Alias>
         <Alias>bar</Alias>
      </OneOf>
   </Destination>
   <Source>
      <Alias>Home</Alias>
   </Source>
   <Text>Hello World!</Text>
</ace:TextMessage>
```

7.1.2 Message Handler

The Message Handler is responsible for sending and receiving messages.

- The Message Handler accepts Messages from the environment and hands them to the Reasoning Engine.
- The Message Handler takes over Messages from the Reasoning Engine and delivers them to other ACEs.

The way the Message Handler realizes the former tasks is defined on the high level only; we will not prescribe or specify certain protocols to use.

For better clarity, we shall differentiate between the abstract Message Handler and the concrete Message Handler Implementation. The abstraction describes high-level functionalities and workflow while the implementation is responsible for the mapping to the actual implementation. The Message Handler Implementation is responsible for the address resolution and for the selection of the appropriate transport protocol.

If the Message Handler is not able to fulfil a request; an error is reported (e.g., a message that is marked as "reliable" is given to a Message Handler that is only able to transport messages in an unreliable fashion).

Message Handlers shall use the header fields[5] of the message and must not touch (modify, resolve, copy, etc.) other parts. Message Handlers may add extra fields to the header – dedicated for inter-Message-Handler purposes only (e.g., stamps, statistics) – but these fields shall be removed before handing the Message to the recipient.

Message Handlers are not used as an abstraction of an overlay network. Overlay networks shall be built up from ACEs and not from Message Handlers.

The Message Handler specification has mandatory and optional parts.

Message Handlers must support the followings:

- Non-reliable message sending.
- Takeover of a Message from another Message Handler.
- Check whether an incoming Message matches the ACE (recipient comparison) and hand the Message to the ACE.
- Report errors in case of non-supported requests.

Message Handlers may support the followings:

- Reliable message sending.
- Transactions.
- QoS.
- Security.
- Other functionalities.

The **Message Handler Implementation must** specify the followings:

- The protocol for delivering the Message to the recipient.

7.1.3 Addressing schemes

ACE communication must ensure the possibility for both anonymous communication and addressing-based communication. We presume to have an ACE ID (ACE identifier) which is not defined more closely (it may be unique or non-unique, may be a name, an alias, group identifier or a set of attributes/the self-model, etc.) It is assumed that the ACE knows its ID(s).

So far, the following addressing schemes have been defined:

Broadcasting. Message is delivered to as many ACEs as possible. Missing recipient tag indicates broadcast.

Recipient (list). The recipient(s) are listed (e.g., via their Aliases or IDs). Only those ACEs matching the recipient(s) will receive the message.

NEARBY. Only the nearby ACEs receive the message. In this case, we make use of the topology of the network. The semantics of this addressing scheme also matches the

[5] "field" may mean tag or attribute

principle of locality (e.g., for pervasive services). Practically, "nearby" may mean the 1-hop paradigm or a similar extension (e.g., 2-hops, 3-hops).

OneOf (list). One of the ACEs from the recipients list will receive the message.

The addresses used in the Message can be compared with the ACE ID, and comparison results in matching or non-matching.

The result of the comparison depends on the actual compositional state of the ACE, e.g., tightly composed ACEs are not allowed to receive Messages directly from the outside world.

Normally, the address used in the message matches the ACE if:

- the message is a broadcast message;
- the message is sent to the NEARBY ACEs;

at least one of the listed recipients matches at least one of the own Ids.

The addressing schema will be object of deep investigation in the next phase of the project.

7.1.4 Message Types

The exact list of message types will be defined in the next phase of the project. Messages will be based on the GN-GA conceptual model (or may extend it)

7.1.5 Communication flow

From the theoretical viewpoint, the interaction of ACEs (such as service usage, composition, etc.), can be divided into three phases:

1. *Discovery*: the parties locate each other
2. *Contracting*: the parties agree on the conditions of the interaction
3. *Interaction:* the real interaction

Phases may be **explicit** or **implicit**.

In simple cases, two messages are enough for the whole interaction process (a question broadcasted in a GN and the answer sent back in a GA). This is a good example for implicit phases: the first message implicitly includes the discovery, a null-contract and the first step of the real interaction, while the second message implicitly acknowledges the null-contract (by answering the question) and completes the interaction phase.

In complex cases (e.g., such as in pervasive supervision) explicit contracting is required to ensure that the parties bound themselves to the explicitly specified points (e.g., the ACE gives access to its Message Handler for the Supervision ACE). Contracting may become more important in case of non-free services (charging).

7.2 Reasoning Engine

The reasoning engine is an automaton that executes the Self-Model, based on the received messages. Context information that enabled ACEs to show context-sensitive behaviour is also tracked by the reasoning engine.

The basic operation of the reasoning engine could be depicted as an infinite loop that can be described with the following meta-code.

```
loop
   receive message
   determine possible transitions in the self-model
   choose transition(s) to be performed
   perform transition(s) & execute the side-effects(s)
end loop
```

The way a message is received and processed raises questions regarding parallel execution, synchronization and queuing. Regarding the message sources, not only external sources are possible but also internal ones (because of proactive behaviour and ability for self-configuration).

A matching operation is needed on order to determine the effect[6] of the Message on the Self-Model. As the matching operation is understood to be intrinsic to the self-model (or rather to the meta-self-model) model, that is why it is not examined in details here.

Selecting that transition (or those transitions) which is (are) going to be performed is the result of planning. The simplest "plan" is to choose randomly from the possible transitions. Reasoning engines may allow the self-model to be active in a single state only or in multiple states at the same time.

Transitions may trigger actions referring to the outside world (e.g., message sending) or to the specific part.

Besides "normal mode" there is an additional, distinguished operational mode of the reasoning engine, called "supervision mode". If the ACE is supervised (pervasive supervision), the reasoning engine switches to supervision mode enabling the supervisor to access the internal part.

This section discusses the possible, most important characteristic properties of the Reasoning Engine. Further research in this field is scheduled for next year.

7.2.1 Parallelism, synchronization, queuing

Reasoning engines may operate in single-threaded mode or in parallel mode. As for now, the single threaded model seems to be enough, but to assure the generality of the model, also the multithreaded model is discussed.

Single threaded execution

Single-threaded execution means that the receiving and the work-up of the message happens in the same thread, consequently, the reception of the next message is blocked as long as the processing of the current message is not finished. Single-threaded behaviour prevents hard-to-reproduce synchronization problems such as cross-effects of independent but overlapping processed messages. On the other hand, single-threaded systems are more sensitive to order-related faults and deadlocks. In distributed systems, the delivery delay of messages may vary, depending on the actual network conditions. The

[6] "effect" is meant on a low level, e.g. a transition becomes enabled

following figure shows a deadlock situation: because of unlucky delivery delays, both parties are waiting for the other one's reply while the processing of the incoming messages is blocked so the other party is not able to reply.

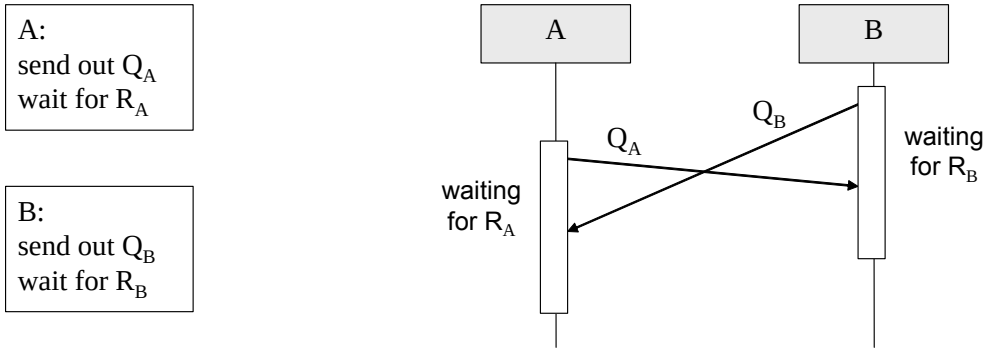


Figure 14 Deadlock situation

Timers and watchdogs are the simplest tools in deadlock detection. Deadlock prevention methods typically need prior knowledge about the other party and the message delivery times, which is not the case in a dynamically changing distributed system.

Multi-threaded execution

In case of multi-threaded execution, the reasoning engine can decide to process the incoming message on a new thread, which means that further messages can be processed at the same time. Obviously, a "smart" reasoning engine can detect and resolve certain deadlock situations (such as the one pointed out above)[7]. Even though this direction seems to be beneficial, the dummy solution is – to start a new thread for each incoming message – may result in cross-relationship faults. Independent messages that are processed overlappingly may put the self-model into corrupt/invalid states. A possible solution is to protect the synchronization-sensitive parts of the self-model with other tools, e.g., with monitor-based exclusive access.

Clearly, the multi-threaded execution model shall not be made "obligatory" for all ACEs, as it may be unnecessarily complex for some ACEs. On the other hand, in case of intrinsically complex ACEs (e.g., aggregated ACEs), this model may help in keeping the component model simple and lightweight.

7.2.2 Message sources, proactive manner, timing

Messages processed by the reasoning engine may originate from several sources. The most important sources are:

- External sources:
  - ACE (either independent or in composition with the receiver ACE)
  - Proprietary/ Legacy devices
- Internal sources:

[7] It's a possible research area how to make the reasoning engine smart enough, based on its Self- and Environment-Models.

- Specific part (return value)
- Internal timer

Received messages are inserted into the processing queue of the Reasoning Engine, where they wait to be processed.

External message sources

Inter-ACE communication is discussed in details in other sections of this document.

Messages originating from proprietary/legacy devices are translated into "standard" inter-ACE messages so technically, they do not differ from the real inter-ACE ones.

External messages are received via the Message Handler.

Internal message sources

The Message Handler receives internal messages directly from the originating source.

A message arriving from the Specific Part may be a response to former specific-part-call or may be produced freely, without former impulse. As the communication is strictly message-based and is supervised by the Reasoning Engine, we think that allowing the Specific Part to freely generate messages does not limit the autonomy or the Self-* properties of the ACE, and makes it theoretically easier to show proactive behaviour. As the Specific Part cannot directly communicate with the ACE environment, message sending to the external world is also realised via a special message to the Reasoning Engine.

Timers are used as special tools to trigger proactive behaviour. When a timer expires, it generates a message that can initiate processes in the Self-Model. The timer is the only internal element that can generate messages independently, without prior trigger. With the help of set-and-expire timers – which can be set to a certain amount of time and expires when it is over – timed and/or periodic behaviour can be realised. A special case of using a timer is the Null Timer when the timer is set to an infinitesimally small amount of time so that it expires immediately. Null Timers can be used to generate proactive behaviour.

7.2.3 Single-state vs Multi-state engines

From the Reasoning Engine point of view, it is an important question how many states of the Self-Model can be active at the same time.

The simplest case is the Single-State model when the Self-Model is in a definite, single state in each moment. Single-State Reasoning Engines are simple: they use the single-threaded model – multi-threading has no sense – with all positive and negative consequences (e.g., deadlock-sensitivity).

In a Multi-State model, several states of the Self-Model can be active at the same time, and incoming messages are cross-probed with each active state. The Self-Model must describe the rules of how to maintain consistency (e.g., hierarchical states, multi-level state-machines). In case of Multi-State Reasoning Engines, the multi-threaded message processing seems to be adequate, especially if the messages processed in parallel affect

independent state sets. The simplest Multi-State model is to execute all possible allowed[8] transitions, and not to care about how many states will become active.

Please note that in this issue similar considerations should be taken as in the case of multi-threaded model: it shall not be obligatory to support multi-state execution in order to keep simple ACEs as lightweight as possible.

7.2.4 Determinism, planning

Determinism becomes important if more than one transitions are allowed and the reasoning Engine has to choose which one(s) to perform. This choice can be understood as the "plan". From the Reasoning Engine point of view, the best plan is if no choice is needed (e.g., because the Facilitator disabled all possible but unnecessary transitions).

7.2.5 Supervised mode

For supervision purposes, an ACE can be put into a contractually agreed supervised mode, meaning that instances of certain message types specified in the supervision contract may be intercepted, changed or removed. The supervision authority needs access to the internal message triggering and relay mechanisms of the supervised ACEs and may also override the decision of the Reasoning Engine and Facilitator, requiring access to the supervision model (e.g., currently executing Self-Model) and internal state assessment information.

7.3 Self-model

The goal of the Self-Model is to describe possible behaviours of the ACE. First of all the Self-Model defines the internal states and the transition among them. Transitions may have input and output. Output may be the result of the assigned action, e.g., call to the specific part or message sending to another ACE.

Several descriptive formalisms can be used for the Self-Model. CASCADAS is focusing on three selected alternatives: extended finite state machine based description, Petri net based description, and SXL based description. The descriptive power of the model may vary depending on the formalism.

The last subsection deals with meta-properties of the Self-Model such as determinism and number of active states at the same time.

7.3.1 Extended finite state machine based model

Finite state machines (FSMs) are well-known and simple tools to describe state-based (context-sensitive) behaviour. The bottleneck of the model lays in its simplicity: the descriptive power of a "normal" state machine is often not enough for real-life problems. A wide range of extensions are known to increase the descriptive power. Basic FSMs have the same descriptive power as Regular Grammars (Chomsky 3 class).

[8] Note that the Facilitator can turn the transitions on/off. This means that the allowedness of the transition does not exclusively depend on the match with the incoming message.

Due to the age and popularity of the FSM model, slightly different things are meant by the same name. In order to prevent misunderstandings, we give formal descriptions first.

Syntax

The basic final state automaton is a 5-tuple.

(1) Finite State Automaton = (S, Σ, δ, q0, F)

- S: finite set of states
- Σ: alphabet
- δ: set of transitions (S * Σ → S)
- q0: initial state (q0 ∈ S)
- F: finite set of accept states (F ⊂ S)

In CASCADAS, we are considering the following extension directions: output, actions and guard conditions.

The FSM extended with output is a 6-tuple:

(2) FSM extension with O = (S, Σ, δ, q0, F, O)

- S: finite set of states
- Σ: input alphabet
- δ: set of transitions
- q0: initial state (q0 ∈ S)
- F: finite set of accept states (F ⊂ S)
- O: finite set of outputs

Compared to the basic model, the introduction of the output is a big difference. The automaton is able to affect its environment, so it can show behaviour not only internally, but also externally.

- δ = (S * Σ → S) / O, where / demarks output

The FSM extended with actions is a 6-tuple.

(3) FSM extension with A = (S, Σ, δ, q0, F, A)

- S: finite set of states
- Σ: input alphabet
- δ: set of transitions
- q0: initial state (q0 ∈ S)
- F: finite set of accept states (F ⊂ S)
- A: finite set of actions

The novelty of this extension – compared to the output model – is that actions are not static but dynamic elements. Actions can produce more complex and more flexible behaviour, too (e.g., the reaction depends on the actual input of the action).

Theoretically, four action types are possible, from which we are focusing on two: namely the Transition Action and Input Action. The Figure 15 shows the possible action types:

State Entry Action is executed when entering a state, State Exit Action when exiting it, Input Action depends on the actual state and input, and Transition Action is executed when performing a certain transition. Provisionally, input actions can be used by the Facilitator, and Transition Actions are the tools to refer to the specific interface, for example.

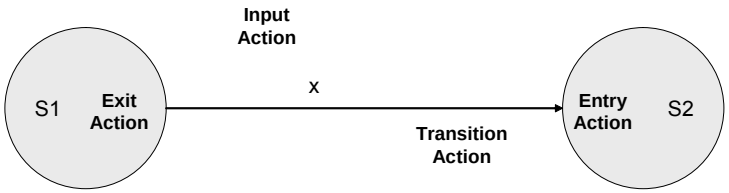


Figure 15 Action types

To be formally complete, here is the missing formal definition of the transition.

- $\delta = (S * \Sigma \rightarrow S)$ // A, where // demarks the side-effect

Actions are functions with range and domain which are not defined in details at this point.

Please note that this model is quite similar to the basic one (1) as it doesn't send response to the outside.

The FSM extended with actions and output capturing is a 7-tuple.

(4) FSM extension with I/A/O = $(S, \Sigma, \delta, q0, F, A, O)$

- S: finite set of states
- Σ: input alphabet
- δ: set of transitions
- q0: initial state ($q0 \in S$)
- F: finite set of accept states ($F \subset S$)
- A: finite set of actions
- O: finite set of outputs

In this extension, the output of the transition is generated by the action. Of course, static answers (no action just constant output) are also possible. Using this extension, the ACE is able to show dynamic and context-sensitive behaviour.

Action and output seem to be similar but are different. Output is a data (message, information), while action is a call to an executable code (method call, script).

Assuming that the input alphabet of the actions is I and the output alphabet is X:

- $A = (I \rightarrow X) / O$

We are not using the $A = (I \rightarrow O)$ notation because it is confusing. The action may result in changes in the internal state of the specific part (which is independent of the Self-Model), and as a side-effect, produces some output.

There is a special case when the output alphabet of the action is part of the input alphabet of the automaton ($O \subset \Sigma$). In this case, actions trigger further actions.

From security viewpoint, it is recommended that truly external output is produced at a few points of the automaton only. Other parts of the automaton should trigger these points to produce the desired external output message. So, normally, the output alphabet of "normal actions" should be a subset of Σ ($O \subset \Sigma$), while a limited number of selected actions may have different output ($O \cap \Sigma \neq \varnothing$).

The FSM extended with guard conditions is a 6-tuple.

(5) FSM extension with G = (S, Σ, δ, q0, F, G)

- S: finite set of states
- Σ: input alphabet
- δ: set of transitions
- q0: initial state ($q0 \in S$)
- F: finite set of accept states ($F \subset S$)
- G: a finite set of functions with Boolean return values

In the guard condition based model, transitions can be executed if the guard condition allows it.

- $\delta = (S * \Sigma \rightarrow S / A \mid g(S * \Sigma))$, where | demarks condition

Guard conditions raise interesting theoretical questions. If the guard condition is complex enough, it can substitute the whole "state" concept, so it may be possible to create an identical 1-state-and-complex-guard-condition automaton for each many-states-normal FSM. Of course if we restrict not only the range of the function but also the domain of it, then such non-orthogonal cases can be excluded[9].

Guard conditions can help in the state explosion problem. If the state consists of *n* state factors (f_i), and the actual value of f_x is only interesting in the aspect if it is above or below a border. In this case, it is a good idea to remove f_x from the state and check it using a guard condition.

Guard conditions can also help if the possible number of states is infinite.

Usability in CASCADAS

As ACEs must show some behaviour (through sending messages), the basic FSM model (1) is definitely not enough.

For very simple ACEs, the FSM with output (2) could be enough. The problem of this model is that it is inflexible, it is not much able to adapt to the situation, as outputs are statically predefined. Another problem is that type (2) FSMs must have finite state space, which means that e.g., no floating point number or non-ranged integer can be in its state descriptor.

For more complex ACEs, where more descriptive power is needed and there is the danger of the state explosion, too, the combination of type (4) and type (5) FSMs are encouraged. The guard condition helps in managing the originally infinite state space, and actions with output make it possible to produce dynamically adaptable responses.
A possible mapping is:

[9] It's a different question if we gain anything with the restriction or not.

- The Self-Model is a type (4,5) extended FSM, managed by the Reasoning Engine.
- In case of non-determinism, the Facilitator has to select the transition(s) that should be executed (by disabling the others).
- The Facilitator should be triggered by Input Actions.
- There should be two types of Transition Actions: Call to the specific interface and External message sending. Calls to the specific interface should be able to produce ($O \subset \Sigma$) outputs. External message sending should be able to produce outgoing messages and hand it to the Message Handler for delivery.
- Guard conditions should help in keeping the state space finite (technically, guard conditions "fake" those state factors that are excluded from the state space).

Other considerations

Meta-properties of the FSM may describe further attributes of the model:

- Behaviour in case of non-determinism, which is important if the Facilitator didn't make the model (or more precisely the model part) deterministic. A choice number can define how many transitions should be executed (1 or *) of the possible ones.
- Number of active states (1 or *). In a simple FSM, exactly one state is active at each time. In an extended model, any number of states can be active.
- Parallel planes of the FSM. It is possible that the self-model consists of several parallel FSMs (e.g., one of them can produce the outgoing messages), which are working independently of each other. Parallel planes may have different descriptors (meta-properties).

Other extension possibilities.

- Trust/probability/credibility values assigned to transitions. As a future connection point with the security domain Reputation/Trust system, values may be assigned to transitions describing the prospective consequences (for example, , for a cheating partner, the transition will sooner or later lead me into S1 with the probability of 0.8 and to S2 with the probability of 0.2; while for a reliable partner, S1/0.01and S2/0.99.) This can help in planning and optimization tasks. Values might be defined on the Self-Model itself, or on the abstraction of the Self-Model (where a series of transitions is represented by a single one).

7.3.2 Petri net based model

Syntax

The Petri nets (PN) are networks consisting of *places*, *transitions* and *directed arcs*. An arc always runs between one place and one transition, never between transitions nor between two places. The place from where the arc is pointed to the transition is the *input place* of the transition, the place where the arc points to from the transition is the *output place* of the transition. A transition can have any number of input and output places.

The places may contain any number of *tokens*. The distribution of the tokens over the places of the network is the *marking* of the network.

A transition is enabled and can *fire*, whenever every input places of it contain at least one token. When the transition fires, it consumes tokens from every input places, performs

some internal processing, and outputs tokens to each output place of it. The firing is a single non-preemptable event. Although multiple transitions can be enabled at the same time, only one of them can be fired. The enabled transitions are fired automatically and non-deterministically. No transition is required to fire – the enabled transition fires whenever it will, between *T=0...∞*.

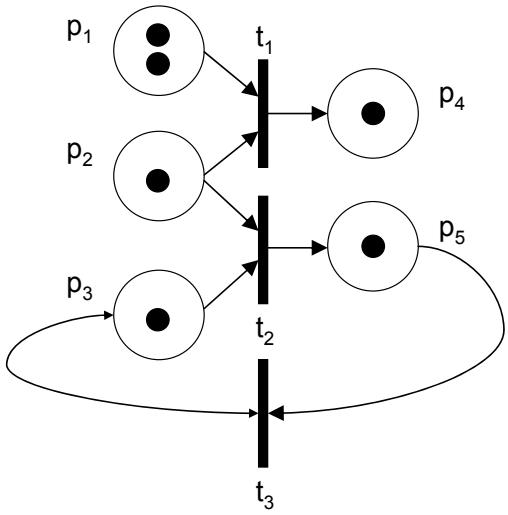


Figure 16 A basic Petri net

The formal description of the basic Petri net is a 4-tuple (P,T,F,M_0), where

- *P* is the set of places.
- *T* is the set of transitions.
- *F* is the set of directed arcs, also known as the flow relations. This set is subject to the restriction: $F \subseteq (P \times T) \cup (T \times P)$, namely that no arcs may connect two places or two transitions.
- $M_0 : P \to N$ is the initial marking, where each $p \in P$ places contain $n \in N$ tokens.

There are two widely used extensions to this basic model.

- $W : F \to N^+$ is the set of arc weights, which assigns to every $f \in F$ arcs an $n \in N^+$ weight meaning that when the transition is fired, it consumes weight number of tokens from the arbitrary input place, and puts weight number of tokens to the arbitrary output place.
- $K : S \to N^+$ is the set of place capacity restrictions, which assigns to each $s \in S$ places an $n \in N^+$ capacity restriction, the maximum number of tokens the place can hold. As a result, a PN can be called a *k-bounded* PN, when every places of it possibly can contain maximally *k* tokens.

Important properties

Petri nets have some important properties: reachability, liveliness and boundedness.

Reachability $R(N,M_0)$ means all the possible states (markings) that can be reached in network *N* from an initial M_0 marking. In modelling, reachability can be used to check whether the system gets ever into a desired or even an unwanted state.

Liveliness is defined by means of the execution of transitions. Five levels of liveliness are defined: a *t* transition is L_0 live (or dead) if there is no $\vec{\sigma} \in L(N,M_0)$ firing sequence (trajectory), where *t* can be executed. L_1 live for *t* means, that there is at least one trajectory, in which t can be fired. The PN is live (L_4 level), if in any reachable M markings ($\forall M \in R(N,M_0)$) any *t* transition is L_1 live.

As seen previously, *boundedness* can be explicitly defined for a Petri net if capacity restrictions are introduced. However, boundedness can also be a possible property of PNs without this explicit definition. A PN having no explicit capacity restrictions for its places is k-bounded, if the *k* is maximum number of tokens that any place can possibly contain in any reachable states. A PN is safe if it is *1-bounded*.

Extensions

There are several extensions to the basic model; we mention some of them possibly interesting in CASCADAS.

Guard conditions: the arcs can have a guard conditions defined; which means an additional precondition to be fulfilled before the transition can fire. If the guard condition evaluates false, the transition is disabled, even if it contains the required number of tokens on all input places of it.

Colouring: the basic PN utilises only one type of tokens that cannot be differentiated. The coloured Petri net introduces the token colouring: every token has an additional colour or value property. Transitions then can classify tokens based on their values. This extension makes possible e.g., flowing different packets through the network.

Prioritized Petri net: in such a PN the transitions have priorities. A transition can then be fired only if there are no competing transitions having a higher priority.

Usability in CASCADAS

The Petri net comes to scope of CASCADAS at the description of the self-model; the usability of this formalism is currently under discussion.

Compared to the FSM, PN basically makes it possible to have the same action-result oriented execution. Guard conditions can function the same way as in the FSM case. In fact, with the restriction that each transition of the PN has exactly one input and exactly one output place, we define an FSM.

However, utilizing the full spectrum of PN's descriptive power, we can have more sophisticated models in some ways. The multiple prerequisites of a transition (multiple input places) allow for a more detailed precondition description of the state-to-state transitions.

The non-deterministic execution of the firing of transitions makes the Petri net a good tool to describe execution of concurrent and competitive tasks. Desired properties of the modelled system can be checked by validating the properties of the network, e.g., the system can never get locked if the Petri net is live or the effect of a bounded system resource can be examined by defining a k-bounded place.

7.3.3 SXL based model

SXL [32] is an executable language for modelling simple behaviour. The basic idea is to give the description of the behaviour in a black box format, called pre-conditions and post-conditions. The model presented here is motivated by SXL but is not completely identical to it.

Syntax

The SXL based model is a 4-tuple:

SXL model = (S, I, O, T)

- S: set of states
- I: input alphabet of tasks
- O: output alphabet of tasks
- T: set of tasks

The goal of SXL is to describe the behaviour of tasks in sense of input-output, preconditions and side-effects.

The description of a task is a 4-tuple: t = (pre, i, o, post)

- pre $\subset$ S, pre-conditions of the task
- i: input of the task
- o: output of the task
- post $\subset$ S, post-conditions of the task (side-effects)

The task description is two-faced: it can describe simple (even state insensitive) black box behaviour through input-output relationships, while keeping the possibility to describe state-sensitive behaviour and side-effects as well.

Originally, SXL was to model finite-state problems. But with applying similar extensions as in the case of FSM (e.g., guard conditions), it is possible to extend it so that to describe infinite state problems, accordingly.

Usability in CASCADAS

For CASCADAS, SXL can be of relevance when describing the behaviour of the specific interface. As the specific interface acts as a quasi-black-box (only observable through its input-output), the SXL based task description seems to provide enough descriptive power for it. In case of non-state-preserving specific part functions, post-conditions are empty sets, and preconditions are guard conditions on the input parameters. For state-preserving specific part functions, preconditions may refer to more than the input parameters, and post-conditions are non-empty.

Using SXL for the specific interface makes it possible for the Facilitator to re-design the self-model (assign/un-assign actions to the transitions of the extended FSM, or rather re-design the given part of the FSM according to the effects of the transitions). As SXL is a formal description, classical deduction techniques and reasoning can be used. For example, if an incoming request asks for a value that can be calculated as f$\circ$g$\circ$h (f,g,h are

specific part calls), the Facilitator can deduce it using the pre-and post-conditions of f,g,h, and create a corresponding state and transition sequence in the FSM.

SXL is only formalism; so it is not meant to substitute GN-GA descriptors but to provide a descriptive syntax/semantics to them.

7.4 Facilitator

As being the core autonomic component of the ACE architecture, the Facilitator is in charge of ensuring the autonomic behaviour. To "ensure the autonomic behaviour" means that the Facilitator is able to review the Self-Model: add/remove transitions, add/remove states, and assign/modify transition actions.

The Facilitator is put in action when the Reasoning Engine receives a Message. Based on its knowledge, the Facilitator may initiate changes in the Self-Model (e.g., make it deterministic for the Message) or may not intervene. The input of the facilitator is at least: the Self-Model (and the actual state of it), the description of the available Specific Part functionalities, and the incoming message. The Facilitator shall be able to formally process ("understand"?) all these information.

The Facilitator may include complex inner procedures/tools like simulation, environment modelling (e.g., history database, probability assignment), self-modelling (e.g., re-working of the Self-Model after aggregation), simulation, planning or optimization.

Adding a new transition to the Self-Model is often mentioned as Join Point concept in this document (the new transition is called Join Point).

7.5 Specific part

The specific part of the ACE contains executable code, and is executed in a container, also known as *sandbox*. This means that the specific part has limited access to resources, and has no control about its own instantiation, or assignment to clients.

Specific part functions have abstract interface descriptions, where input/output relationships, preconditions and side-effects are given. The abstract description should be written in a language/formalism that is understandable for the Facilitator/Reasoning Engine/Self-Model.

7.5.1 Resource access

Specific interface semantically describes the specific part of an ACE and allows the internal part of the ACE to access the features implementing the specific part. The specific part can provide output for the incoming call, which (the output) may be a request to send a message to the external world, but the last decision still remains in the hands of the Reasoning Engine only, and they can be invoked through the Reasoning Engine only. Technically, there might be one exception: when the function is meant to wrap/abstract a simple but low-level resource access (e.g., a database call), it may open legacy communication channels to do so, but as it is outside of the ACE architecture, nothing can be guaranteed. In order to get guarantees, the low-level communication shall be performed using a Specific Message Handler which is able to translate ACE-understandable Messages to the resource-specific low-level messages and vice versa.

The specific part can be accessed through the Reasoning Engine, exclusively.

7.6 Interfaces

There are some interfaces identified so far which will be detailed and enriched during the future activities of the work package:

- Inter-ACE communication: Message Handler and GN-GA protocol
- ACE – Legacy protocols: Specific Interface
- Intra-ACE technical interfaces:
    - Message Handler – Reasoning Engine
    - Specific Part – Reasoning Engine
    - Facilitator – Reasoning Engine

8 Realising Autonomicity

This section aims to explain how the key autonomic features are supported by the architecture described in the previous sections.

8.1.1 Self-Similarity

Self-similarity is about aggregating components while the aggregate is *identical* to its parts. The purpose may be to increase scalability, ease configuration or re-use solutions on different levels of abstraction. A common example is a WWW server accessible via a domain name, but actually distributing the load to a number of different computers that serve the same content, have the same structure, address, etc. ACEs are not only self-similar because their common interfaces are not allowed to be modified and have to be implemented by every ACE, but also because of the way composition is handled, allowing for an elegant use of group communication. For example consider an aggregated ensemble of 10 ACEs whose functionality is accessible via 10 specific interfaces. Using a composed ACE, the same functionality could still be accessible via a single common specific interface using group communication primitives, like **`one-of`** or **`all`**. A composed ACE would transparently delegate the task to either a single ACE or the whole group. Please refer to [14] on how such primitives may be implemented.

A controversially discussed topic refers to the difference between the hosting environment of an ACE and the ACE itself. If one refers to a system exhibiting the self-similarity property, wouldn't that also mean that both, execution environments as well as ACEs would need to expose this property? A solution was found within the so-called "Service Execution Environment ACE's". SEE ACEs are elements that implement the container functionality used to host other ACEs. They create a homogeneous environment independent of the underlying platform by realising the *common interface* and the functionality described by the *specific interface* of a SEE ACE. Realising the hosting container as an ACE itself ensures the principle of self-similarity and enables the hosting code to benefit from all other functionality available via the common part of ACEs.

SEE functionality is specific to the underlying platform (e.g., certain communication primitives) and the code realising it is found in the specific part of an SEE ACE. Every ACE needs to expose a similar common part; the SEE ACE does this by exporting its specific functionality through the common interface, enabling other ACEs to bind to its specific functionality by using the common interface. This mechanism is depicted in Figure 17.

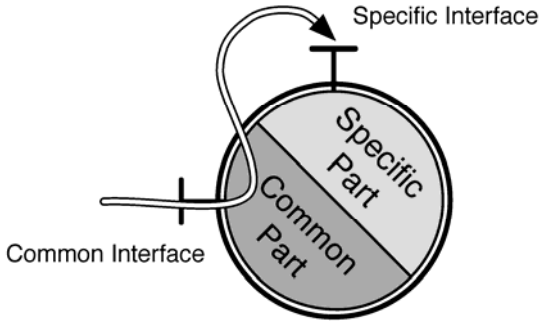


Figure 17 Concept of the Service Execution Environment ACE

As SEE ACEs are highly dependent on the underlying platform it may be assumed that they will not be able to migrate because of their dependence on immobile functionality. Regarding bootstrapping, we will not prescribe a standard way of bootstrapping SEE ACEs, but leave this question open to the decision of implementers. A certain implementation has to decide in a proprietary way on how to supply the initial binding information to the bootstrapped element.

8.1.2 Self-Healing by Using Dynamic Binding

Automated repair of functional dependencies among cooperating ACEs is the most prominent example for *self-healing* aspects. Figure 18 shows an example of binding dependencies between two ACEs: A *SEE ACE* and a hypothetical *"My" ACE* as a placeholder for any other ACE. The *Common Interface* is understood as the access point to the *Common Part*, exposing bindings to the functionality that is shared among both ACEs. The *Common Part* contains functionality that implements the *Common Interface*. In the case of a *SEE ACE* the *SEE Interface* is used to realise the *Common Interface* functionality.

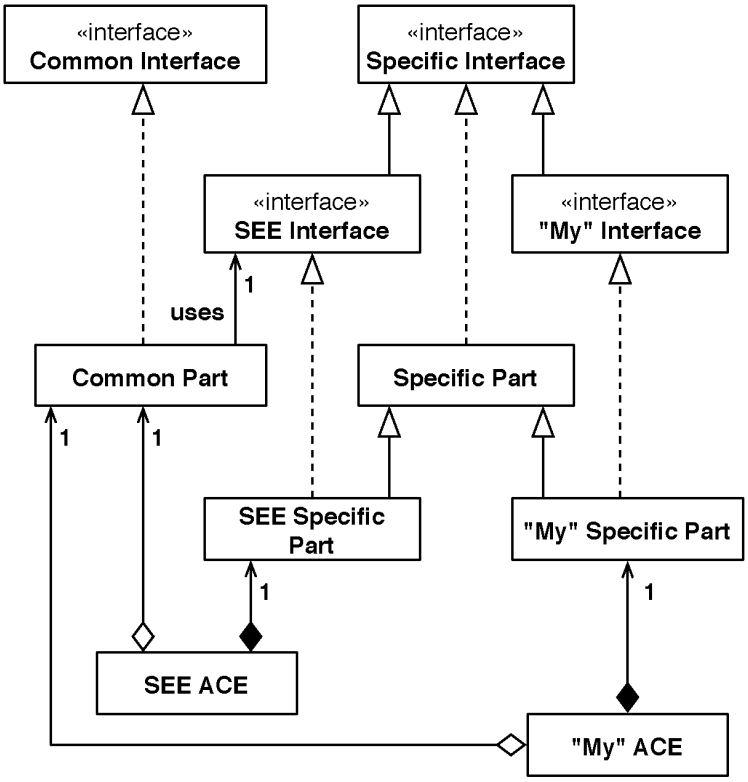


Figure 18 Dependencies of ACE bindings

The *SEE Interface* gives an access point to the platform dependent code of the *SEE ACE*, realised in the *SEE Specific Part*. The *SEE Interface* is a *Specific Interface*. *"My" Interface* is also a *Specific Interface* and realised in the *"My" Specific Part*, which is a *Specific Part*. The *"My" ACE* consists of a *Common Part* and the *"My" Specific Part*.

Implementing a new ACE, i.e., *"My" ACE*, requires a definition of the *"My" Interface* and its realisation in the *"My" Specific Part*. If *"My" ACE* would be realised as an internal composition of existing ACEs, then *"My" ACE* would inherit all bindings from its contained ACEs. It would then be possible for it to substitute existing bindings with appropriate ones from its own *Specific Part*.

As we expect an SEE ACE to be the first available element on a host, several ACEs would be created with functional bindings depending on this element (e.g., during a migration or copy of the SEE ACE). If we consider the case of termination of such an ACE, then a direct consequence would be the invalidation of functional bindings of any dependent ACEs which would lead to the termination of the cooperating ensemble of ACEs. Automatically repairing such functional bindings, e.g., by redistributing components as proposed in [13], re-assigning roles of the provisioned cooperation or by automatically searching and using ersatz bindings are understood to contribute to the self-repair capabilities of ACE based systems.

8.1.3 Self-Organisation

Self-organisation means that the structural and functional organisation of ACEs is done by the ACEs themselves. This has to include the discovery of other ACEs and provisioning of transport-level communication primitives and addressing; negotiation of rules, policies, roles, and constraints of the cooperation; instantiation and supervision of the cooperation. As the CASCADAS project has a complete work package (WP 3) devoted to self-organisation, the results of this work package will be used for selection of appropriate strategies and optimal parameterisation of the cooperative models.

Key features of ACEs to enable self-organisation capabilities are found in the composition mechanisms (both external and internal one), by exploiting context information accessible through the knowledge network, and in the potential ability to migrate between different locations.

8.1.4 Self-Awareness and Self-Description

Regarding to ACEs, the term "Self-Awareness" is used in reference to the concepts of introspection and reflection. "Self-Description" is a necessary prerequisite for achieving self-awareness, as it gives an ACE the ability to communicate models of behaviour among ACEs using common semantics.

Introspection refers to the capability of an ACE to gain information about its structural and functional constitution, e.g., the interfaces of services exposed by itself or contained ACEs or certain operational data. This information is to be accessed at runtime through analysis of the self-model that every ACE has access to.

Reflection aims at enabling an ACE to obtain information about how its behaviour is perceived by other ACEs, in other words to obtain an external view about itself. This is useful for example in scenarios where an ACE becomes a victim of a hostile program (e.g., a virus). In this case an ACE using introspection might conclude wrongly that it is working in a normal operation mode, whereas other ACEs might notice that it started to transmit a huge amount of malformed messages on the network. Using reflection would enable an ACE to ask its neighbours about their opinion of the situation, in the example leading the ACE to realise that it has been compromised. Reflection is to be implemented as part of the protocol that enables ACEs to exchange models and reason about the abilities.

8.2 Interaction models and communication primitives

ACEs need to interact with each others in order to establish collaborations, aggregation and to get knowledge about other ACEs features and the information. Moreover ACEs need to deal with different degree of dynamicity avoiding any centralized control.

8.2.1 The importance of the interaction model

In order to achieve the goals mentioned above we need an interaction model which allows ACEs to coordinate each others, access the right information from other ACEs and invoking the right ACEs without any centralized orchestration, planning or directory service.

8.2.2 The basic assumption

The interaction model outlined in this section is mainly based on the assumption that if two ACEs are close (using a proper distance heuristic) we can state that the two ACEs deal with correlated topics.

The assumption above should allow facing one of the key challenges in avoiding centralized control: limiting the overhead due to the interactions among ACEs needed to reach an agreement and to plan the right organisation.

The implementation of a "semantic time to live" (STT) is needed to avoid that each time an ACE advertise its goal this is forwarded to any other ACEs without any filter. This STT works on the following assumption: if an ACE receives a GA related to its GN, it would be proper to forward that information to neighbour ACEs.

The following example should briefly clarify what we would like to achieve. The picture below contains 4 ACEs, A1 and A2 provided by the service provider 1 and B1 and B2 provided by service provider 2. These are the GAs provided by the different ACEs:

- A1: it is able to run a query to get the Italian city;
- A2: query to get a list of restaurant given a city name;
- B1: it runs a simple web server;
- B2: it is able to produce HTML pages given information about cities;

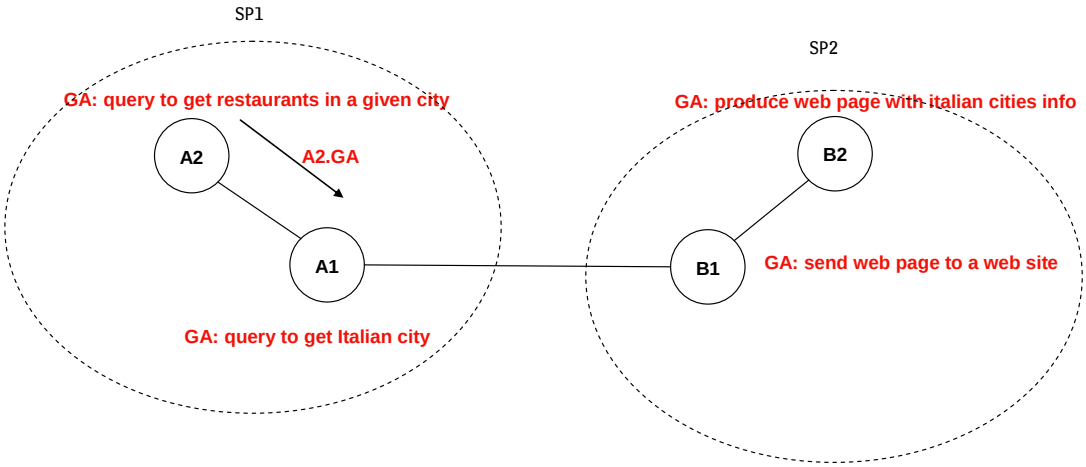


Figure 19 interaction example

What we need in the example above is that the semantic distance between A1 and B1 is such as A1 doesn't forward any information to B1 about A2's GAs. On the other hand if the example is that in the picture below, where the service provider 2 contains a node B3 which is able to produce HTML pages containing restaurant information, it is needed that B1 forward the A2's GA information to ACEs in service provider 2.

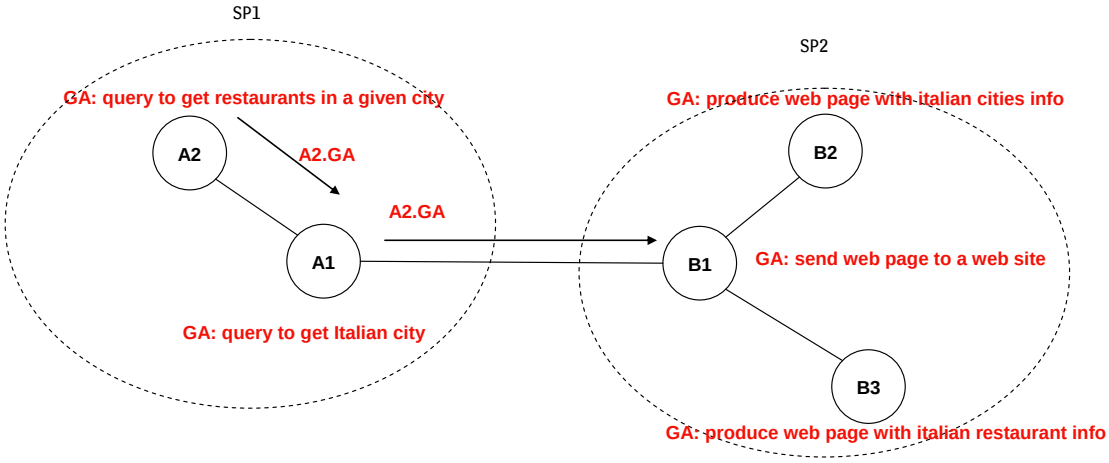


Figure 20 interaction example

8.2.3 A metaphor behind the interaction model

We can describe the key aspect of the interaction model using a simple metaphor: the ACE interaction model is fully based on a "pull" semantic which generates a sort of altruistic environment. Each ACE is altruistic: if it is able to do something and it is available to do it, the ACE will propose its help other ACEs as much as possible.

The key challenge in this case is to ensure that the information about ACEs availability and capability reaches the proper ACEs in a proper time. In order to achieve this we think it is needed to define a P2P protocol aimed to diffuse the information about ACEs capability and availability in a proper time and without unacceptable overhead.

The interaction model based on the above mentioned metaphor works in the following way:

- If an ACE is able to do something (expressed by means of GAs) and it is ready to do it, it advertises this information to all its neighbours (the information is flooded through the network).
- When an ACE needs something (expressed by means of GNs), it doesn't need to start any search. The ACE only need to check if someone has already advertised a feature (GA) which can address its need: this mechanism may be based on a blackboard metaphor.

8.2.4 Self-aggregation by means of P2P interactions

As stated in previous sections, the current vision is based on the idea that a certain service (to be created, executed and provided to a User) could be described in terms of a set of so-called goals.

As specified in the previous section, the key idea for enabling self-aggregation is the propagation of specific information (in terms of GAs) rather than "needs" (in terms of GNs). For achieving that, we introduce a so-called virtual board (distributed over the ACEs) where both goals (needed and achievable) could be semantically described and reported: Specifically the part of this DVB reporting the ACE GNs is exposed externally, and then accessible, via the so-called common interface, by other ACEs; the part of the ACE virtual

board reporting the GAs is kept internal the ACE and it is necessary to make the semantic matching GN – GA.

In summary, each ACE contributes to the overall virtual board (of the ACE population) reporting an internal portion of this distributed board subdivided into two parts:

- A public part, accessible via the common interface by other ACEs, listing the GN; given a semantic matching GN – GA, the GN will be taken by the proper ACE enabling the self-aggregation.
- An internal part listing the GA propagated; it should be noted that the key concept enabling the autonomic self-aggregation is the propagation of specific information (in terms of goal achievable) rather needs (in terms of GN).

As mention, the key concept is the peering propagation of a semantic representation of goal achievable.

The second main assumption is that each ACE receiving such information (semantic representation of goal achievable from another peer ACE) makes three actions: first it checks the semantic matching with its GN; second it elaborates the received information combining it with a semantic description of its GAs; third it properly propagates the resulted information (combination of the received GAs with its own GAs) to other peers.

The figure below shows the sort of GAs wave which is created among ACEs based on the defined mechanism.

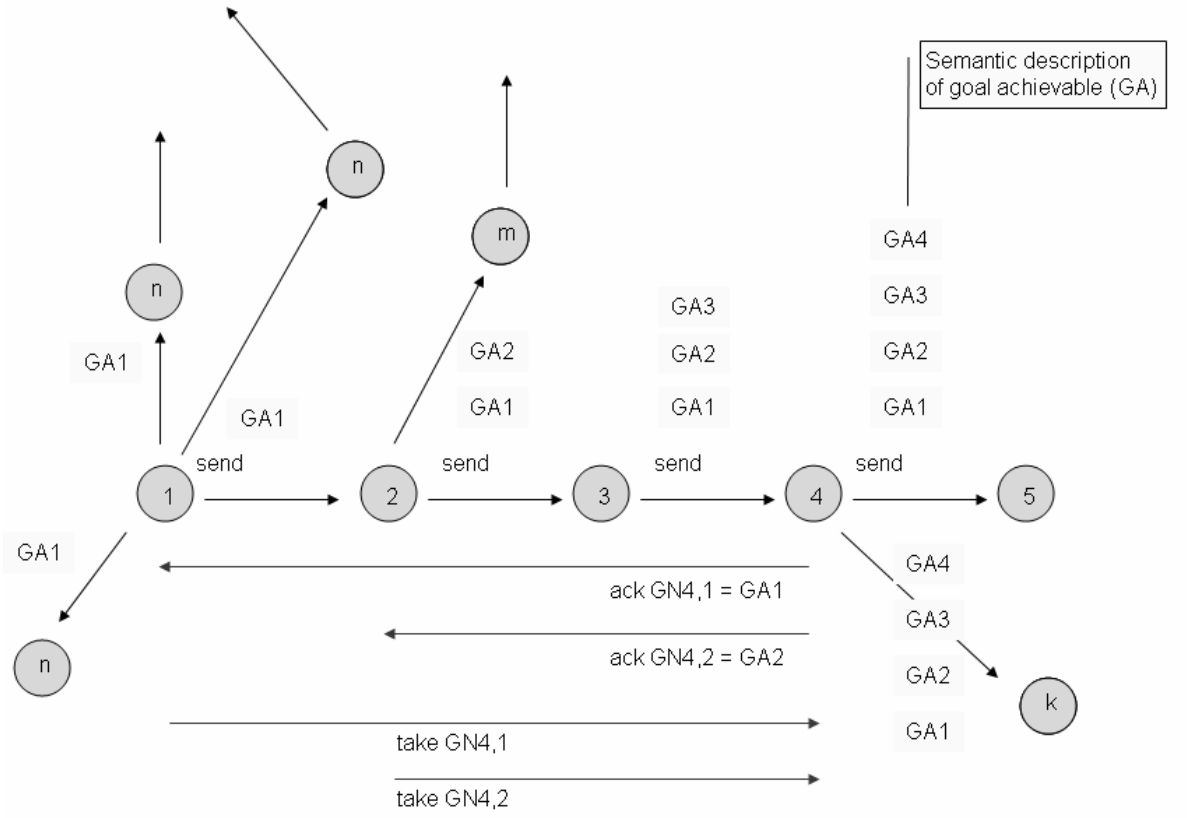


Figure 21 Self-aggregation of ACEs by distributing semantic information

Furthermore, it should be noted that:

In case an ACE discovers a semantic matching of a received GA with its GN, it sends back a kind of acknowledgment to the ACE in charge of the matching GA; the next step is a take of such GN by the tail-end ACE.

When an ACE elaborates the received information and it combines it with a semantic description of its GA, it modulates the "intensity" of the "wave" of information (to be propagated); for example it may reduce the level of details of the semantic description of the list of GA. This should allow scaling in the distributed information and a natural smooth decade of the propagation.

An ACE expected to propagate the elaborated information (combination of the received goal-achievable with its goal achievable) to other ACEs belonging to the same semantic domain. Anyway from time to time, randomly, an ACE has to propagate the information to other domain in order to allow the potential cross-correlation of different domains. This cross-connection of domain may enable satisfying new unpredicted emerging needs (creation of new services). As a matter of fact, randomness and fluctuations (or noise) play an important part in allowing the system to find optimal solutions and/or lead to the emergence of the right type of collective pattern. In some cases, it is even possible to identify an optimal level of noise that is most likely to result in the discovery of optimal solutions. Optimality is largely achieved through a balance between fluctuations leading to innovation and accuracy of communications or behaviour. Emergent collective behaviour can be robust with respect to other sources of noise like, for example, fluctuations caused by small populations of atypical individuals. Noise can be present either at the level of the individuals themselves or in the interactions among them. Since it appears to play a key role in natural self-organizing phenomena, incorporating a controlled noise level into the design of artificial systems and determining its optimal intensity should be given a high priority if such systems are expected to exhibit similar emergent properties.

Another aspect of this approach, is allowing a best-fitting competition among ACEs offering the similar GA to match a certain GN. Even if there might be more than one ACE making a take of a GN, there will be a natural selection of the "best" ACE matching the GN (also in terms of performance or other criteria). Another aspect, strictly related to this, is ACE replication (cloning): when an ACE self-detects some internal degradation may self-replicate itself and self-exclude itself from participating (with expected performances) to a certain aggregation.

8.2.5 A use case: behavioural pervasive advertisement

This section describes the application of the self-aggregation solution proposal for BPA scenario introduced in chapter 3.

The description doesn't contain any implementation details and for sake's clarity every semantic description of the ACE goals needed and achievable reported in DVB is given in natural language using XML tags to make a distinction between the types of goal.

The picture below is the representation scenario in which ACEs act and collaborate to realize the application: it will be the result of the ACEs aggregation mechanism based on the semantic matching GA-GN among ACEs. Four different actors take part to the scenario: 1 telecommunication operator and 3 service providers. The service provider 3 is needed to start the scenario, sending a SMS to the user's device: this match with the GN of the ACE aboard the user handset. The same mechanism is valid toward other ACEs .The

arrows in the following pictures are useful to understand the wave of GAs among ACEs to achieve the main goal of the scenario i.e., to personalize the Screen Window with the Ads images based on the population preferences deduced from the information gathered from user's handsets.

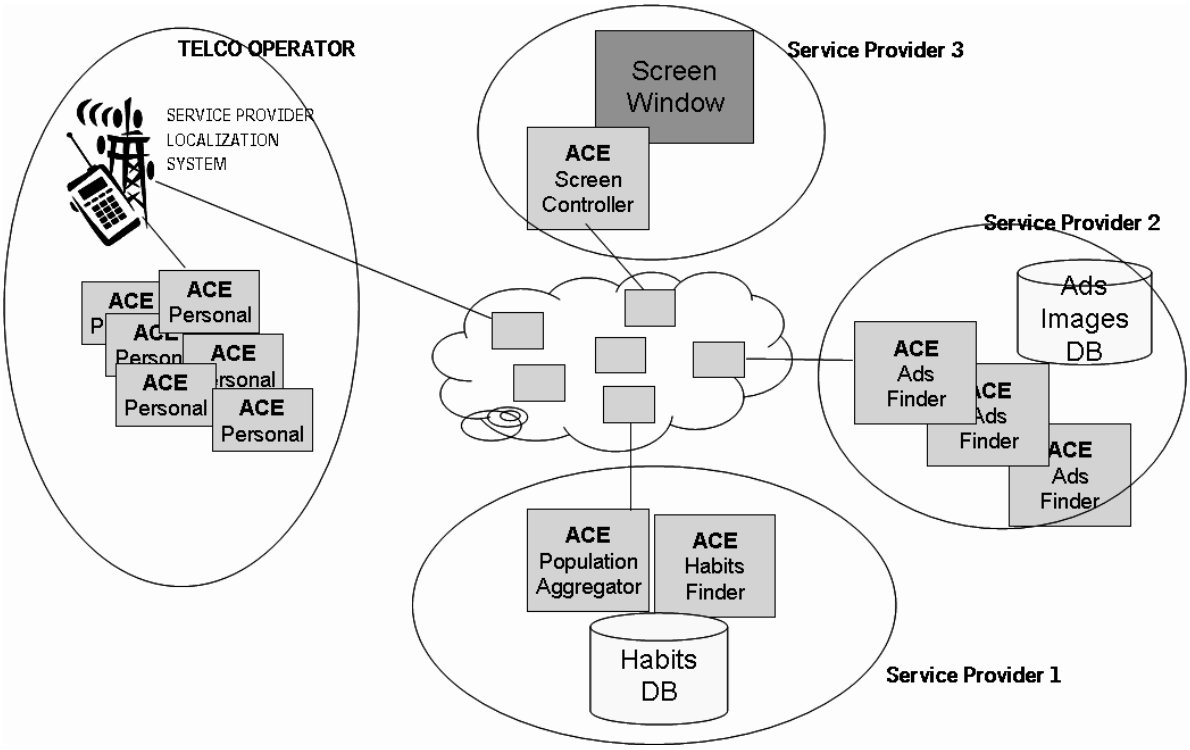


Figure 22 Scenario representation

The picture below shows the blackboard (DVB) made available by each ACE participating to the application. Each blackboard contains the GN needed by the ACE to reach its goal. The publication of the blackboard allows each ACE to take part to the application as soon as an ACE, able to meet the need exposed, sends that information by the GA wave and this latter is able to reach the blackboard through the inter-domains links.

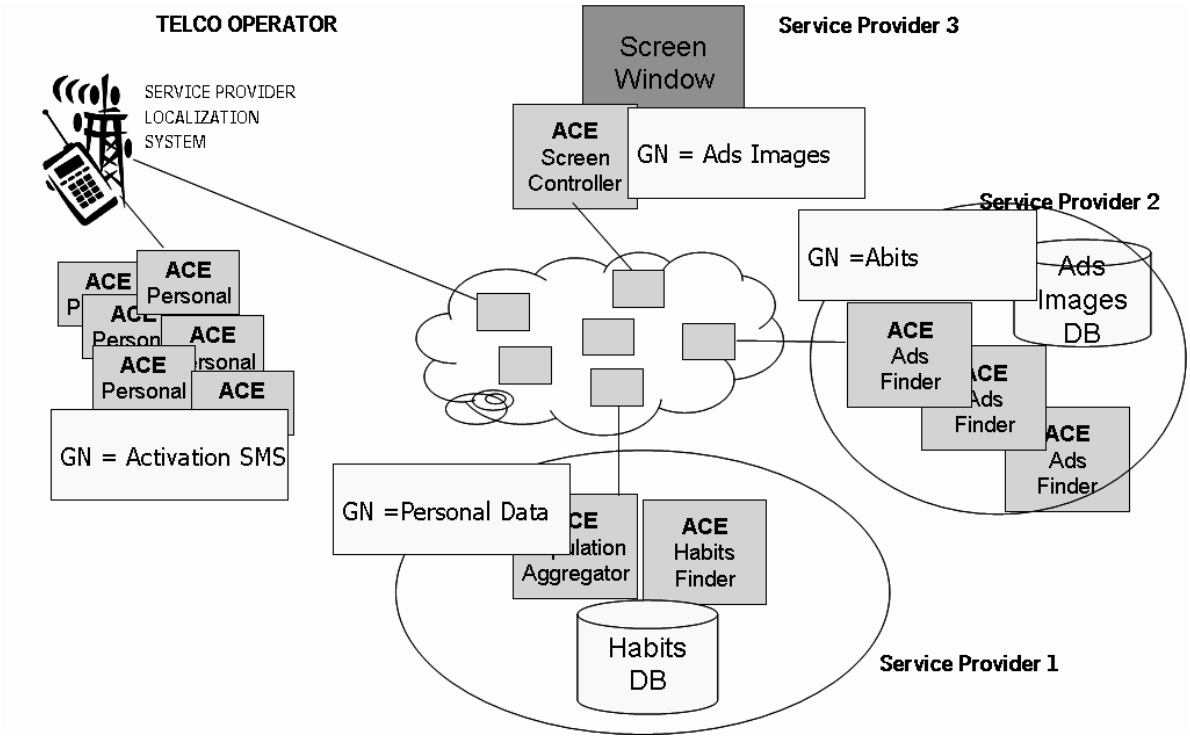


Figure 23 ACE Distributed Virtual Board

As soon as the ACEs Personal receive the activation SMS, they start to propagate the wave with their goal achievable. As the ACEs are properly connected to an ACEs network and an inter-domain links exist, the wave should be able to propagate until the right ACEs are reached. The figure below shows the waves propagation in the scenario.

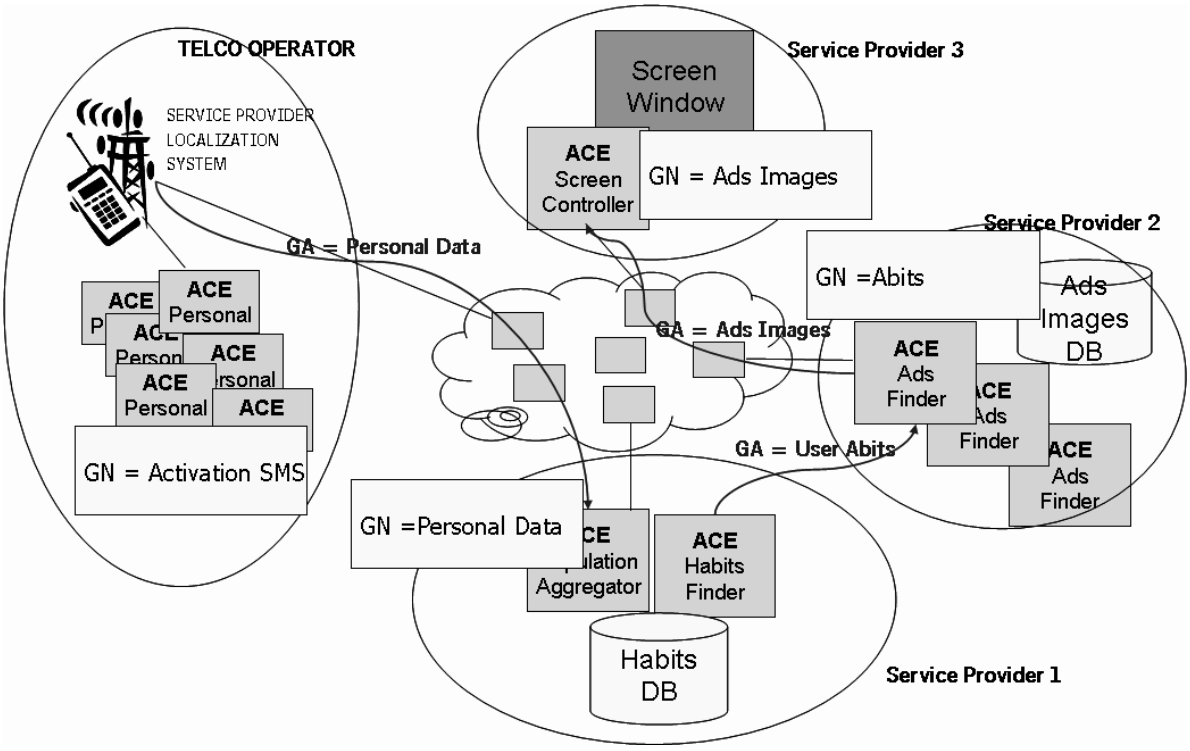


Figure 24 Wave propagation

The two following pictures show how the random mechanism introduced in the model enables an autonomous behaviour. The unpredictable event is that a new handset with new capabilities is introduced in the operator network. Such handset is able to show images so in its blackboard the GN = "Images needed" appears. As soon as this GN appears and the wave containing the GA = Ads Images is randomly propagated a new inter-domain link is created: images and operator handset start to deal each other so the ACEs of the service provider 2 start to sent images to the operator handset too.

The following picture shows the new propagation of the GA = Ads Images wave due to the introduction of a new handset with video capability in telecommunication operator network.

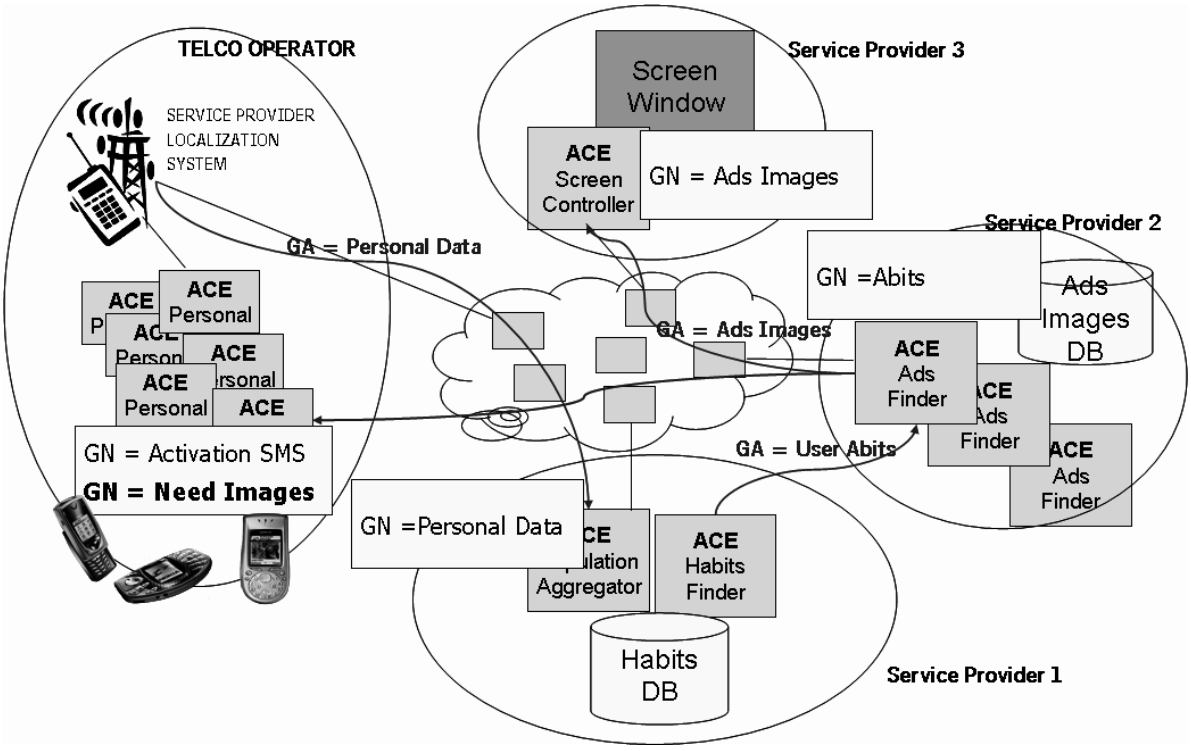


Figure 25 The new wave propagation

The key idea for enabling component self-aggregation by the propagation some specific information (in terms of goal achievable) rather needs (in terms of goal needed). For doing that a so-called virtual board (distributed over the components) is introduced. Both goals (needed and achievable) could be semantically described and reported in such distributed virtual board (DVB): specifically the part of this virtual board reporting the component goal needed is exposed externally, and as such it is accessible, via the so-called common interface, by other components; the part of the virtual board reporting the goal achievable is kept internal the components and it is necessary to make the semantic matching goal needed – goal achievable.

8. Conclusion

The overall objective of CASCADAS is to develop and to validate an autonomic framework for the creating, executing and provisioning situation-aware and dynamically adaptable communication services. Particularly the project development activities aims at prototyping a toolkit based on distributed self-similar components (Autonomic Communication Elements) characterised by autonomic features (self-configuration, self-optimization, self-healing, self-protection, etc.) The Autonomic Communication Element (ACE) is the basic component abstraction over which the CASCADAS vision is built. Services are being created and executed (in a distributed way) by the self-aggregation of ACEs

This document, constituting the Deliverable 1.1 "Report on state-of-art, requirements and ACE model", reports the main results (achieved during the first year of the project) about the definition of the ACE model and its interactions mechanisms.

CASCADAS adopted an application-oriented approach: starting from scenarios and related use-cases, high level requirements have been defined and are being used by WPs activities.

The project vision aims at validating a so-called Open Autonomic Service Environment defined as a highly distributed platform for composing, executing and providing situation-aware and dynamically adaptable communication and content services.

The essence of the innovation stands in exploiting highly distributed resources (even commodity servers of low-cost) running autonomic S/W solutions based on distributed self-aggregating, self-organising components (ACEs). The overall self-similar architecture (both pizza-box servers and clusters of servers have the same functional architecture) supporting a distributed replication of data. This will allow high levels of availability also starting from low-cost commodity H/W.

The tool-kit developed in the project will be used to demonstrate such vision specifically referring to some use-cases of particular interest (such as pervasive communications, etc).

References

[1] Annex 1 – CASCADAS Description of Work

[2] Autonomic Communication Forum (ACF), available online: www.autonomic-communication-forum.org

[3] Parashar, M., Liu, H., Li, Z., Matossian, V., Schmidt, C., Zhang, G., Hariri, S.: "AutoMate: Enabling Autonomic Grid Applications", Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing, Kluwer Academic Publishers, Vol. 9, No. 1, pp. 161-174, 2006

[4] Horn, P.: "Autonomic Computing: IBM's Perspective on the State of Information Technology", IBM T.J. Watson Labs, New York 2001, available online: www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

[5] Carreras, I., Chlamtac, I., De Pellegrini, F., Mionardi, D.: "BIONETS: Bio-Inspired Networking for Pervasive Communication Environments", 2005, available online: www.bionets.org.

[6] Chlamtac, I., Conti, M., Liu, J.: "Mobile ad hoc networking: imperatives and challenges", Ad-Hoc Networks Journal, Vol. 1, pp. 13-64, 2003.

[7] Chaintreau, A., Hui, P., Crowcroft, J., Diot, C., Gass, R., Scott, J.: "Pocket switched networks: Real-world mobility and its consequences for opportunistic forwarding", University of Cambridge, Technical Report Number 617, 2005

[8] F. Sestini, "Situated and Autonomic Communication an EC FET European initiative.", *ACM SIGCOMM Computer Communication Review*, Vol. 36, 2006, pp. 17–20

[9] F. Saffre, H. Blok, "SelfService: a theoretical protocol for autonomic distribution of services in P2P communities.", *Proc. IEEE Workshop on Engineering of Autonomic Systems*, Washington DC, USA, 2005, pp. 528–534

[10] H. Liu, M. Parashar, S. Hariri, "A Component Based Programming Framework for Autonomic Applications.", *Proc. IEEE International Conference on Autonomic Computing*, New York NY, USA, 2004, pp. 10–17

[11] P.T. Eugster, P.A. Felber, R. Guerraoui, A. Kermarrec, "The Many Faces of Publish / Subscribe.", *ACM Computer Surveys*, Vol. 35, 2003, pp. 114–131

[12] O. Droegehorn, F. Carrez, K. David, H. Helin, S. Arbanowski, et al., "Generic Service Elements and Enabling Middleware Technologies.", *Wireless World Research Forum (WWRF)*, Working Group 2, Whitepaper

[13] M. Mikic-Rakic, N. Medvidovic, "Support for Disconnected Operation via Architectural Self-Reconfiguration.", *Proc. International Conference on Autonomic Computing*, New York NY, USA, 2004, pp. 114–121

[14] C. Reichert, D. Witaszek, "An Implementation of the Group Event Notification Protocol.", *Fraunhofer FOKUS Technical Report TR-2002-0301*, Berlin, Germany, 2002

[15] Chaouchi, H., Smirnov, M., "Autonomic Communication: Business-Driven Revolution", IEEE Intelligent Systems, Vol. 21, Issue 2, pp. 57-58, 2006.

[16] Clips - Online resource: www.ghg.net/clips/CLIPS.html.

[17] Costa, P., Coulson, G., Mascolo, C., Picco, G. P., Zachariadis, S.: "The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems", 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05), 2005.

[18] Smirnov, M.: "Autonomic Communication: Research Agenda for a New Communication Paradigm", Fraunhofer FOKUS White Paper, 2004, available online: www.autonomic-communication.org/publications/doc/WP_v02.pdf.

[19] Ganek, A. G. and Corbi, T. A.: "The dawning of the autonomic computing era." IBM Systems Journal, Vol. 42, No. 1, 2003.

[20] IBM: "An architectural blueprint for autonomic computing", IBM White Paper, 2003

[21] Internet World Stats, available online: www.internetworldstats.com/stats.htm

[22] Kariv, O., Hakimi, S. L.: "An algorithmic approach to network location problems, II: The p-medians", SIAM Journal on Applied Mathematics, Vol. 37, No. 3, pp. 539-560, 1979

[23] Meier, R., Cahill, V.: "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks", 22nd International Conference on Distributed Computing Systems, Workshops (ICDCSW '02), pp. 639-644, 2002

[24] Mirchandani, P. B., Francis, R. L.: "Discrete Location Theory", Wiley, 1990

[25] Nolle, T.: "A New Business Layer for IP networks", Business Communications Review Magazine, pp. 24-29, 2005

[26] Oikonomou, K., Stavrakakis, I.: "Scalable Service Migration: The Tree Topology Case", IFIP Fifth Annual Mediterranean Ad Hoc Networking Workshop, 2006

[27] Kaufman, J., Lehman, T., Deen, G., Thomas, J.: "OptimalGrid – Autonomic Computing on the Grid", IBM article, 2003, available online: www-128.ibm.com/developerworks/grid/library/gr-opgrid/

[28] Scott, J., Hui, P., Crowcroft, J., C. Diot, "Haggle: a Networking Architecture Designed Around Mobile Users" Third Annual Conference on Wireless On-demand Network Systems and Services (WONS 2006), Les Menuires, France, 2006

[29] Strassner, J. C., Agoulmine, N., Lehtihet, E.: "FOCALE – A Novel Autonomic Networking Architecture", Latin American Autonomic Computing Symposium (LAACS), 2006

[30] Sterritta, R., Parasharb, M., Tianfieldc, H., Unland, R.: "A concise introduction to autonomic computing", Elsevier, Advanced Engineering Informatics 19, pp. 181–187, 2005

[31] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C.: "Grid service specification", 2002

[32] S. Lee, S. Sluizer, "An Executable Language for Modeling Simple Behavior," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 527-543, Jun., 1991.

[33] Fitzpatrick, A., Biegel, G., Clarke, S., and Cahill, V.: "Towards a Sentient Object Model", Workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE), 2002.

[34] Sun Microsystems: "JavaBeans specification", available online: java.sun.com/beans.

[35] Englander, R.: "Developing Java Beans", O'Reilly, 1997.

[36] Armstrong, E., Ball, J., Bodoff, S., Bode Carson, D., Evans, I., Green, D., Haase, K., and Jendrock, E.: "The J2EE™ 1.4 Tutorial", Addison Wesley, 2005, available online: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

[37] Lewandowski, S. M.: "Frameworks for Component-Based Client/Server Computing", ACM Computing Surveys, Vol. 30, No. 1, pp. 3-27, ACM Press, 1998.

[38] The Object Management Group (OMG): "Common Object Request Broker Architecture: Core Specification", Version 3.0.3, 2004, available online: http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf.

[39] Stephen E. Arnold "The Google Legacy" (Infonortics, Tetbury, England; September 2005) Chapter3 - http://www.infonortics.com/publications/google/technology.pdf

Acronyms

ACE	Autonomic Communication Element
ACF	Autonomic Communication Forum
AutoComm	Autonomic Communication
HSDPA	High-Speed Downlink Packet Access
GPS	Global Positioning System
GSM	Global System for Mobile Communication
HSDPA	High-Speed Downlink Packet Access
HSUPA	High-Speed Uplink Packet Access
PDA	Personal Digital Assistant
P2P	Peer to Peer
QoS	Quality of Service
RFID	Radio Frequency Identification
TMF	Telemanagement Forum
UMTS	Universal Mobile Telecommunications Services
UWB	Ultra-Wideband
Wi-Max	Worldwide Interoperability for Microwave Access