Deliverable D1.4

Integrated Prototype (release 2)

Status and Version:	Version 1.0 final	
Date of issue:	30.06.2008	
Distribution:	Project Internal	
Author(s):	Name	Partner
	Nermin Brgulja	UNIK
	Rico Kusber	UNIK
	Edzard Höfig	Fokus
	Peter Deussen	Fokus
	Marco Mamei	Unimore
	Borbála Katalin Benkő	BUTE
	Antonio Manzalini	TI
	Mario Giacometto	TI
Checked by:		
	Ricardo Lent	ICL
	Richard Tateson	BT

Abstract

Since the last deliverable [D1.3] several modifications and improvements have been applied to the ACE Toolkit in order to improve its stability, reliability and to provide new ACE features required by other work packages for integration. The overall ACE structure as defined in [D1.3] chapter 3 remained unchanged and is therefore not covered here. This document focuses on ACE Toolkit improvements, new features and overall CASCADAS project integration activities. It describes our efforts and the achievements regarding ACE Toolkit mobile device portability. A section on ACE Toolkit practical guidelines covers some most common questions that might arise when programming with ACEs. Furthermore, this deliverable summarises the ACE Toolkit experiments and evaluation results which have been carried out by WP1.

Table of Contents

1 Introduction

1.1 Purpose and Scope

This document constitutes deliverable D1.4 "Integrated Prototype (release 2)". It describes the new features and ACE Toolkit improvements that have been researched, developed and implemented since the last deliverable [D1.3]. Our efforts and achievements regarding porting of the ACE Toolkit to mobile devices are described here. The D1.4 deliverable also contains examples explaining how to perform common tasks that frequently occur when programming with the ACE Toolkit. Furthermore, the overall CASCADAS project integration work is summarised in this document as well.

Beside the technical achievements and descriptions, this deliverable summarises the ACE Toolkit experiments and evaluation results which have been carried out by the WP1. These results will be the basis for further ACE Toolkit developments and enhancements.

Since the ACE component model and its evolution have been described in earlier deliverables [D1.2] and [D1.3], we reference previous work wherever possible. Nevertheless, the reader might find some parts in the document which are very similar to the deliverables [D1.2] and [D1.3] but which needed to be included because of their relevance to the document.

1.2 Reference Material

1.2.1 Reference Documents

[ACEL] ACELandic - A Scripting Language for Autonomic Communication Elements, available at: https://www2.mik.bme.hu/repositories/cascadas/trunk/wp1/dev/doc/ACELandic.doc.

[ANDROID] Google Android platform for mobile devices, available online: http://code.google.com/android/.

[Benk08] Benkő, B. K., Brgulja, N., Höfig, E., and Kusber, R.: "Adaptive Services in a Distributed Environment", in Proceedings of the Eighth International Workshop on Applications and Services in Wireless Networks – ASWN 2008, Kassel, Germany, 2008 (to appear).

[Betal08] Benkő, B. K. et al: "Autonomic Communication Elements: Design Principles, Architecture and Implementation", IEEE Transactions on Computers, July Issue 2009, (submitted).

[CACAOJVM] CACAO Java Virtual Machine (JVM), available online: www.cacaojvm.org.

[Chsw] Chainsaw log viewer, available online: http://logging.apache.org/chainsaw/.

[CLASSPATH] Gnu Classpath open source implementation of the standard class library for the Java programming language, available online: www.gnu.org/software/classpath/.

[D1.2] Deliverable D1.2: Prototype implementation (release 1), July 2007.

[D1.3] Deliverable D1.3: First Prototype Integration, January 2008.

[D3.1] Deliverable D3.1: Aggregation Algorithms, Overlay Dynamics and Implications for Self-Organised Distributed Systems, December 2006.

[D2.4] Deliverable D2.4: Long-term supervision and integrated supervision pervasion (3rd evaluation prototype), June 2008 (to appear).

[D3.3] Deliverable D3.3: Software Implementation of Modules for Adaptive Aggregation, June 2007.

[D3.5] Deliverable D3.5: Report on rule-based modules for unit decision-making using autonomous unit rules and inter-unit communication, June 2008 (to appear).

[D4.1] Deliverable D4.1: Preliminary version of security architecture and mock-up of the interaction, December 2006.

[D5.3] Deliverable D5.3: The Open Toolkit for Knowledge Networks, January 2008.

[D6.8] Deliverable D6.8, Assessment Studies on the communication paradigms developed within CASCADAS, June 2008 (to appear).

[D8.5] Open-source CASCADAS integrated toolkit (release 2), June 2008 (to appear).

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

[Log4J] Log4J, available online: http://logging.apache.org/log4j/.

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

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

[RuleML] RuleML - Rule Markup Language. RuleML 0.88 stripped syntax is used, available online: www.ruleml.org/0.88/.

[SunGC1] JVM Garbage Collectors – Ergonomics, available online: java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html.

[SunGC2] JVM Garbage Collectors – Tuning, available online: http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html.

1.2.2 Acronyms

ACE	Autonomic communication element
AWT	Abstract Window Toolkit
BCO	Bus Checker Object
DIET	DIET Agent Platform (see [DIET])
GA	Goal achievable
GN	Goal needed
GUI	Graphical user interface
ID	Identification

JIT	Just in Time
JVM	Java Virtual Machine
Log4J	Logging framework (see [Log4J])
PEX	PlanExecutor
REDS	Reconfigurable Dispatching System (see [REDS])
WP	Work package
XML	Extensible Markup Language

1.3 Document History

Version	Date	Authors	Comment
0.1	27/05/2008	Nermin Brgulja, Rico Kusber	Proposal of initial document structure and responsibilities
0.2	26/06/2008	Ricardo Lent	Revision (R. Lent)
0.3	29/06/2008	Richard Tateson	Revision (R. Tateson)
1.0	30/06/2008	Nermin Brgulja, Rico Kusber	Final editing after internal review

1.4 Document Overview

In the first part of the document, acronyms, general definitions, and used references are clarified. Chapter 2 gives an overview of new ACE features which have been integrated during the latest research period. ACE parameterisation which allows developers to specify ACE initialisation parameters, self model syntax enhancements, ACE cloning features, etc. are described here. Chapter 3 describes our efforts, results and future plans regarding the ACE Toolkit mobile device support.

Chapter 4 contains practical ACE programming guidelines, which should help developers to easier understand how to program with ACEs. Selected ACE programming topics such as ACE contracting and context utilisation are described here. In chapter 5, ACE Toolkit experiments and results are presented and summarised.

Chapter 6 describes the current status of the CASCADAS integration work and future integration plans. It contains an overview of project wide CASCADAS integration activities as well as the detailed integration activities and plans per WP. The last section of this document, chapter 7 summarises deliverable D1.4 and concludes with an outlook on how the research and development of the CASCADAS ACE Toolkit will continue.

2 New ACE Features at a Glance

During the latest research period, the main ACE architecture (number of ACE organs and their purpose) remained unchanged. Beside mobile device support and project-wide integration activities, our work focused on enhancing existing ACE capabilities and implementing new ones. This section describes new ACE capabilities and features which have been developed in this research period.

2.1 ACE Parameterisation

The motivation to introduce ACE parameterisation came from several WPs which wanted to start the same ACE logic with different parameters (e.g. goal names). Before this extension, the parameters had to be specified inside the plan, for example in the first transitions; which made it inconvenient to re-use the code. ACE parameterisation helps in separating the operational logic from the actual input values.

2.1.1 General Details

ACE parameterisation allows the user to pass start-up parameters to an ACE instance.

- Parameters are defined in the aces.xml file.
- Parameters are available in the global session of the ACE instance.

The following example passes three parameters to the ACE instance "SENDER". After start-up, plans and functionalities can access the parameters from the global session as `?globalSession://s`, `?globalSession://i`, and `?globalSession://b`.

```
<?xml version="1.0" encoding="UTF-8"?>
<aces>
<ace name="SENDER" type="conf/aces/reftest-sender/ace-type.xml">
     <param id="s" type="java.lang.String">apple</param>
     <param id="i" type="int">15</param>
     <param id="b" type="boolean">true</param>
 </ace>
</aces>
```

The usage of ACE start-up parameters helps in minimising the manual work because it is no longer necessary to include the parameter initialisation transitions in the plan. It is also beneficial in terms of code-reusability.

Changes are backward compatible; all old applications continue to work without modification.

2.1.2 Syntax

Parameters can be listed in the aces.xml file. In order to add a parameter, insert a param tag inside the ace tag. Params must have two attributes: id (unique per ACE) and type. The value of the parameter is specified as the contents of the param tag.

In the following sample code we declare a parameter: the parameter name is "i", its type is "int", and the value is 15.

```
<param id="i" type="int">15</param>
```

Parameters can be of arbitrary type, and they are automatically resolved (from String) to the given class/primitive type. The following primitive types are supported:

- `boolean`
- `char`
- `double`
- `float`
- `int`
- `short`

Any complex type is supported, for example "`java.lang.String`" for Strings, or a user-defined complex type. The only requirement is that the type class must have a constructor that gets a String as a parameter (the String is the content of the tag).

2.1.3 XML as Parameter Content

It is also possible to have XML as parameter content, but the user should take care of the resolution. (The system forwards the XML segment as a String, without resolution.)

The following example shows how to use a custom type resolver (`FruitBasket`) to work up the XML parameter contents. The highlighted document segment will be passed to the constructor of the resolver.

```
<param id="X" type="package1.FruitBasket">
  <apple>red</apple>
  <pear taste=”fine”>yellow</pear>
</param>
```

The "`FruitBasket`" class must have a constructor that gets one String as a parameter, and it should take care of resolving the received XML segment to its internal fields. (Please note that the String passed to the constructor may not be a valid XML on its own because there might be no single root tag.)

```
package 1;
public class FruitBasket {
     public FruitBasket(String xmlSegment) {
          // resolves the XML segment to own fields
       }
}
```

2.2 Parallel Plan Relationships

When a new plan is introduced, we differentiate between the following relationship models.

- Independent new plan. The new plan gets a new execution environment: a newly instantiated execution session and a new, empty input queue. The new plan starts from its initial state.
- Child plan. The new plan inherits the execution environment of its parent plan: they share the execution session and the content of the parent's input queue is copied to the child's input queue. The child is started from its initial state. The execution of the parent plan is not influenced by the start-up of the child plan (except for a short lock-out for the event queue duplication).
- Replacement plan. The old plan is replaced by the new plan. The new plan inherits the execution session of the old plan, inherits the contents of the old input queue (if not specified otherwise by the Facilitator), and also inherits the active state. In case there is no matching state in the new plan for the old active state (e.g. the matching state has been removed by the plan modification steps), the new plan is started from its initial state.

The new models make it possible to set up more specific client-handling models: for example a separate plan for each communication partner.

Independent new plans can be created with the common functionality "`start_plan_service`". Child plans can be created using the common functionality "`start_child_plan_service`". Replacement plans are announced by the Facilitator based on the plan modification rules.

Table 1 summarises the differences between plan types.

Plan type	Execution session	Input queue	Starts from state
Independent plan	New	New	Initial state
Child plan	Inherited from the parent	Copy of the parent's	Initial state
Replacement plan	Inherited from the parent	Copy of the parent's	Old plan's active state (if possible)

Table 1: Plan types

An already active plan can be stopped in two ways:

- Calling the common functionality `stop_plan_service` with the id of the plan which should be stopped.
- Using `stopPlan` expression within the plan creation or modification rules as presented in the example below.

```
<Implies closure="universal">
  <And>
    <Atom closure="universal">
      <Rel>?event</Rel>
      <Ind>cascadas.ace.gateway.CancelContractEvent</Ind>
    </Atom>
  </And>
  <Atom closure="universal">
    <Rel>stopPlan</Rel>
    <Ind>plan1</Ind>
  </Atom>
</Implies>
```

2.3 Extended Self Model Syntax

2.3.1 Recursive-referential Syntax in the Self Model

The aim of the extension was to support more dynamic, template-based self-model creation. From now on, parameter names can be not just strings but also dynamically resolved references.

An example is: `?executionSession://?inputMessage://paramName`. The resolution happens iteratively, from right to left. So, if the current value of

"`?inputMessage://paramName` " is resolved to "`weight`",

then "`?executionSession://?inputMessage://paramName`"

is equivalent to "`?executionSession://weight`".

The binding is dynamic: the result of the internal resolution step is not cached. This may cause some overhead in the processing time but implicitly avoids consistency problems that easily may occur otherwise.

The new syntax is available for the following variables:

- ?executionSession
- ?globalSession
- ?inputMessage
- ?contextData.

The number of reference levels is theoretically not limited. For example, this is a 4-level recursion:

"`?executionSession://?globalSession://?contextData://?inputMessage://x`"

Changes are backward compatible (so no code modification is needed in existing applications).

The new syntax is automatically available for guard conditions and actions.

2.3.2 New Guard Condition Operators

The set of built-in guard condition operators has been extended with new text processing primitives. The changes were motivated by requests from other WPs which wanted to use text processing guards besides the already available logical and mathematical conditions.

The new operators are the following:

- STARTWITH(s1, s2). True if s1 and s2 are non-null Strings, and s1 starts with s2.
- ENDSWITH(s1, s2). True if s1 and s2 are non-null Strings, and s1 ends with s2.
- CONTAINS(s1, s2). True if s1 and s2 are non-null Strings, and s1 contains s2.

Operators are case-sensitive.

If any of the parameters cannot be interpreted as a String, or their value is null, the compilation returns false.

2.4 New Toolkit Features to Support Supervision

This paragraph summarises a number of new features of the ACE Toolkit to support supervision. A more detailed description can be found in [D2.4].

2.4.1 Interrogation Function

New interrogation functions based on the interaction of a potential Bus Checker Object (BCO) and the Executor and Repository ACE organs have been provided. Recall that a supervisor is capable of deploying Supervision Checker Objects for the Bus and Gateway into a supervised ACE. A BCO is able to introduce new events to the internal communication Bus, and intercept events travelling on it. Interrogation functions make use of this feature by defining a number of events containing requests for information which are handled by the Executor and – in one case – by the Repository organ. These organs

respond by issuing events containing the requested information. These responsive events are again intercepted by the BCO, and delivered to a Sensor ACE associated with the supervised ACE under consideration. Requests are:

1. Currently executed plans
2. States in which those plans are
3. State of the input queues of each of the executed plans
4. Global session and execution session
5. Events which are possibly issued by a specific repository function (those requests are handled by the Repository organ utilising a special interface provided by the Output Mapper of a user defined function).

2.4.2 Advanced Control

Model based supervision results in the computation of non-local coordinated contingency plans to define a collective course of actions which are intended to lead an ACE based configuration from a problem state back into normal operation. Actions may be

1. Passive: Prohibit a certain transition from being executed
2. Active: Enforce a certain transition by issuing a triggering event

Since transitions are enabled in specific plan states, a contingency plan local to a certain plan execution can be expressed as a pair of mappings (p, a), where p: *State* → **Set**<*Transition*> maps a plan state into the set of transitions allowed to be executed in this state, and a: *State* → **Set**<*Event*> defines the set of events to be issued by a supervisor in order to trigger certain transitions. These events are derived from a dedicated base class (namely SupervisionControlEvent) which is also used by the planning algorithm to identify those events which are intended to be actively controlled by the supervisor.

Passive control (i.e. an interpretation of the function p) is realised by an extension of the plan executor implementation, which checks the current state s whether a transition to be executed is in the set $p(s)$. Active control (i.e. an interpretation of the function p) is done by the BCO which checks for the current state s whether the set $a(s)$ is not empty, and if so, introduces events for all elements of $a(s)$ to the internal communication Bus, where they will travel to the plan executor instances and will be handled there.

2.5 ACE Cloning

Within the latest research period the ACE lifecycle management has been enhanced. Beside init, start, stop, reset, and destroy functionality, a cloning ability has been developed. In the context of the CASCADAS Toolkit, cloning is defined as the process of creating an exact copy of an already existing ACE while that ACE is being executed. Therewith, we differentiate between shallow and deep cloning.

Shallow cloning means that the ACE copy created is of exactly the same ACE type, has the same functionalities as its parent, but does not inherit runtime specific parameters like active plans and their current states, queues including messages, or contracts. In consequence, the clone is able to fulfil the same tasks and to provide the same services as its parent. It will start its execution in an initial state and has no information about past situations.

Deep cloning, on the other hand, creates a copy of an ACE that inherits ACE type, functionalities and runtime properties. Thereafter, the clone starts all plans that have been

active in its parent, in exactly the same state in which the parent ACE will continue as well. The new ACE knows about enqueued messages and established contracts.

In the Toolkit release accompanying this deliverable, we have implemented shallow cloning. That means ACEs can be duplicated during runtime on demand so that the clones start in their initial state. The common functionality *request_life_cycle* can be used therefore in the following way.

```
<transition id="tr1">
      <source>state1</source>
      <destination>state2</destination>
      <priority/>
      <trigger>@auto</trigger>
      <guard_condition/>
      <action>request_life_cycle(action=clone)</action>
</transition>
```

Each ACE clone gets a new name <newName>, which is derived from its parent name <oldName> by adding ":clone" and a number <x> that represents how often the parent ACE was cloned.

<newName> = <oldName>:clone<x>

The diagram in Figure 1 shows which lifecycle actions can be requested depending on the lifecycle state. Moving and deep cloning of ACEs are currently under development and not yet supported. Shallow cloning can be requested when an ACE is in the lifecycle state *running*.

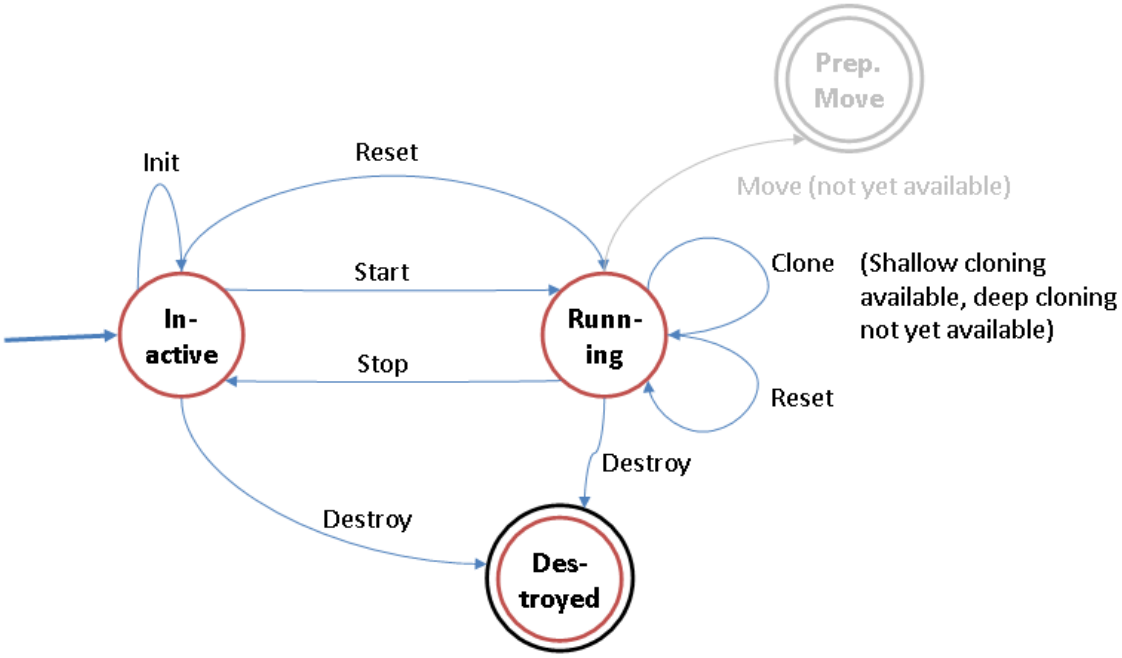

Figure 1: State diagram of the Manager for lifecycle action requests

2.6 Facilitator Enhancements

The Facilitator organ is responsible for creating ACE plans and adapting the ACE behaviour with respect to new requirements. The ACE plan is a predefined sequence of actions which have to be performed in order to fulfil a certain goal. The behaviour of an ACE is defined by the developer within the self model.

During the ACE initiation phase, the Facilitator loads and verifies the self model. On ACE start-up it creates the default plan and from there on it continuously evaluates the self model and takes the desired actions. These actions comprise creation of a new ACE plan and modifications to existing ACE plans.

The Facilitator continuously listens to changes in the ACE environment and the ACE operation status. Information about the ACE environment is gathered from so called context provider ACEs who provide contextual information about the environment. Monitoring the ACE operation happens via the event bus. After analysing the ACE self model, the Facilitator subscribes to the events defined within the plan creation and modification rules. Each time an event of the desired type has arrived on the bus, it will be applied to the rule set and evaluated. For more detailed description of the Facilitator organ please look at chapter 3.4.1 in the deliverable [D1.3].

The Facilitator organ has been enhanced in two ways. Firstly, it allows developers not only to specify types of events upon which an action has to be taken, but also to extract parameters from these events and compare them independently if these parameters are of simple or complex type. Secondly, it allows developers to specify in the self model the different levels of security for different services while specifying the conditions under which the communication between two ACEs needs to be encrypted.

Using `?event` within the plan creation and modification rules in the self model, ACE developers can specify any particular type of event to be considered. When reading the self model, the Facilitator will extract all relevant event types and will subscribe for receiving them. As soon as one of the defined events arrives on the bus, it will be applied together with the current context data as a fact to the RuleML. If the new facts satisfy the rules and conditions, the defined actions will be taken.

```
<Implies closure="universal">
  <And>
    <Atom closure="universal">
      <Rel>?event</Rel>
      <Ind>cascadas.ace.gateway.CancelContractEvent</Ind>
    </Atom>
  </And>
  <Atom closure="universal">
    <Rel>startPlan</Rel>
    <Ind>plan1</Ind>
  </Atom>
</Implies>
```

The rule above states that if an event of type `cascadas.ace.gateway.CancelContractEvent` arrives on the bus, the plan with the `id=plan1` should be started.

As mentioned above, the Facilitator allows developers to not only listen to events, but also to extract parameters from them, and compare their values with parameters stored in session objects. The Facilitator can compare among parameters of all types, complex as well as simple types.

The Facilitator can listen to any type of event regardless of whether it is a standard or developer specific event. In order to allow the Facilitator to extract the parameters from it, the event class must implement the `cascadas.ace.event.ParameterizedEvent` interface. The Facilitator will get the parameter value using the `getParam(String key)` method and will apply it to the RuleML. Within the self model, the parameter value can be requested using `?event://paramName`. Please note that there must be a parameter with such a name inside the event. Otherwise a null value will be applied.

```
<Implies closure="universal">
  <And>
    <Atom closure="universal">
      <Rel>?event</Rel>
      <Ind>cascadas.ace.gateway.CancelContractEvent</Ind>
    </Atom>
    <Atom closure="universal">
      <Rel>?event://contract</Rel>
      <Ind>?executionSession://myContract</Ind>
    </Atom>
  </And>
  <Atom closure="universal">
    <Rel>createTransition</Rel>
    <Ind>tr7a</Ind>
    <Ind>tr7b</Ind>
  </Atom>
</Implies>
```

The rule above can be interpreted in the following way: If a `cascadas.ace.gateway.CancelContractEvent` arrives on the bus and the cancelled contract `?event://contract` equals `?executionSession://myContract` than two new transitions `tr7a` and `tr7b` should be created.

As already mentioned, within the self model developers can define different levels of security for different services while specifying the conditions under which the access to an ACE service or the entire communication between two ACEs needs to be encrypted. These conditions can be either security relevant context data which is provided by context provider ACEs or the local events reflecting the ACE operation which could lead to the conclusion that an attack is taking place (e.g. unexpected or unknown service request).

If an ACE decides to provide its service in an encrypted way, it has to find and contact the confidentiality ACE. Before sending any service request or response, the payload has to be encrypted by the confidentiality ACE first. There is a parameter named `encrypted` which is located at the `ServiceUsageEvent` class which is set by default to `false`. Both, the service request and the service response can be encrypted if required.

Using guarding conditions, a receiving ACE can check if the message payload is encrypted or not. If the message is not encrypted, it will start with the regular operation and in case of an encrypted message, it has to start the decryption plan which will locate the confidentiality ACE, decrypt the message and pass the decrypted payload back to the original plan for further processing.

```
<transition id="tr4">
  <source>state2</source>
  <destination>state3</destination>
  <priority>1</priority>
  <trigger>cascadas.ace.event.ResponseCallEvent</trigger>
 <guard_condition>EQUALS(?inputMessage://encrypted,true)</guard_condition>
  <action>start_child_plan_service(planID=decryptionPlan)</action>
</transition>
<transition id="tr5">
  <source>state2</source>
  <destination>state4</destination>
  <priority>2</priority>
  <trigger>cascadas.ace.event.ResponseCallEvent</trigger>
<guard_condition>EQUALS(?inputMessage://encrypted,false)</guard_condition>
  <action>process_service_response</action>
</transition>
```

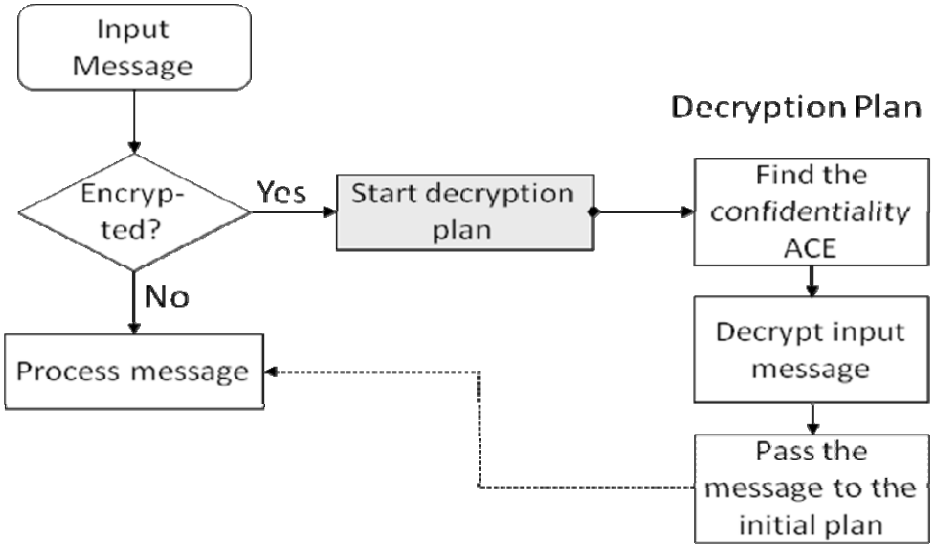

Figure 2: Input message decryption process

As depicted in the example above, if the incoming message is encrypted, a decryption plan will be started and the message will be decrypted. The decrypted payload will then be processed in a regular way.

3 Mobile Device Support

In order to use the ACE Toolkit, a device that is capable of executing a Java virtual machine is needed. In detail, Java 2 Standard Edition, Version 5.0 is the required runtime environment. Any operating system and underlying hardware architecture that supports such execution environment can host ACE based applications without any specific limitation. To reach unhindered performance, the device on which ACEs will run is recommended to be equipped with at least 128MB RAM and a sufficiently fast processor, (i.e., 500MHz or better). In any case, the performance of applications depends on the number of executed ACEs and their overall computational complexity.

Besides the Toolkit itself, the following set of libraries needs to be installed. All of them are Java 2 Standard Edition, Version 5.0 compliant.

- OO jDREW (Object-oriented Java Deductive Reasoning Engine for the Web), implementation of a reasoning engine used for plan creation
- XOM (XML object model), a library for xml handling
- REDS (REconfigurable Dispatching System), a publish-subscribe messaging system used for GN-GA communication
- DIET Agents (Decentralised Information Ecosystem Technologies), the underlying agent platform technology
- Log4j, a logging system used by OOjDREW
- Code from the Ptolemy project, for handling graph data structures

In addition to the above mentioned devices, we evaluated how to adapt the Toolkit in order to enable its execution on other mobile devices. The following sections describe three approaches that we have researched to enable the use of mobile computing units. Section 3.1 presents a set of machines that are natively supported without major changes and

explains what is necessary to execute the Toolkit on a Nokia N800 device. In section 3.2 we elaborate on how to adapt the Toolkit to Google's Android platform. Section 3.3 depicts the approach of extending the ACE concept in order to utilise java ME based devices like smart phones and PDAs.

3.1 Natively Supported Mobile Devices

The ACE Toolkit supports a variety of mobile devices. In addition to standard laptop computers, we evaluated the devices listed in Table 2 (ordered from the most to the least powerful in terms of storage and processing capabilities). All of them fulfil the minimal hardware and software requirements listed above. Hence, they are capable of executing ACE based applications without restrictions.

Device Type	Characterisation	Comment
IBM X41 Tablet PC	Intel Pentium M, 1.5GHz, 1,5GB RAM Microsoft Windows XP Tablet PC Edition	
Motion Computing LS800 Tablet PC	Intel Pentium M, 1.2GHz, 1GB RAM Microsoft Windows XP Tablet PC Edition	
Sony Vaio VGN-U71P	Intel Pentium M, 1.1GHz, 504MB RAM Microsoft Windows XP Professional	
OQO 01+	Transmeta Crusoe 1GHz, 512MB RAM Microsoft Windows XP Tablet PC Edition	Slow with stability issues.
Panel PC Wincomm WLP-6820	VIA Nehemiah, 733MHz, 224MB RAM Microsoft Windows XP Professional	Very slow, only suitable for a small number of ACEs running in parallel; No battery, permanent power supply necessary

Table 2: Mobile devices that natively support ACE based applications

In order to extend the number of natively supported mobile devices, additional research efforts were spent towards porting the ACE Toolkit to non standard Java implementations. Since the ACE Toolkit code as well as the required libraries were developed using Java 1.5, our goal was to identify Java implementations for mobile devices which natively support Java 1.5 code and to adapt and test ACE Toolkit implementation for these devices.

Because Sun Microsystems itself does not offer a standard Java implementation for mobile devices an alternative java implementation was required. GNU Classpath (see [CLASSPATH]), an open source implementation of the standard class library for the Java programming language that has been successfully tested on a series of mobile devices,

appeared to be an appropriate one. It is available under the GPL license and can be downloaded from www.gnu.org/software/classpath.

GNU Classpath provides an open source implementation of the standard class library for Java programming language but in order to execute the Java code a Java Virtual Machine (JVM) is additionally required. A JVM uses Java class library for interpreting and executing Java classes. There are several JVMs which use GNU Classpath as their library implementation. Some of them are: AegisVM, GCJ, JamVM, Jaos, Kaffe, SableVM, CACAO and many more. For porting the ACE Toolkit we have chosen the CACAO JVM [CACAOJVM] because of its light implementation and Just in Time (JIT) compiler which allows fast Java code execution on devices with limited processing power.

CACAO is a research Java Virtual Machine developed at the Vienna University of Technology. It has a compile-only approach, which means there is no interpreter at all but a JIT compiler ported to different architectures, like Alpha, i386, ARM, MIPS, PowerPC, and x86_64. CACAO is free software distributed under the GNU General Public License.

The ACE Toolkit has been successfully tested using CACAO JVM and GNU Classpath on a Nokia N800 mobile device. Since GNU Classpath 0.97.2 does not contain a full implementation of the Java 1.5 class library, additional modifications to the ACE Toolkit have been applied. For this reason, when creating ACEs using GNU Classpath, the following limitation applies:

- Logging functionality is limited to console logging only.
- Limited GUI capabilities available. Only AWT is supported.
- XML Schema validation which is used for validation of self models is not supported.
- Check GNU Classpath release notes for further limitations.

Figure 3: ACE Toolkit on Nokia N800

In addition to the work described above, minor modifications and improvements have been applied to the Reconfigurable Dispatching System middleware (cf. [REDS]). As already mentioned in deliverables [D1.2] and [D1.3], ACEs use REDS middleware for service discovery purposes. An ACE sends a goal needed message over REDS in order to find another ACE that can fulfil the goal. Because REDS was not initially developed to run on mobile devices, we experienced some instability issues when using it on Nokia N800. After some time, the REDS communication channels were closed because of a socket error which caused the ACE to not being able to send or receive GN/GA messages. The REDS

middleware has been improved such that the socket errors are handled internally while re-establishing the socket when closed.

3.2 Google Android Platform

The Android platform [ANDROID], developed by Google Inc. and the Open Handset Alliance[1], is a software stack for mobile devices including an operating system, middleware and key applications. Developers can create applications for the platform using the Android SDK. Applications are written using the Java programming language and run on Dalvik, a custom virtual machine designed for embedded use which runs on top of a Linux kernel. A preview release of the Android software development kit has been available since the end of 2007, which includes development and debugging tools, a set of libraries, a device emulator (see Figure 4), documentation and more. According to the Google Inc. the first Android compatible mobile devices should be available by the end of the year.

Using its very innovative ideas, Google has a very large impact on the IT industry. It is expected that with the Android platform Google will have at least the same impact to the telecommunication market as well. For this reason our goal was to provide a full version of the ACE Toolkit for the Google Android platform.

Even though the home page explicitly states that they are "offering an **early look** at the Android Software Development Kit", there has been some criticism about the instability and the many bugs of the SDK[2]. We believe that in the near future Google will improve its reliability and that the implementation on mobile devices will be much more stable than the emulator one.

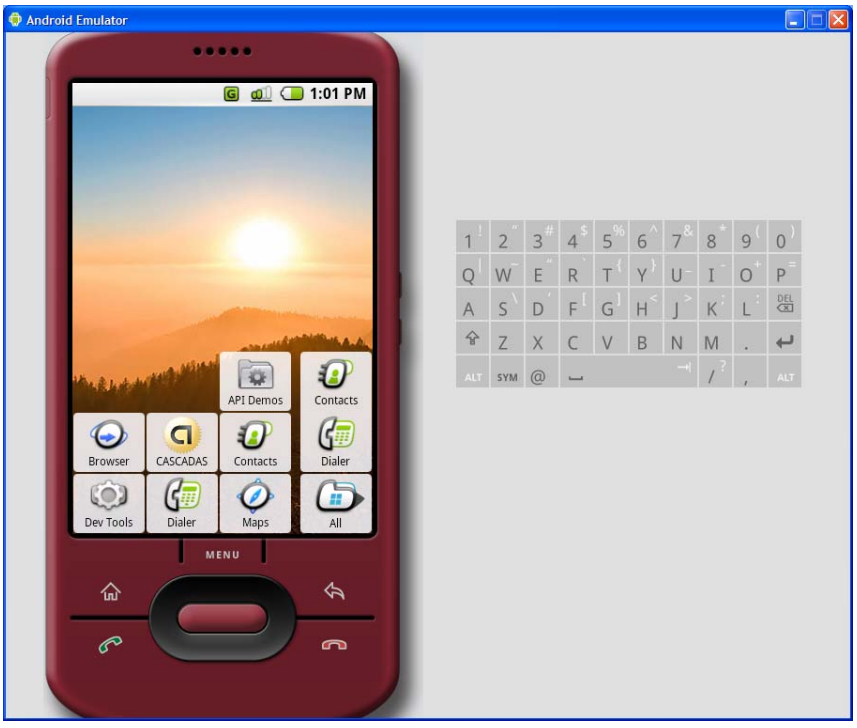

Figure 4: Android emulator

[1] http://www.openhandsetalliance.com/

[2] http://arstechnica.com/news.ars/post/20071219-google-android-plagued-by-dysfunctional-development-process.html

Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language. Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimised for minimal memory footprint, and relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

The platform includes a set of C/C++ libraries used by various components of the Android system. These capabilities are exposed to developers through the Android application framework. Some of the libraries are: System C library (a BSD-derived implementation of the standard C system library, tuned for embedded Linux-based devices), Media Libraries, that support playback and recording of many audio and video formats, LibWebCore (a web browser engine which powers both the Android browser and an embeddable web view), SGL (the underlying 2D graphics engine), FreeType (bitmap and vector font rendering) and SQLite, a lightweight relational database engine available to all applications.

On Android platform, developers have access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components and allows components to be replaced by the user. Underlying all applications is a set of services and systems, including for example an extensible set of Views that can be used to build an application (including lists, grids, text boxes, buttons, and even an embeddable web browser), a Resource Manager, providing access to non-code resources such as localized strings, graphics, and layout files, and an Activity Manager that manages the life cycle of applications and provides a common navigation back stack.

The most important development tools that come with the SDK are:

- Android Emulator - A virtual mobile device that runs on the host computer. The emulator is used to design, debug, and test applications in an actual Android run-time environment.
- Android Development Tools (ADT) Plugin for the Eclipse IDE.
- Dalvik Debug Monitor Service (ddms) - Integrated with Dalvik VM, this tool lets the programmer manage processes on an emulator or device and assists in debugging.
- Android Debug Bridge (adb) - The adb tool lets you install your application's files on an emulator or device and access the emulator or device from a command line. It can also be used to link a standard debugger to application code running on an Android emulator or device.

Programmers are free to use command line tools to create, build and debug Android applications, but the best way is to use the Eclipse 3.2 or later IDE through Android Development Tools Plugin.

3.2.1 Status of ACE Toolkit on Android

Porting of the ACE toolkit to Android is in progress. We are working on the core source files that are incompatible with the platform capabilities. For example all GUI related has to be removed because Android does not support Swing or AWT, but has its own graphic packages.

Android applications are written with some combination of four building blocks:

- Activity

- Intent Receiver
- Service
- Content Provider

An activity[3] is usually a single screen in the application. Each activity is implemented as a single class that extends the Activity base class. Most applications consist of multiple screens: in this case, each of these screens would be implemented as an activity, and moving to another screen is accomplished by a starting a new activity.

Android uses a special class called Intent to move from screen to screen. This is accomplished by resolving intents, by means of a related class called an IntentFilter. While intent is effectively a request to do something, an intent filter is a description of what intents an activity is capable of handling. The class IntentReceiver is used when some code in the application must be executed in reaction to an external event.

A Service[4] is code that is long-lived and runs without a UI. A content provider is useful to make the application's data available to be shared with other applications. It is a class that implements a standard set of methods to let other applications store and retrieve the type of data that is handled by that content provider. Obviously not every application needs to have all four building blocks.

Aiming to test the correct functionality of all the components of the ACE Toolkit, we began by linking the toolkit to a generic Activity. This is just a temporary development solution since the Toolkit on Android should be implemented as a service and the issue of how to link UI samples (probably starting new Activities) to the core ACE toolkit has not been addressed yet.

Several problems have been solved, but there are still issues to face. Currently, we can compile and execute a version of the core ACE Toolkit, going through the initialisation phase and instantiating all the organs, reading configuration files, self-model and functionalities of the ACE. The facilitator organ is able to parse the self-model and create plans, which are run by the executor organ starting local actions if necessary. The main thing that is missing is the communication part since message delivery by REDS is not working, and there are also other parts which have not been tested yet.

The most important changes/updates done so far on the ACE Toolkit source code are the following:

- Modifications applied to several DIET classes to fit Android requirements (to be tested).
- Changes to configuration file management. Due to the security model of Android, and since there are differences in the way the classes of the package java.io operate (see 3.2.2), all the configuration, self-model and functionality files are deployed on the emulator separately from the ACE toolkit application to minimise the modifications in the source code.
- XML management: this is one of the areas where Android lacks most compared to JDK 1.5 packages. Part of the code responsible for finding and using a SAX2 parser had to be changed, and serialization of DOM objects had to be completely rewritten.

[3] http://code.google.com/android/reference/android/app/Activity.html
[4] http://code.google.com/android/reference/android/app/Service.html

Next steps will be finding a solution to the communication problems and writing some sample UI.

3.2.2 Running the ACE Toolkit

This section gives some basic information on how to execute the current version of the ACE Toolkit on the Android emulator. We consider a Windows XP system with Eclipse 3.3 and the Android Development Tools (ADT) Plugin for the Eclipse IDE (see http://code.google.com/android/intro/installing.html for details on installation).

The Android emulator can be started directly from the File Manager double clicking on the `emulator.exe` file in the `<path_to_Android_SDK>\tools` directory, or from a Command Prompt (it's useful to add the path to the SDK `tools` directory to the developer's path), or from Eclipse, when the ADT plugin is installed. The ADT plugin, which adds integrated support for Android projects and tools, includes a variety of powerful extensions that make creating, running, and debugging Android applications faster and easier, so we refer to an environment in which it is installed in the following.

When the emulator is running, it is possible to use the Android Debug Bridge (adb) to get a shell on the Android virtual machine with the line:

```
C:\> adb -d emulator-tcp-5555 shell
```

Even if the commands available are quite limited, it is useful to issue `ps` commands to see the active applications or to terminate a process with the command `kill -9 <pid>`. To inspect or change the file system the usual *nix commands can be used.

To make a fresh start and erase an older version of an application, the developer can issue the command:

```
C:\> emulator -wipe-data
```

Warning: this resets the emulator to its initial state, so that all files written to the file system and the deployed applications are lost. It is useful to erase many files to overcome the limitations of the adb shell if just 1 or 2 applications are deployed.

The first step to execute or debug the ACE toolkit on the emulator is the upload of the configuration files to the emulator file system.

Android applications usually operate on private files associated to a Context's application package, using their Context's openFileInput() and openFileOutput() methods. This way files get written in the following folder on the emulator:

```
/data/data/<project_package>/files/
```

This folder appears when the application is first run and is erased when the application is undeployed. It is like a "working directory" for the openFileInput() and openFileOutput() functions, but it seems that path separators are not allowed inside their arguments, and the prefix "./" doesn't work in the filename.

Anyway, if the application just requires reading a file, Android allows the use of some of the standard Java functions taking the full path of the file as an argument (this is not clear from the documentation, anyway, and should be checked for next SDK releases).

For now it looks convenient to put configuration files in the `/data/data/conf` folder, outside the application sandbox, so that they are not erased every time the application gets compiled and deployed on the emulator. Configuration files upload can be done using the Android Debug Bridge from the command prompt with the line:

```
C:\> adb push <path_to_the_toolkit>\toolkit\conf /data/data/conf
```

Then, in the modified source code of the ACE toolkit for Android, properties and XML files are read prepending the files paths with a `BASE_CONF_PATH` constant set to `/data/data/`, while paths included in the configuration (for example in `ace-type.xml`) must become relative.

In Eclipse, with the ADT plugin, the Android project type becomes available with the same characteristics as a standard Java project. A new perspective is available (DDMS, that lets a developer inspect the status of Dalvik VM), but the developer can use the usual Java and Debug perspectives.

Here it is possible to set breakpoints, step over the code and inspect variables in the usual way. Since Android has its logging system and a standard ConsoleHandler does not work, for the time being the ACE logging has been patched to use a private log file in the application sandbox.

Using adb this file can be transferred from the emulator to the host PC for a better analysis:

```
C:\> adb pull /data/data/<project_package>/files/<logfile> <destination_file>
```

This is the current method to inspect execution results without debugging, since Android UI for the ACE Toolkit is not available yet.

3.3 TinyACE

The idea of the TinyACE is to develop a lightweight version of the ACE model which is less capable than the regular ACE, but which runs on java ME MIDP 2.0. This ACE could run on a large number of small devices possibly including SunSPOT sensor nodes.

The goal is to extend the spectrum of platforms where to run ACEs:

- The Standard ACE Toolkit runs on servers, PCs, and the mobile devices listed in section 3.1.
- An almost standard ACE Toolkit runs on Nokia N800.
- A TinyACE runs on small devices (MIDP 2.0 compliant).

3.3.1 TinyACE specification

The TinyACE model will take into account the ACE **communication mechanisms only**.

It will try to implement both the GN-GA communication mechanisms based on REDS and the contract-based communication based on DIET. **It will not implement the ACE internal architecture** (i.e., self model, etc.).

Accordingly the TinyACE model will not provide an organ-based architecture (as in the case of standard ACEs). It will only provide some classes offering methods to:

- Send/Receive GN-GA messages
- Establish/Send/Receive Contract

These classes can then be imported in any standard MIDP application that will use it as needed. Accordingly, TinyACE will be programmed following the MIDP programming model. Thus any autonomic behaviour will be eventually programmed directly in Java within the TinyACE code. From the outside, such a MIDP application resembles an ACE. From the inside it is a standard MIDP with no ACE features being supported. For example,

with regard to this point, a limitation of the TinyACE model is that, at least at the beginning, TinyACEs will not be supervisable. In fact, they will be just standard J2ME applications with some classes offering advanced communication mechanisms.

The development of this model will proceed in two main steps:

1. We already started the development of a hybrid solution in which the TinyACE communicates to other ACEs via a gateway (access point). The communication between TinyACE and the gateway is socket-based and relies on a simple exchange of strings (see section 3.3.1).
2. The development will continue in the direction of moving both GN-GA and contracts to the MIDP devices, possibly trying to eliminate the access point gateway.

3.3.2 First steps towards TinyACE

We conducted a survey of the J2ME technology with the aim of evaluating the feasibility of porting ACEs to MIDP devices. We have found the following:

- **TCP sockets** are available for most devices. (The support of TCP is recommended, but not required in the MIDP specification. However, most device manufacturers implement both, client and server socket.)
- **Multi-threading** is available as in the standard J2SE.
- No standard general-purpose **serialisation** is available and custom methods have to be developed to serialise and de-serialise objects.

To quickly start the development of the TinyACE model, we have developed a gateway-based solution: The gateway bridges any communication between TinyACEs and (normal) ACEs performing also a protocol conversion. Figure 5 and Figure 6 show the GN-GA interaction by means of the MIDP ACE gateway.

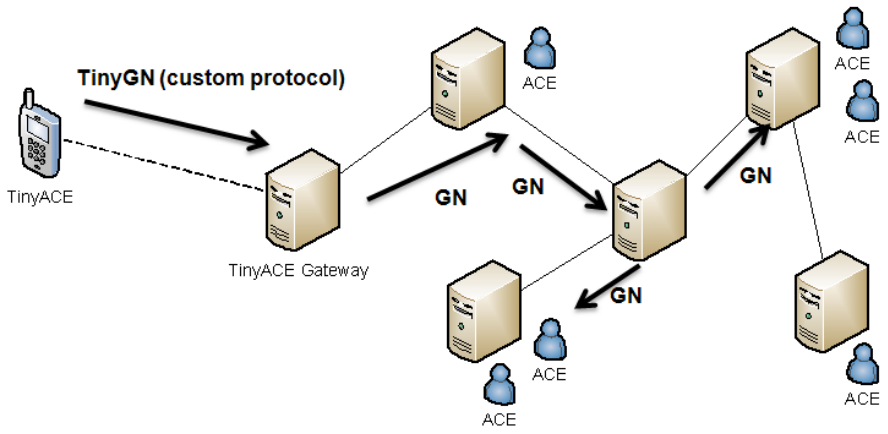

Figure 5: MIDP ACE GN Invocation

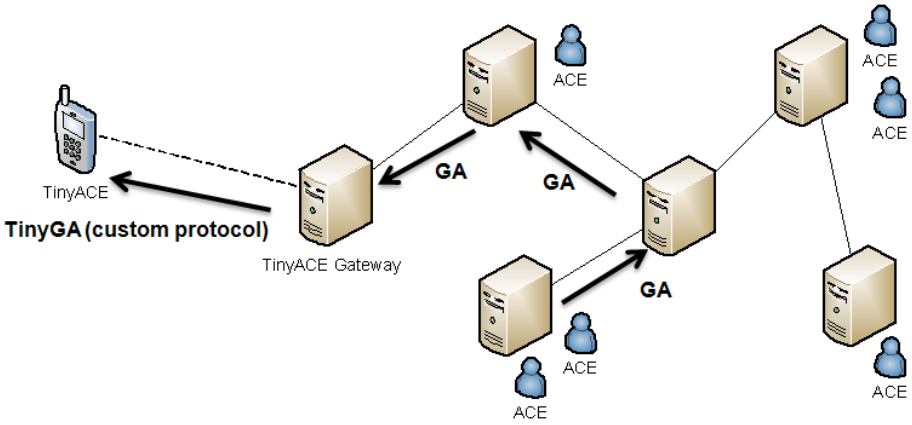

Figure 6: MIDP ACE GA reply

To test these ideas we have developed a simple demo in which an S60 phone communicates with other ACEs (see Figure 7).

Figure 7: S60 ACE client connecting to the ACE supporting MIDP

The development will continue in the direction of moving both GN-GA and contracts to the MIDP devices, eliminating the protocol conversion gateway.

To test the ACE MIDP extension, we are trying to develop an innovative case study scenario highlighting the MIDP support. In particular we would like to develop services in the direction of "Social Networking for Mobile Users". These services aim at integrating users' activities and preferences with real-time location data. Specifically, mobile devices will be able to collect and exchange information and data to perform certain tasks through distributed reasoning and self-organising algorithms.

- When and where can I meet my friends today?
- When and where should we go for a business dinner tonight?
- What event can I attend this afternoon?

These kinds of services could be easily integrated within the pervasive advertisement scenario. The case study will possibly illustrate autonomic principles by showing that resources are retrieved in a context-aware way. For example users looking for restaurants can be returned only nearby places with enough free tables. In this context the use of self-organisation clustering algorithms (WP3) to aggregate users depending on their location and on common interests in effective social networks will be studied.

4 Practical Guidelines

This chapter contains examples describing how to perform common tasks that frequently occur when programming with the ACE Toolkit.

4.1 Contracting

Communication in the ACE Toolkit can follow two different paradigms:

1. using a connectionless model (facilitating a Publish-Subscribe paradigm), meaning that messages are disseminated to all interested recipients: The GN-GA protocol builds on this paradigm
2. employing *contracts*, which define a set of connections between multiple parties, allowing each party to contact any other directly using role names; from an implementer's perspective, it is also possible to send messages to all participants at once

Contracts need to be established explicitly. The cancellation of a contract is done explicitly, as well, but may also happen in an implicit manner (e.g. due to a faulty network connection). When a contract is being cancelled, it is impossible for all parties involved to further employ it for communications. A new contract needs to be put in place before communication can be resumed. All events relating to contracts (establishment, cancellation and events transmitted using a contract) are visible in the self-model and can be used as triggers for transitions between ACE states, making it possible to implement arbitrary sorts of failure handling (e.g. failover, re-try, escalation, etc.) in the self-model.

To establish a contract, an ACE needs to gather a set of addresses. This may be done in any way imaginable, but it is most often done using the GN-GA protocol. An ACE would ask for a certain goal and receive a number of replies containing the addresses of the

ACEs able to provide the goal. It would then run a selection procedure on the answers and subsequently establish a contract with the selected ACEs.

The next sections describe how to establish a contract using either the ACELandic high-level language, or the more complex self-model XML syntax directly. The following example will only use two parties (client and provider), but the described mechanisms would be used in a similar manner when employing contracts with more than two parties involved.

4.1.1 Contracting Using ACELandic

Figure 8 depicts the flow of messages for establishing, using and cancelling a contract, along with the ACElandic statements that are employed to trigger or process these messages.

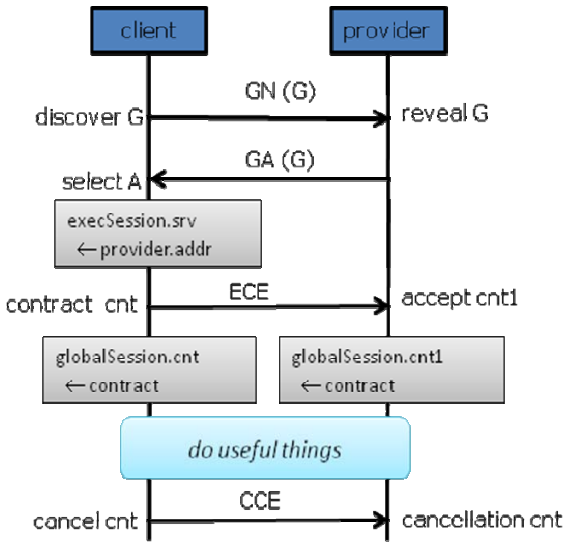

Figure 8: Contracting Using ACELandic

4.1.1.1 ACELandic on Client Side

The client first looks for other ACEs that might be contracted by employing the **discover** statement.

```
discover G {
   select srv {…}
}
```

This directive sends out a GN message with the specified *goal*, and listens for incoming GA messages in the subsequent `select` clauses. The statement sequence performed within the `select` clauses is for each incoming GA with the specified *goal*; the sender address of the GA is stored in the local execution session under the key ***srv***. The intention is that the select code can be used to determine an appropriate service provider. After storing the addresses, the client tries to establish a contract using the **contract** directive.

```
contract cnt [
  client <- self/address,
  provider <- local/srv
];
```

Here, ***cnt*** is the key under which the contract is stored in the global execution session. An optional **role** specification comprises a sequence of *role* / *address* mapping pairs, where

role is the role of an ACE in the specified contract, and *address* is an expression evaluating to the address of an ACE which assumes that role, e.g. ***client*** is a role name assigned to the address of the ACE executing the self model.

Once this is successful, the contract can be used to send events to the involved parties using contract identifier and role name. When the cooperation between ACEs is finished, the cancel directive is used.

```
cancel cnt;
```

4.1.1.2 ACELandic on Provider Side

The provider ACE starts looking for GN statements using the **reveal** statement.

```
reveal G;
```

The **reveal** directive responds to an incoming GN message, containing the specified *goal* G, with a matching GA. Once this is done the provider would want to **accept** the contract.

```
accept -> cnt1;
```

The optional *name* identifier ***cnt1*** initiates storage of the contract under the given name in the global session. Optionally a *contractor* expression can be given (not shown here) – it is evaluated to the address of an ACE and the contract is accepted only if it is initiated by a contract directive on the contractor's side. This enables an ACE to accept contracts based on the ACE that is offering them.

The contract would now be in place and ready for use. An optional **cancellation** block can be used to execute statements in case that a contract has been cancelled.

```
cancellation cnt1 {…}
```

4.1.2 Contracting Using the XML Self-Model Syntax

Contracting might also be done directly using the XML self model syntax. This is more complex, but gives the implementer maximal flexibility. We encourage developers to use the ACELandic approach as this saves one from a lot of hassles due to typos and XML formatting. Nonetheless, considering the pure XML form is worthwhile to understand the internal mechanisms that enable the contracting functionality.

4.1.2.1 Self Model on Client Side

Only transitions between the plan states are described, as state names are irrelevant for understanding the contracting mechanism. Furthermore, details that are not of interest have been substituted by the […] symbol. The following example is similar to the prior one in ACELandic, with the notable exception, that an additional timeout mechanism was specified. Such a mechanism can also be specified in ACELandic, please refer to the ACELandic specification [ACEL]. The timeout mechanism is used to detect that a sent GN has not been answered during a given time span. It is triggered by calling the **add_timer_service** from a transition action and supplying a timer identifier **tID** and a **timeout** value.

```
<transition id="SetTimer">
      […]
      <action>add_timer_service(timerId=tID,millis=timeout)</action>
</transition>
```

Following the setting of the timer, a GN is sent using the **gn_sender_service**. Parameters are the goal name **G** and the address of the ACE that executes the self model.

```
<transition id="SendGN">
     […]
     <action>
          gn_sender_service(
          goalName=G,
          myAddress=?globalSession://aceAddress)
     </action>
</transition>
```

The client ACE might introduce an additional transition that is activated once the previously specified timer is expired, enabling the ACE to handle this case, e.g. by re-transmitting the GN or choosing an alternative functionality:

```
<transition id="GNExpired">
     […]
     <trigger>cascadas.ace.event.TimerExpiredEvent</trigger>
     <guard_condition>EQUALS(?inputMessage://timerId,tID)</guard_condition>
     […]
</transition>
```

In a successful case, a previously sent GN would be answered by at least one corresponding GA. The GA triggers a transition if the included goal **G** equals the one the ACE is looking for. In the action part, the address of the GA originator is stored in the execution session under the key ***srv*** using the **add_to_execution_session_service** common functionality.

```
<transition id="ReceivedGA">
     […]
     <trigger>cascadas.ace.event.GoalAchievableEvent</trigger>
     <guard_condition>EQUALS(?inputMessage://goalName,G)</guard_condition>
     <action>
          add_to_execution_session_service(srv=?inputMessage://providerAddress)
     </action>
</transition>
```

After the address has been determined, the contract can be established using the **contract_n_establishment_service**, which takes a variable number of arguments. The last argument is **contractId**, a locally unique identifier for the contract (chosen by the implementer, in this case ***cnt***), whereas all leading arguments are role / address mapping pairs to be used for establishing the contract. Here two pairs are used: the **client** role is assigned to the address of the ACE that executes the current self model, whereas the **provider** role is mapped to the address that is stored in the execution session and was originally taken from the GA.

After successful establishment of the contract, the contract object can be found in the global, and execution session, under the chosen identifier. It can then be used to send messages to any party involved in the contract.

```
<transition id="EstablishContract">
     […]
     <action>
          contract_n_establishment_service(
          client=?globalSession://aceAddress,
          provider=?executionSession://srv,
          contractId=cnt)
     </action>
</transition>
```

Once the cooperation between ACEs is finished, a contract is cancelled using the **cancel_contract_service** common functionality.

```
<transition id="CancelContract">
     […]
     <action>cancel_contract_service(contract=?globalSession://cnt)</action>
</transition>
```

4.1.2.2 Self-Model on Provider Side

The provider ACE is expecting a GN that asks for the goal **G**, this is implemented by a transition that is triggered by the GN. When such an event is being encountered, the **gn_answer_service** is employed to send a matching GA back to the originator of the GA. A service name is also included that might be used by the GN originating ACE to select the proper contracting partner when processing the GA answers.

```
<transition id="ReceivedGN">
     […]
     <trigger>cascadas.ace.event.GoalNeededEvent</trigger>
     <guard_condition>EQUALS(?inputMessage://goalName,G)</guard_condition>
     <action>
          gn_answer_service(
          goal=?inputMessage://goal,
          serviceName=S,
          myAddress=?globalSession://aceAddress)
     </action>
</transition>
```

Once the contract has been established, an **EstablishContractEvent** is appearing on the bus of the provider ACE. This can be used to store the contract under the key ***cnt1*** in the global session, enabling the provider to send events to the contracted parties.

```
<transition id="ContractEstablished">
     […]
     <trigger>cascadas.ace.gateway.EstablishContractEvent</trigger>
     <action>add_to_global_session_service(cnt1=?inputMessage://contract)</action>
</transition>
```

Sending events is possible using the common functionality **send**, which needs two parameters: *event* (fully qualified Java class name of the event) and *contract* (the contract id to use). An optional third parameter *role* can be used to specify the target for the event, otherwise the event will be sent to all parties involved in the contract.

In case of cancellation of a contract, a **CancelContractEvent** appears on the bus. It can be used to trigger some cleanup functionality, e.g. for removing or re-establishing a contract.

```
<transition id="ContractCancelled">
     […]
     <trigger>cascadas.ace.gateway.CancelContractEvent</trigger>
     <guard_condition>
          EQUALS(?inputMessage://contract,?globalSession://cnt1)
     </guard_condition>
     […]
</transition>
```

4.2 Context Utilisation

Being aware of the environment is a very important ACE capability. The ability of adapting its behaviour with regard to changes in the environment, which is described in form of

context data, is one of the main ACE characteristics. An ACE might be interested in multiple types of contextual information: its own context such as processing capabilities available in the local environment, a user specific context like the user's location, a more general type of context like the current temperature in a city etc. But how does the ACE get this context information?

In terms of ACEs we distinguish between context provider and context consumer ACEs. Context provider ACEs provide simple context data like for example a sensor reading or more abstract context information based on context inference rules. Context consumer ACEs might use this information for example in order to adapt their behaviour with regard to the new conditions, or might require context information for service provisioning.

From the self model point of view, context data is defined using `?contextData://param` and can be used either within the plan creation and modification rules or within the plan itself as a transition condition or action input parameter.

4.2.1 Context Provider ACE

From its conceptual point of view, a context provider ACE is a regular ACE which provides access to a certain type of context information as service. A typical example would be an ACE with a specific functionality which can read sensor values from a sensing device. The goal which can be achieved by such a context provider ACE corresponds to the type of the accessed sensor data.

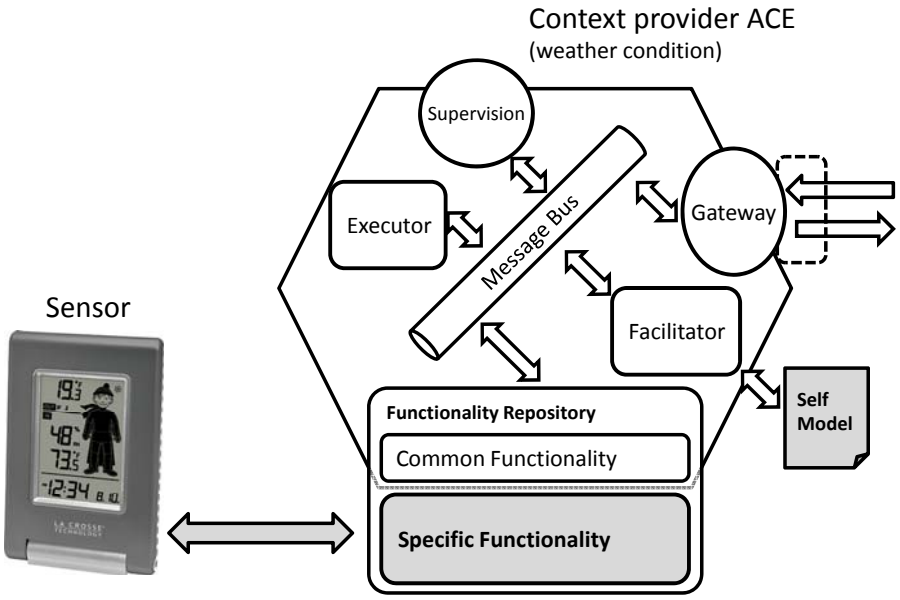

Figure 9: Context provider ACE conceptual view

From its self model point of view, the context provider ACE's self model is rather simple. The ACE simply waits for a GN message and answers with a GA message in case it is in line with its provided context data. The contract will be established by the context consumer ACE. The context provider ACE waits for service calls, and sends the latest sensor data to the context consumer ACE. In the current ACE Toolkit implementation, the context data access is utilised using the polling method.

The following example shows a simplified self model which can be used when creating context provider ACE.

```
<selfModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="../../../dtd/selfmodel.xsd">
  <plan id="Plan1" default="true">
    <description>
    </description>
    <states>
      <state id="state0">
        <friendly_name>ready</friendly_name>
        <desirability_level>1</desirability_level>
      </state>
      <state id="state1">
        <friendly_name>providing context data</friendly_name>
        <desirability_level>1</desirability_level>
      </state>
    </states>
    <transitions>
      <transition id="tr0">
        <source>state0</source>
        <destination>state1</destination>
        <priority>1</priority>
        <trigger>cascadas.ace.event.GoalNeededEvent</trigger>
        <guard_condition>
          EQUALS(?inputMessage://goalName,temperature)
        </guard_condition>
        <action>gn_answer_service(goal=?inputMessage://goal,
                                  serviceName=temperature_provider_service,
                                  myAddress=?globalSession://aceAddress)</action>
      </transition>
      <transition id="tr1">
        <source>state1</source>
        <destination>state1</destination>
        <priority>1</priority>
        <trigger>cascadas.ace.event.ServiceCallEvent</trigger>
        <guard_condition>
          EQUALS(?inputMessage://serviceName,temperature_provider_service)
        </guard_condition>
        <action>temperature_provider_service</action>
      </transition>
    </transitions>
    <creationRuleML>
      <Assert>
        <And>
          <Atom closure="universal">
            <Rel>createState</Rel>
            <Ind>state0</Ind>
            <Ind>state1</Ind>
          </Atom>
          <Atom closure="universal">
            <Rel>initState</Rel>
            <Ind>state0</Ind>
          </Atom>
          <Atom closure="universal">
            <Rel>createTransition</Rel>
            <Ind>tr0</Ind>
            <Ind>tr1</Ind>
          </Atom>
        </And>
      </Assert>
    </creationRuleML>
    <modificationRuleML>
    </modificationRuleML>
  </plan>
```

```
</selfModel>
```

4.2.2 Context Consumer ACE

The Context consumer ACE uses context information provided by the context provider ACEs for service provisioning or in order to adapt its behaviour. Within the ACE self model, context data can be accessed using `?contextData://contextName`, but before it can be read, it must be acquired first. An ACE which requires context data needs firstly to find and contract context provider ACEs that can provide the required context data. Subsequently, it has to start polling the context data from them. This is done using the `Context_acquisition_service` which is a common functionality that periodically polls context data from a context provider ACE. These two steps are usually carried out in the self model's initial plan, where all required context information is acquired first to make it available for all other follow-up plans which might use the context data.

In order to find a desired context provider ACE, the context consumer ACE sends a GN message containing the name of the needed context. The context provider ACE will answer with a GA message containing its address upon which a contract between these two ACEs can be established.

```
<transition id="tr1">
  <source>state1</source>
  <destination>state2</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition></guard_condition>
  <action>gn_sender_service(goalName=temperature,
                            myAddress=?globalSession://aceAddress)</action>
</transition>
<transition id="tr2">
  <source>state2</source>
  <destination>state3</destination>
  <priority>1</priority>
  <trigger>cascadas.ace.event.GoalAchievableEvent</trigger>
  <guard_condition>EQUALS(?inputMessage://goalName,temperature)</guard_condition>
  <action>contract_n_establishment_service(user=?globalSession://aceAddress,
                                provider=?inputMessage://providerAddress,
                                contractId=temperatureContract)
  </action>
</transition>
```

As soon as the desired context provider ACE has been found and a contract to it successfully established, the periodical context data polling process can begin. The `Context_acquisition_service` requires four input parameters:

- query_interval context data polling interval
- service_name name of the service to be called
- variable_local_name context name in the ?contextData://paramName
- contract contract which should be used to acquire the context

```
<transition id = "tr3">
  <source>state3</source>
  <destination>state4</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition/>
  <action>Context_acquisition_service(query_interval=1000,
```

```
                                service_name=?inputMessage://serviceName,
                                variable_local_name=temperature,
                                contract=?executionSession://temperatureContract)
  </action>
</transition>
```

In the example above, temperature will be polled form the context provider ACE every 1000ms and the new value will be stored to `?contextData://temperature`. Every time the context value changes, the new context value will be provided at `?contextData://temperature`.

The context data can be used within the ACE plan as an input parameter for an action or a transition condition, or it can be used as the parameter within the plan creation or modification rules.

```
<transition id = "tr3">
  <source>state3</source>
  <destination>state4</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition>GT(?contextData://temperature,17)</guard_condition>
  <action>my_service(local_temperature=?contextData://temperature)
  </action>
</transition>
```

Here is an example showing how to use context data within the plan creation and modification rules.

```
<Implies closure="universal">
  <And>
    <Atom closure="universal">
      <Rel>?contextData://temperature</Rel>
      <Ind>22</Ind>
    </Atom>
  </And>
  <Atom closure="universal">
    <Rel>createTransition</Rel>
    <Ind>tr7a</Ind>
    <Ind>tr7b</Ind>
  </Atom>
</Implies>
```

4.3 Settings and Configuration Guidelines

This chapter describes different ACE settings and application configuration guidelines. Most of these parameters can be specified within the *settings.properties* file which is the main ACE application configuration file.

4.3.1 The REDS Registry

The Toolkit is based on the REDS publish-subscribe middleware to send GN and GA events. REDS employs so-called *Broker* objects that take care of routing messages from sender to receivers. Each Java virtual machine will start its own broker as a singleton whenever the AceFactory is created. For a distributed application to work properly, all existing brokers need to find each other. The current implementation employs a **Broker Registry** object for this purpose. This *Registry* needs to be running and all started ACEs need to be configured with its address.

To run the registry

Start the **cascadas.ace.gateway.brokerregistry.BrokerRegistryServer** class contained in the Toolkit distribution. This will create a Broker Registry on port 8080.

To configure the ACEs

Set the **reds.registry** property in the **settings.properties** configuration file to a URL that points to the started registry, e.g. if you started the registry on a machine with the name "*snoopy",* a URL like **http://snoopy:8080/** would need to be used. Two things are very important here: Name resolution for the machine you are using must work over the complete domain where the application is run, and the URL string has to be terminated with a trailing slash.

4.3.2 DIET Parameters

Several parameters in the **settings.properties** configuration file control the behaviour of the ACE Toolkit, notably the DIET configuration parameters. DIET Agents ([DIET]) are used as the underlying technology for the ACE Toolkit. ACEs are implemented using DIET agents.

DIET uses a so-called *fail-fast* approach: The communication mechanisms are constantly checked against constraints set forth in the configuration parameters (e.g. buffer sizes or timeout values) and exceptions are raised once a constraint is breached. Such an approach is valuable for an environment with limited resources and enables a fine-grained control over the resources that DIET agents utilise.

A symptom that can often be seen when these constraints are breached is the implicit cancellation of contracts. For example: A network is congested and an event could not be sent within a given time-frame. DIET will then close the connection, which in turn implicitly cancels the contract using this connection.

To deal with such issues, it is either possible to define self-models that explicitly handle the cancellation of contracts (e.g. re-establish them), or alternatively to modify the DIET constraints. Of course this can only be done if there are enough resources at hand to fulfil the modified constraints. The following table describes all DIET configuration options available through the configuration file.

Property	Default Value	Description
diet.mirror.channel.timeout	10000	Allowed idle time of the DIET communication channel for mirror creation (in ms)
diet.mirror.channel.ping	3000	Defines the period for cyclic channel pings in DIET mirrors (in ms)
diet.mirror.setup.timeout	5000	Maximum time allowed for mirror setup (in ms)
diet.socket.timeout	10000	Maximum idle Time for DIET sockets (in ms)
diet.socket.byeack.timeout	2000	Time allowed for DIET agents to acknowledge a bye message (in ms)
diet.socket.max	100	Maximal number of sockets used by DIET
diet.thread.max	200	Maximal number of threads used by DIET
diet.messagechannel.buffer	100	Maximum number of packets to buffer for DIET

		message channels
diet.connection.max	50	Maximum number of connections a DIET agent may have (Needs to be greater than diet.connection.buffer)
diet.connection.buffer	10	Maximum number of connections requests to buffer for DIET message channels
diet.disconnection.buffer	10	Maximum number of disconnection requests to buffer for DIET message channels

4.4 Distributed Application Guidelines

Using the Toolkit to create distributed ACE based applications can be achieved by defining and configuring a number of local applications that are supposed to work together, and connecting these applications with the help of a REDS registry. Each local application comprises that subset of all ACEs which will run on one device, i.e. in one Java virtual machine. How a stand-alone application can be configured and started is explained in detail in section 6.3 of [D1.3].

In order to create a distributed application the following issues are very important:

- IP routing between the devices running ACE worlds must be enabled.
- In order to enable ACE distributed discovery, local REDS brokers must be assigned to the appropriate REDS registry.
- If a device contains more than one network interface, the appropriate one has to be specified. By default, the ACE Toolkit will choose the first network interface that has been discovered.
- In order to avoid socket timeout exceptions when using wireless communication, the diet.socket.timeout parameter has to be set properly.

To enable communication between the single applications a REDS registry is required. Section 4.3.1 of this document describes how to start up the REDS registry and how to put into practice the REDS based communication. Every ACE world needs to be configured in the way that ACEs can register themselves at the REDS registry and can find other ACEs who are already registered. In order to register the ACE world at the REDS registry, the `reds.registry` property in the settings.properties configuration file has to be set to the REDS registry's URL (see chapter 4.3.1).

The appropriate network interface has to be defined within the settings.properties file in case multiple network interfaces are available on the device. Using the `hostname` parameter in the settings.properties file, the default network interface where the ACE communication will take place can be specified.

Wired network communication is much more reliable than the wireless one. In the case of wireless communication, signal interference and other problems of the wireless communication can lead to large packet losses, which require retransmissions. For this reason a socket establishment process might take longer in this case than when using wired type of communication. Developers can adjust `diet.socket.timeout`, `diet.mirror.setup.timeout` and `diet.mirror.channel.timeout` in the settings.properties file to adjust the distributed application to the communication channel requirements.

As soon as all local ACE worlds are started and connected to the registry, the participating ACEs will find each other by exchanging GN-GA messages and contracts will be established in the usual way. Whether an ACE application is local or distributed, does not influence the general work flow at all. Each participating ACE can operate in the same way as in a local application.

[Benk08] describes the advertisement example application executed in a distributed manner. Figure 10 illustrates how the participating ACEs are dispersed among three different devices.

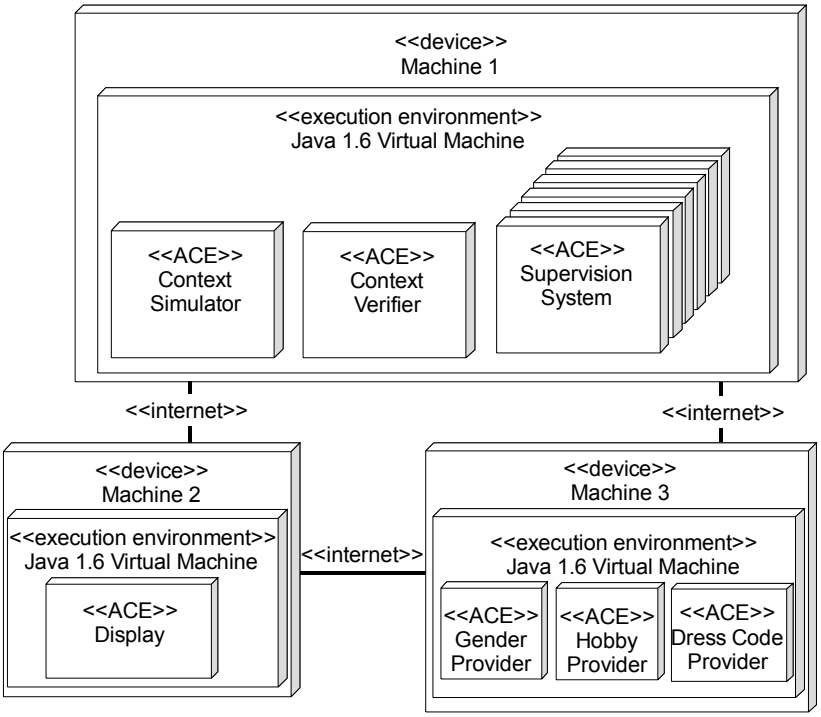

Figure 10: Overview of the distributed advertisement example application

5 Toolkit Evaluation and Results

In this section we present some experiments that we performed to analyse the ACE model. In a first set of experiments, we conducted some measurements on the ACE architecture to assess its performance from a distributed system perspective. In another set of experiments, we tried to measure the efficacy of different approaches in the design of the ACEs' program (i.e., their plan). The results of these experiments have also been reported in a journal paper submitted to IEEE Transaction on Computers ([Betal08]).

5.1 Distributed System Perspective

To evaluate the effectiveness of the ACE architecture, it is important to analyse it from a distributed system perspective. In particular, we conducted a set of experiments aiming at measuring: *(i)* threads, *(ii)* memory usage, and *(iii)* communication delay of ACE applications.

5.1.1 Thread Analysis

In this set of experiments we analysed with a profiling tool the thread consumption of the pervasive advertisement application described in the previous section. Figure 11 reports thread use over time of the different organs composing the ACEs in the application. Looking at the graph it is possible to see that, after a bootstrap phase, the number of threads involved in the application remains stable for most of the organs. The only elements deviating from this pattern are timers associated to various tasks in the application. These elements add up in the wait state, until the associated time periods expire freeing the thread. This creates the saw-tooth plot in Figure 11.

Figure 12 shows how the number of ACEs influences the overall number of threads. For this experiment, an application was developed that starts with one ACE and subsequently adds more of them until a number of 100 ACEs is reached. The ACEs we used in that experiment are simple and do not need any timers. In consequence, the curve does not exhibit a saw-tooth like behaviour. What can be seen is that the number of threads grows linearly with the number of ACEs leading to the conclusion that, in terms of threads, ACE based applications scale efficiently.

The overall flat behaviour of the threads being used by our application and the linear growth when the number of ACEs increases show the stability of the ACE architecture which supports the use of ACEs in large-scale applications.

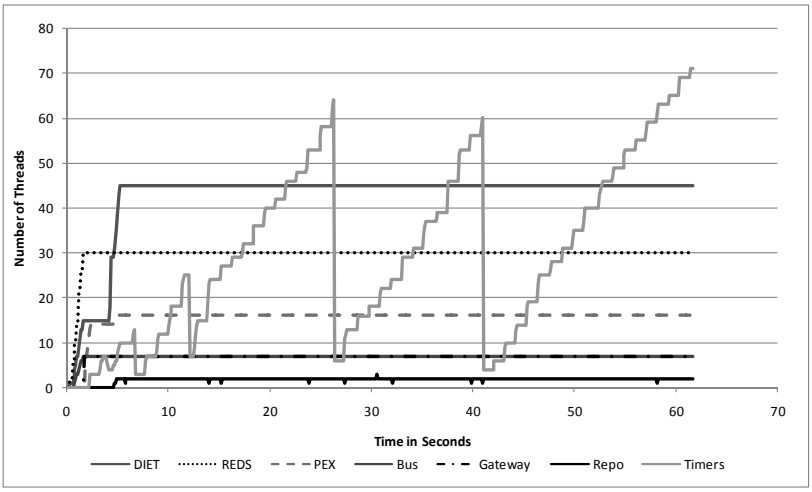

Figure 11: Thread classification

From a complementary perspective, Figure 13 illustrates how the CPU time is divided among the kinds of threads in the system. Looking at the graph it is rather easy to see that the DIET infrastructure, on which the ACE architecture is executing, accounts for almost one third of the total CPU time. Since the application being considered is based on a number of ACEs interacting in a communication-intensive service, the combination of the Gateway organs and REDS infrastructure accounts for another third of the total CPU time. The actual execution of the plan (PEX – Plan Executor) and of internal communication (Bus) accounts for the remaining third of the operations. The fair balance of operation within the ACE architecture hints at a correct division of responsibilities among ACE organs.

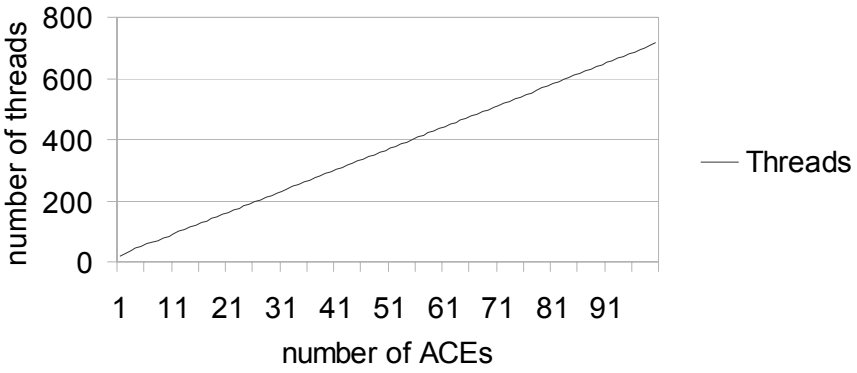

Figure 12: Number of threads over number of ACEs

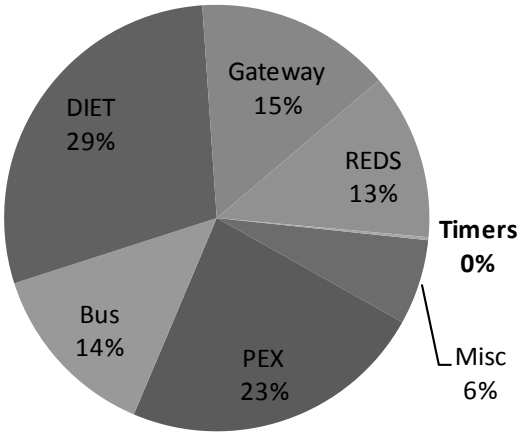

Figure 13: CPU usage per thread class

5.1.2 Memory Analysis

In this experiment we analysed the memory consumption of ACE based applications. In particular, we set up an experiment in which ACEs are sequentially created, starting from 1 and counting up to 100. We measured the memory that is allocated to the application. Different garbage collectors have been utilised and the test results have been compared accordingly. Figure 14 depicts the memory allocation patterns for the garbage collectors *ParallelOldGC*, *ParallelGC* [5] , *SerialGC* [6] (see [SunGC1] and [SunGC2]). Figure 14 represents the memory allocation pattern for *IncGC*[7] (see [SunGC1] and [SunGC2]). While the former three collectors exhibit a similar behaviour – suggesting a constant amount of allocated memory – the latter one, IncGC, shows a memory consumption curve that linearly increases with the amount of ACEs. The oscillatory character of all garbage

[5] ParallelOldGC and ParallelGC are intended to be used on parallel machines, so that they can occupy one processor while application execution continues on another one.
[6] SerialGC stops application execution while cleaning up.
[7] IncGC stops only for a short time and cleans up only partially in order to not block application execution too long.

collectors is due to the fact that they are not permanently cleaning up the memory but are only running periodically with a certain time delay between each execution (see Figure 15).

In conclusion of our experiment we can say that the memory consumption of ACE based applications in detail depends on the garbage collector used. Nevertheless, the results show that the amount of allocated memory grows linearly with the number of ACEs within the application. This also supports the scalability of the ACE platform for large scale applications.

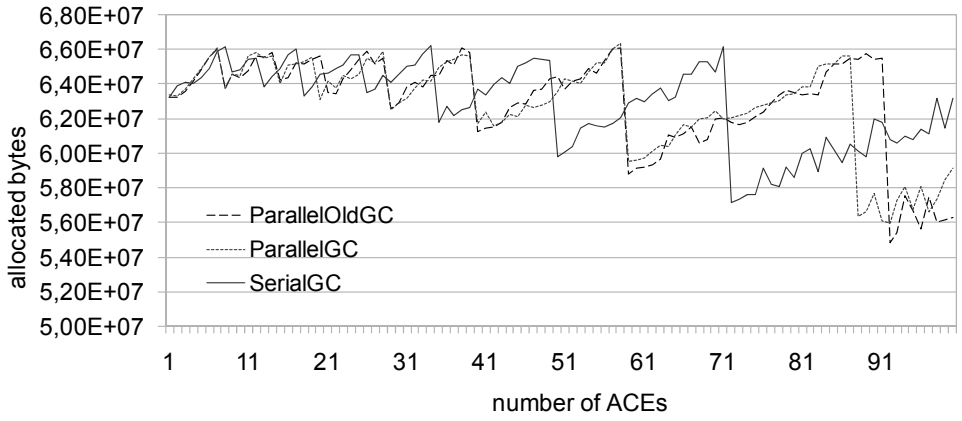

Figure 14: Memory usage with different garbage collectors: ParallelOldGC, ParallelGC, and SerialGC

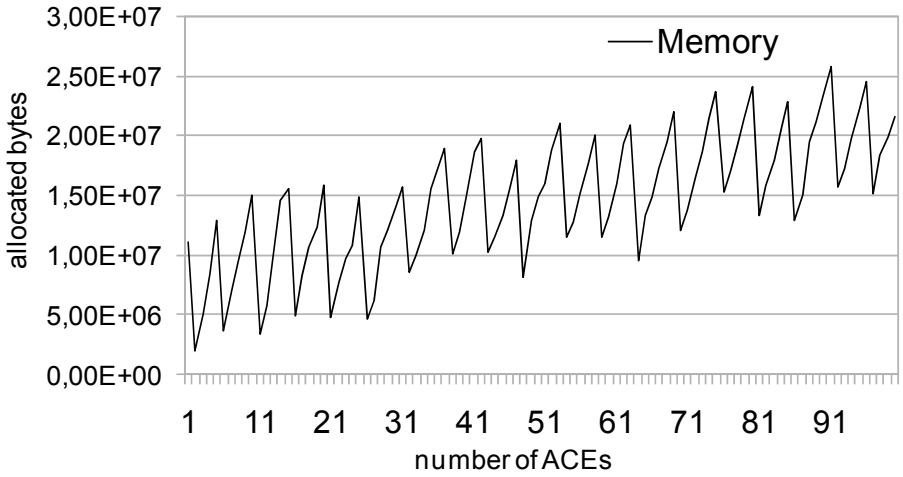

Figure 15: Memory usage with Incremental Garbage Collector IncGC

5.1.3 Communication Delay

In another set of experiments we measured the delay in terms of communication in a distributed test-bed. Communication capabilities have been tested experimentally through executions on a real test-bed composed of machines distributed over a LAN. These machines are configured in a way that ACE communication was the only traffic among them at the time the testing took place.

In the initial setting, two ACEs find each other and start communicating through a classic ping protocol. To this extent, a ping packet is sent every 500ms, with the source ACE

recording the transmission time. The packet is received and acknowledged at destination, so that this latter packet can be received at the original ping source and the reception time can be recorded. Each two minutes, a new ACE joins the system. This is quickly found by other ACEs which are already present in the ACE world, and suddenly involved in the ping process.

The communication delay is simply calculated as the difference between the transmission time of the ping packet and the reception time of the acknowledgement packet. It is worth noting that timings are recorded at application level. Thus, the time recorded includes the actual network delay and the time needed for the packet to be processed (at both ping and acknowledgement destinations). Therefore, considering that the communication takes place in a LAN environment of dedicated machines, communication delays can be expected to impact the recorded timings to a lesser extent. As a consequence, delay variations can be mainly imputed to variations in the packet processing times. The linear increase of delays with the number of ACEs shows a linear scalability in ACE distributed applications (see Figure 16).

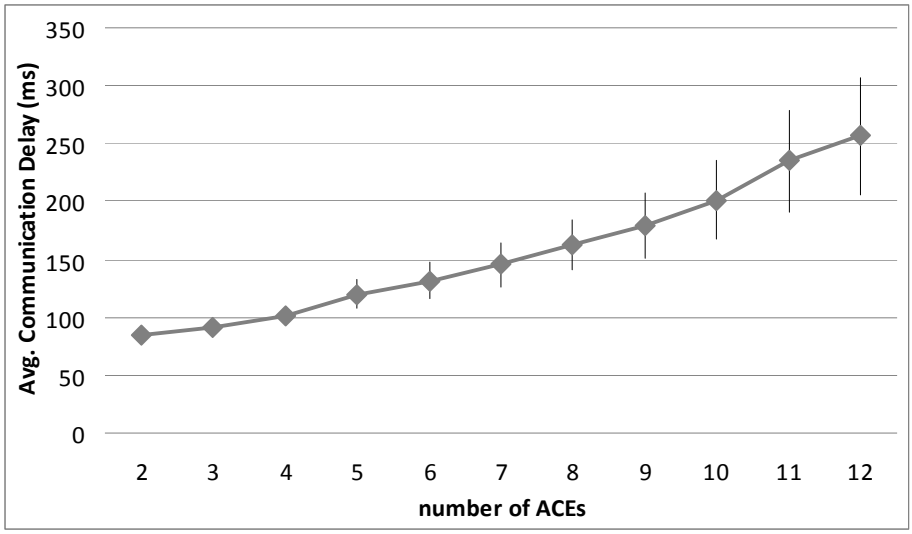

Figure 16: The average communication delay increases linearly with the number of ACEs in the system

In another experiment we measured how the response time for a simple service in a local test-bed is affected by the parallel execution of other independent services. In this setting, there are *n* couples of ACEs: *n* client ACEs request a service *i* different for each of them, provided by just one of the *n* server ACEs. Every client ACE records a timestamp before the service invocation (when the contract with the provider has been acknowledged) and another one when the response to the service request has been provided by the server. The round-trip time is calculated as the difference between these timestamps. While the average response time increases quite linearly with the number of couples of ACEs, that is, with the number of different services provided by the platform, bigger oscillations can be seen in the maximum response times, due for example to temporary congestion conditions. In a situation like this, it is possible to look to fine tune the system based on the maximum response times that can be tolerated by the clients and the throughput that should be provided by the servers (see Figure 17).

In conclusion of these experiments, it is possible to see that the ACE architecture is scalable along many dimensions (memory, threads, communication delay) in a distributed setting. This supports the applicability of the ACE model in large-scale autonomic communication scenarios.

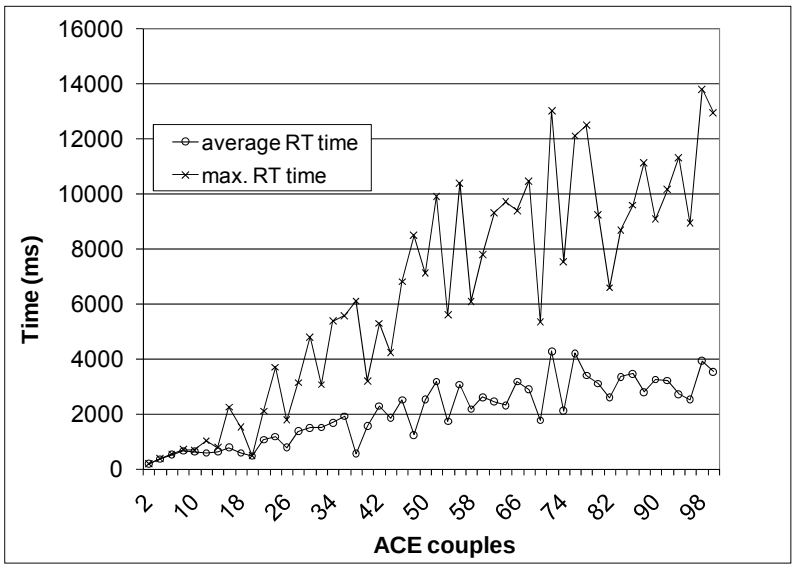

Figure 17: Average and maximum execution (round trip) time versus number of ACEs in the system

5.2 Comparison of Design Approaches in Plan Development

In a second area of experiments, we measured the efficacy of different approaches in the design of the autonomic components program. In particular, we wanted to test the trade-offs implied in different approaches of ACE plan design.

To this end, we created a graph visualising the memory consumption and planning overhead of different wordings of the same plan. The three design approaches (i.e., wordings) are:

- *One complex plan*. One monolithic plan encompassing all the ACE activities (i.e., one big finite state automaton).
- *Sequential decoupling with plan replacement*. Sequential decoupling of a plan means to divide it into successive sections, then to extend the sections with a small number of extra states and transitions, so that when a section reaches its final state it requests the announcement of the next section from the Facilitator, and stops itself. This can also be understood as a simple form of lazy planning.
- *Parallel/child plans*. This design approach extends sequential decoupling allowing the plan to be split in a hierarchy of child-plans that can possibly run concurrently.
- *On-the-fly plan modification*. This last design approach is similar to self-modifying code. Using the advanced features of the Facilitator organ, it is possible to rewrite dynamically the plan the ACE is executing.

The memory consumption is measured in terms of the number of states in the Executor. The planning overhead is a complexity measure proportional to the number of states and transitions in the model. Both factors are shown along a logical time axis where a tick

means the arrival of the next concerning input event. The measurements were made using ACE self models created for the pervasive advertisement application described in the previous section. In all the experiments, the latter three approaches (sequential decoupling, parallel plans, and on-the-fly plan modifications) are compared with the naïve "one-complex-plan" approach.

Using the "sequential decoupling" approach, it is possible to reduce the average number of states in the Executor without any additional considerable overhead in the overall number of states or in the planning. Figure 18 illustrates the case in which a plan is decoupled into *N*=5 successive sections (all with different numbers of states). The decoupled version utilises *N* times less memory on average (see Figure 18 left). Moreover, the execution of the decoupled sequence takes somewhat longer than the execution of a single plan because of the plan announcement and removal steps (see Figure 18 right). However, the concerning added overhead is linear with the number of states.

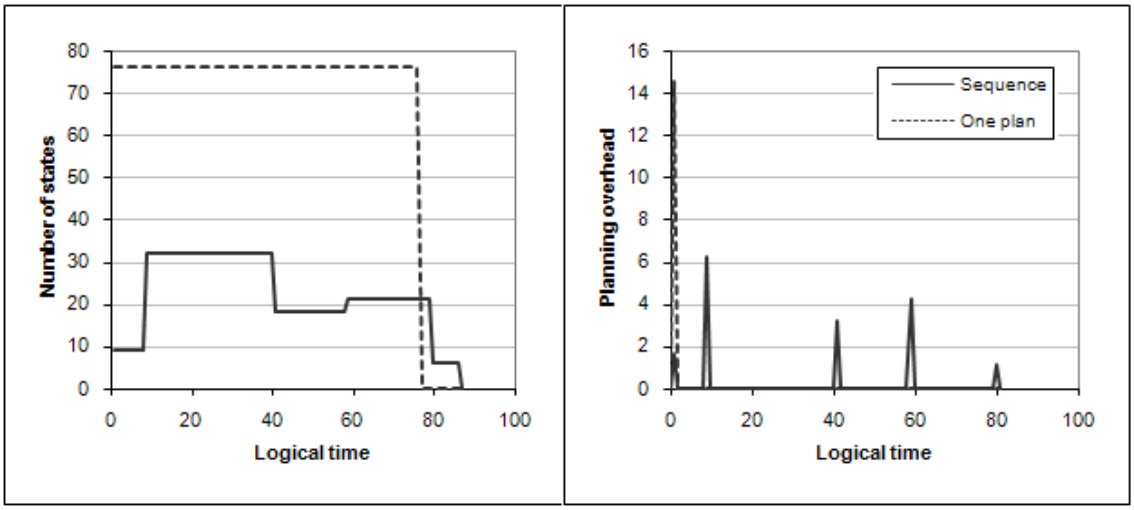

Figure 18: Sequential decoupling of a plan

Plans executed in parallel offer a polynomial decrement both in the average number of states and in planning time, compared to a single complex plan. This is because a single complex plan is theoretically the superset of the sub-plans (a state of the super-plan is the cross product of the sub-plan states). Figure 19 shows the case in which a first plan of 12 states starts the concurrent execution of another plan of 25 states (this happens after 8 simulation ticks). For comparison we plotted the un-optimised (i.e., cross product) and optimised (i.e., plan designed by hand) single plans equivalent. The overall planning time for parallel plans is considerably smaller than for complex plans, mostly because of the difference in the number of states and transitions.

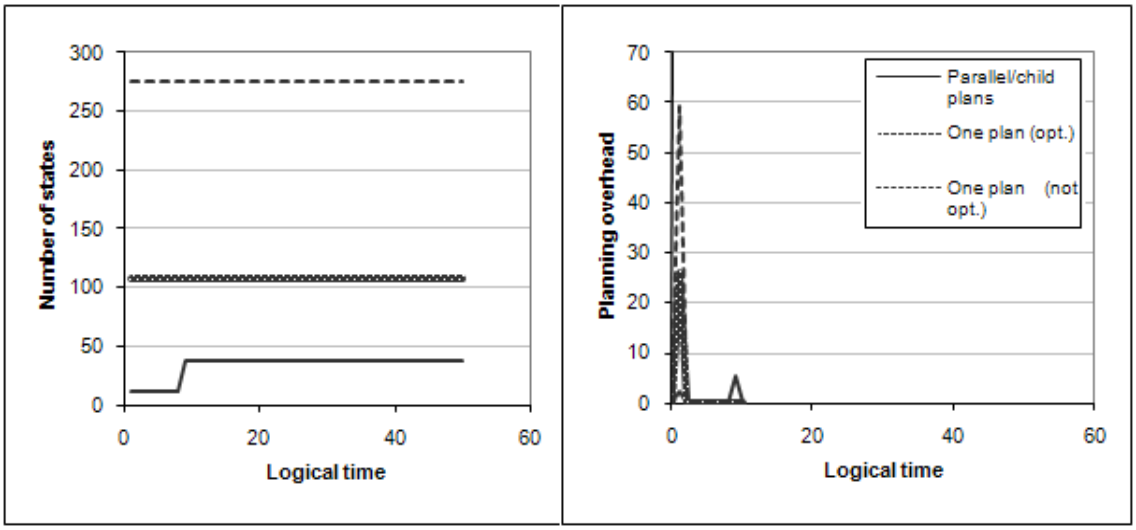

Figure 19: Parallel plans

On-the-fly plan-modification is advantageous when we need fast intervention and cannot afford the overhead of the complete plan creation and announcement process during the operation of the ACE. Figure 20 shows a case in which a plan (11+13 states) either gets modified after state 11, or gets replaced by another plan (please note that in the case of replacement, the plan changes by one transition only). Looking at the picture, it is possible to see that the on-the-fly modification approach involves a higher average number of states (it must include all possible states from the beginning) but has just a minimal planning overhead at operation time. The plan replacement option includes a small number of additional states (for plan replacement), and results in some plan creation overhead, but results as well in a lower average number of states.

In conclusion, it is possible to see that the ACE architecture effectively supports several forms of modularity in design, making it possible to separate and reuse sub-plans depending on different conditions. Not only does such a modular approach lead to cleaner and more engineered software components, but it also exhibits better performance in terms of memory consumption and planning overhead.

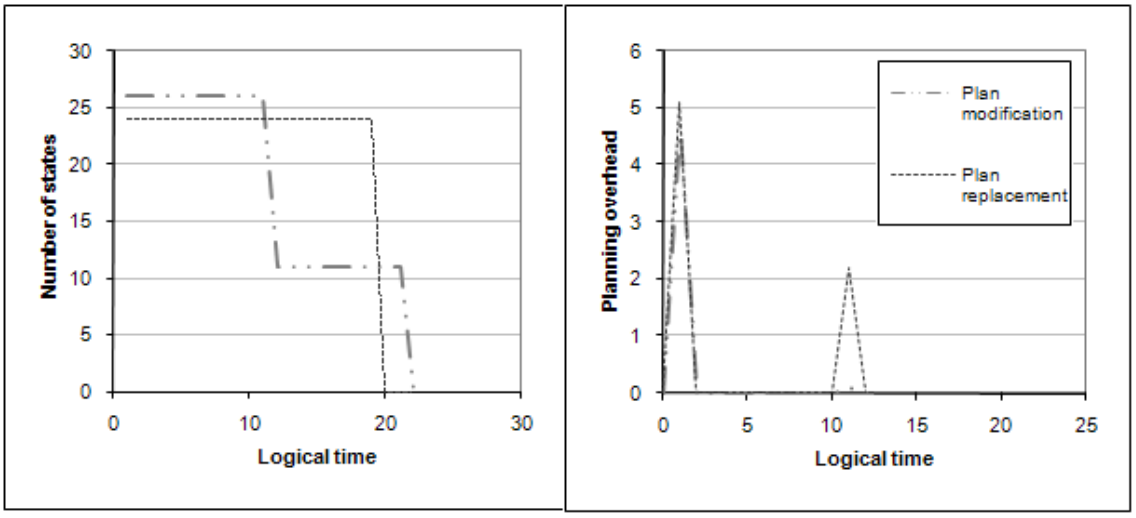

Figure 20: On-the-fly plan modification

6 CASCADAS Integration Activities

6.1 General Overview

The CASCADAS Toolkit has been designed and developed in Java as a run-time environment capable of supporting ACEs in all their features, i.e. lifecycle management, interaction with other ACEs and integration of legacy code etc.

The CASCADAS Toolkit, through successive releases, will progressively incorporate concepts and software libraries dealing with advanced solutions investigated in all WPs, for example for self-supervision (WP2), self-aggregation (WP3), security (WP4), distributed access to knowledge networks (WP5), etc. WP6 acted, and is acting, as a collector for making experiments with the Toolkit in the context of the use-case scenario "Behavioural Pervasive Advertisement". Specifically the Toolkit has been successfully used to build a fully working prototype system to suit the needs of a future use-case scenario, particularly interesting from an industrial perspective.

This section is reporting a synthesis of the status of WPs integration activities at month 30. It describes the overall CASCADAS project integration activities from the viewpoint of an individual work package. Specifically,Table 1 Table 3 is highlighting the main cross-WP questions currently under consideration.

	WP1	WP2	WP3	WP4	WP5	WP6
WP1		1) interrogation and control functions and failure modes 2) ACE Toolkit embedded Supervision	1) implementation of self-* algorithms in the ACE self model 2) Enhancements of the ACE architecture to support self-organisation.	1) definition of the security architecture and ACEs functionalities following the ACE model 2) implementation of security ACEs as specific service functionalities 3) use of crypto ACEs for the ACE components	1) Providing ACEs active access to knowledge network and its resources	1) PBA use-case (including portability on mobile N800) 2) evaluation of Toolkit in the use-case
WP2			1) self-organisation to place supervisors and build up a supervision pervasion (reverse clustering) 2) Self-* algorithms for ACE Toolkit embedded Supervision	1) use monitoring and data correlation infrastructure for trust management 2) use of reputation to detect faulty alarms and discriminate malicious components from faulty ones	1) validation of liveliness of knowledge atoms 2) maintenance of backup knowledge containers	1) PBA use-case: Seller and AuctionCentre are supervised ACEs
WP3				1) study rational nodes and the neighbour selection strategies which can impact the self-aggregation algorithms.	1) Possible use of WP3 self-aggregation algorithms for managing large amounts of correlated data	1) Use-case with TinyACEs showing self-*

WP4					1) possible use of crypto ACEs for the knowledge network.	1) Integration of ACE supporting encryption services in application scenario 2) Evaluation of the benefits introduced by migrating ACEs as a mitigation mechanism for denial-of-service attacks
WP5						1) Knowledge networks in use-case PBA

Table 3: Status of WPs integration activities (M30) in a nutshell

6.2 Integration Activities Status

In the first six months of 2008, a very close WPs interaction under tight coordination has been carried out. By means of scheduled weekly phone conferences, face-to-face meetings, and informal communication, WPs continuously discussed with each other to ensure that requirements have been appropriately captured in the ACE Toolkit implementation, to achieve a common understanding of the Toolkit's design principles, and to share know-how in programming with the Toolkit. In the following, detailed integration activities per work package will be described.

6.2.1 WP1 Integration Activities

From the point of view of WP1, the integration effort has been mainly focused on providing additional ACE functionalities (or enhancing the existing ones) as requested by other WPs. The integration work regarding new feature requirements implementation has been mainly handled in close cooperation with WP2, WP3, WP4 and WP5.

Integration activities between WP1 and WP2 focused mainly on providing additional supervision specific capabilities to the ACE. On one side, WP1 has integrated into the ACE architecture the "Supervision Organ" (cf. [D1.3]) which provides the supervision system the capabilities of monitoring the ACE operation and actively supervising its further activities from outside. On the other side, WP1 has enhanced the overall ACE capabilities with regard to interrogation and active control functionalities.

Integration activities between WP1 and WP3 focused on enhancing the ACE architecture in order to provide capabilities for implementing WP3s' self-* algorithms in the ACEs. In particular, the ACE architecture has been modified in the way that creating overlay networks over a group of ACEs is supported by every ACE.

As a result of a close cooperation between WP1 and WP4, ACEs have been equipped with the capability to handle different confidentiality levels. In particular an ACE can decide when to encrypt its messages before sending them to the other party. In the same way, the receiving ACE can detect if the message is encrypted or not, and will request the local confidentiality ACE to decrypt it in case the message is encrypted.

WP1-WP5 integration work focused in the first instance on providing ACEs with WP5 required functionalities. Knowledge Networks (KNs) which are developed within WP5 are

based on ACEs and therefore it was required to provide all ACE functionalities and features which are needed for implementing KNs. On the other side our cooperation focused on providing ACEs with capabilities to query knowledge networks and receive desired information from them.

Integration with WP6 (which focuses on an ACE application scenario and its evaluation) has been carried out in the way that WP1 provided support to the implementation of the demonstration scenario (developed by WP6) and WP6 provided feedbacks to WP1 on how to improve the ACE capabilities and the list of potential bugs (see ACE bug list in bugzilla) which have been found during the testing and evaluation phase. Moreover, in the context of WP1-WP6 integration activities, WP1 has successfully ported the ACE Toolkit to the Nokia N800 mobile device and provided it to WP6 for ACE Toolkit evaluation purposes.

6.2.2 WP2 Integration Activities

WP2 cooperation activities with other WPs are graphically represented in Figure 21.

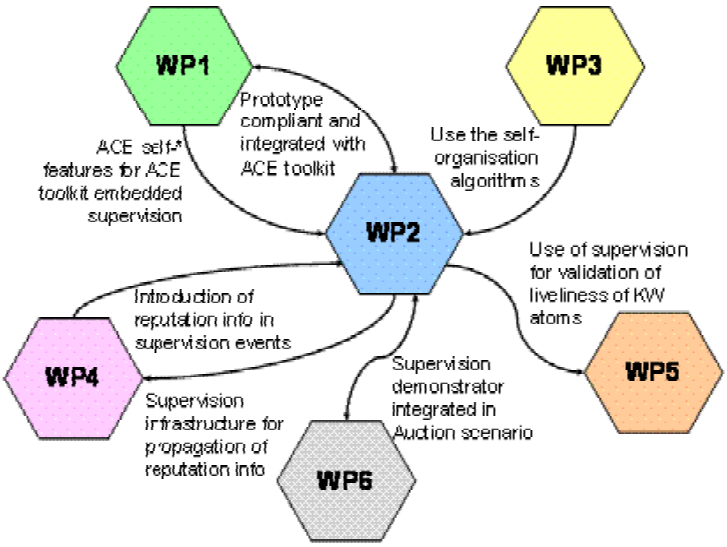

Figure 21: WP2 cooperation activities with other WPs

Some activities are investigating how WP2 solutions could be applied in other contexts of CASCADAS:

- Work with WP4 is investigating on how to apply the supervision infrastructure to circulate reputation information in the bidder scenario;
- Work with WP5 is analysing the use of supervision solutions for the validation of liveliness of knowledge atoms (e.g., periodic "test queries", de-registration if no answer is received in time);

Other cooperation activities aim at improving the adoption in the supervision mechanisms of solutions developed in other WPs:

- Investigation on how to use the self-organisation algorithms, developed by WP3, to place supervisors and to build up a supervision pervasion, and to set-up the overlay for exchange of supervision information in the ACE Toolkit embedded supervision;

- Investigation on how to apply self-* features of the WP1 ACE Toolkit for implementing local supervision logic for the ACE Toolkit embedded supervision;
- Work with WP4 for the introduction of reputation information in the events sent by supervised ACEs to their supervisors;

Finally, other activities have the objectives to enforce the integration of supervision solutions in the CASCADAS Toolkit and demonstration scenarios:

- Joint work with WP1 for the compliance and the integration with the ACE Toolkit of the supervision prototype;
- Work with WP6 for the extension of the Auction scenario to demonstrate capabilities of the supervision prototype.

6.2.3 WP3 Integration Activities

It has already been mentioned that integration activities between WP1 and WP3 have been focused on enhancing the ACE architecture in order to provide capabilities for implementing WP3s' self-* algorithms in the ACEs. In particular the ACE architecture has been modified in such a way that creating overlay networks over a group of ACEs is supported by every ACE. Also the WP2-WP3 investigations on how to use the self-organisation algorithms, developed by WP3, to place supervisors and to build up a supervision pervasion, and to set-up the overlay for exchange of supervision information in the ACE Toolkit embedded supervision have been mentioned. Work has begun at the WP2-WP3 interface looking at general ways in which the WP3 simulation work, which has focused largely on decisions (or actions) relating to movement around networks or autonomous individual state changes, can be re-interpreted as decisions or actions relating to alternative supervision choices. An initial description of such a scenario in the collective decision-making arena has been included in the latest WP3 deliverable ([D3.5]).

By month 33 (September 2008) the WP3 kit for incorporating the self-aggregation algorithms into the ACE Toolkit will be released on SourceForge. This is part of Deliverable D 3.6. This kit will be composed of the following elements:

- A self-model controlling the start-up and termination of the self-aggregation algorithms.
- A set of functionalities that implement the execution of the algorithms.
- A new version of the gateway able to correctly interpret the events that will be published and received during the execution of the self-aggregation algorithms.

6.2.4 WP4 Integration Activities

WP4 has exploited the work carried out in WP1 to implement examples for the integration of security functionalities in the ACE Toolkit. To achieve integration, WP4 has defined security ACEs in accordance with the security architecture defined in [D4.1], and integrated security as specific service implemented in the functionality repository. This work is fully integrated in the activities of WP1, as the security ACE can be used/aggregated when the type of communication requires a higher security level.

As mentioned in section 6.2.2, WP4 has established close cooperation with WP2 to understand the dynamic of the supervision system so that the monitoring capabilities of this system can be exploited to collect information on the behaviour of the ACEs. On the way

around, the supervision system is in the position to take advantage of the application of the reputation management system to filter out false information and detect false alarms raised by malicious ACEs.

Still in the same context, the work on the application of reputation and on rational nodes have some aspects that are important for the aggregation algorithms defined in WP3 as a tool to modify the neighbour selection strategy. This is because the presence of rational nodes can impact the dynamics of the system and the concept of reputation can be used to rule out those nodes that do not follow the protocol specifications.

Joint activities have been defined with WP6 with regard to two specific aspects. The first consists of the definition of security principles and solutions for the distributed auction scenario which is part of the implementation of the CASCADAS application. The second aspect concerns the demonstration of security aspects in a simple application scenario. In the former case, specific solutions to prevent and limit denial of service attacks have been tested on the pervasive advertisement scenario, while, in the latter case, the implementation of confidentiality ACEs has required close cooperation between the two WPs so that the code might be used if needed to protect sensitive information. In this last direction, the cooperation is continuing and it is targeting the implementation of authentication mechanisms.

6.2.5 WP5 Integration Activities

WP5 has interacted very strongly with WP1, WP2, and WP6, and it has also interacted with WP3 and WP4.

With regard to WP1, WP5 has continued expressing to WP1 their requirements for enabling ACEs to be more and more effectively exploited for knowledge networks, which ended up in integrating parameterisation into ACEs (a feature that was found out to be necessary to facilitate knowledge querying and knowledge self-organisation).

With regard to WP2, as already stated above, integration activities have concerned the possibility of using pervasive supervision to check liveliness of knowledge atoms and, therefore, to increase the reliability of knowledge networks.

With regard to WP6, WP5 has continued collaborating with the implementation of the pervasive advertisement case study, i.e., on supporting the integration and the proper use of knowledge networks within it.

As in 2007, interactions with WP3 and WP4 have been mostly of an informative nature, aimed at ensuring the full integrity of the self-organisation algorithms of WP3 and of the security solutions of WP4 with the knowledge networks concepts and implementations.

6.2.6 WP6 Integration Activities

WP6 Task 6.2 (Socio-economic Impacts) is mainly in charge of making techno-socio-economic evaluations of the potential impact of CASCADAS technologies and solutions for the evolution of the Service Frameworks of future Telecommunications-ICT and Internet. During the first six months of 2008, WP6 T6.2, in strict coordination with technical WPs, has worked on identifying qualitative and quantitative indicators for CASCADAS technologies and solutions and ways to evaluated them (cf. [D6.8]).

In general, WP6-WPs integration has been carried out in the way that WPs provided support to the implementation of the demonstration scenario (developed by WP6) and WP6 provided feedbacks to WPs on how to improve the ACE capabilities and the list of potential

bugs (see ACE bug list in bugzilla) which have been found during the testing and evaluation phase (cf. [D8.5]).

Specifically, in the context of WP1-WP6 integration activities, WP1 has successfully ported the ACE Toolkit to Nokia N800 mobile device and provided it to WP6 for ACE Toolkit evaluation purposes. Moreover, in the same direction, WP1-WP3 are jointly developing the TinyACE (running self-* algorithms) for integrating in the WP6 demo other mobile devices. In the context of WP1-WP2-WP6 integration, supervision solutions in the CASCADAS Toolkit and demonstration scenarios have been enforced.

As mentioned, WP4 and WP6 are carrying out integration activities concerning: 1) definition of security principles and solutions for the distributed auction scenario which is part of the implementation of the CASCADAS application; 2) demonstration of security aspects in a simple application scenario.

WP5 and WP6 are cooperating for the implementation of the pervasive advertisement case study, i.e., on supporting the integration and the proper use of knowledge networks within it.

7 Conclusion and Outlook

Implementation and research activities on the ACE component model have proceeded in a smooth way during the first half of the third year of the CASCADAS project. There have been no significant problems or deviations from the planned objectives. A high degree of interactions took place, not only among the WP1 partners but also among WP1 and other CASCADAS work packages.

The activities have led to the achievement of several important results, as reported in this document, and in particular:

- Improvements of the overall ACE Toolkit stability and implementation of additional ACE features.
- ACE cloning functionality has been developed successfully.
- A full version of the ACE Toolkit has been successfully ported to the Nokia N800 mobile device and first results regarding the ACE Toolkit on Google Android platform are available.
- Initial steps towards the Tiny ACE approach have been made and a first implementation is available.
- The ACE Toolkit has been evaluated with regard to its resource requirements and processing power consumption. The underlying ACE architecture has been improved based on the evaluation results.
- The open source ACE Toolkit has been released on SourceForge.

The innovative nature of the above findings has led to the production of several scientific papers, which include an overview paper on "Autonomic Communication Elements: Design Principles, Architecture and Implementation" which has been submitted to the IEEE Transactions on Computers journal.

The ACE Toolkit has been successfully integrated with the concepts and tools provided by other work packages. The example application included in the Toolkit shows some of the Toolkit's potential in a practical manner for the project's pervasive advertisement scenario.

Bringing Autonomic Services to Life

In summary, the current release of the CASCADAS Toolkit is in line with our plans and expectations. For the future we plan to continue to enhance the ACE Toolkit while fixing potential bugs and developing additional features like for example advanced contracting and ACE migration functionality. Our work regarding mobile device support will continue in the future. We plan to extend the number of supported mobile devices to the Google Android platform and if time and resources allow, to implement a lightweight version of the autonomic communication element called Tiny ACE. In cooperation with other work packages we will continue our project wide integration work and will provide the required ACE features and capabilities.