UNIVERSIDAD POLITÉCNICA DE VALENCIA

Escuela Técnica Superior de Ingeniería Informática

PROYECTO DE FIN DE CARRERA

Distributed Goal Oriented Computing

Autor: Javier Palanca Cámara

Dirigido por: Vicente Julián Inglada

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46020 Valencia, Spain

Contents

List of Figures

List of Tables

Abstract

For current computing frameworks, the ability to dynamically use the resources that are allocated in the network has become a key success factor. As long as the size of the network increases, it is more difficult to find how to solve the problems that the users are presenting. Users usually do know *what* they want to do, but they don't know *how* to do it. If the user knows its goals it could be easier to help him with a different approach. In this work we present a new computing paradigm based on goals. This paradigm is called Distributed Goal-Oriented Computing paradigm. To implement this paradigm an execution framework for a Goal-oriented Operating System has been designed. In this paradigm users express their goals and the OS is in charge of helping the achievement of these goals by means of a service-oriented approach.

1

Introduction

The amount of developed software and its complexity has currently increased so much that it has lead to discover that traditional paradigms of software development are not enough to create complex software. That is why there is a constant work on new paradigms, to improve the level of abstraction needed to develop increasingly complex applications. Among these paradigms, we can highlight the Service-Oriented Computing paradigm and Multi-Agent Systems.

Service-Oriented Computing (SOC) is a paradigm where the fundamental component for developing applications is the service. Using single services or service compositions it is possible to achieve solutions to problems in a decentralized manner with a high degree of adaptability. This paradigm, coupled with the cloud-computing one, is becoming very important at the present moment because both paradigms allow us to develop applications based on platform-agnostic, distributed and low-cost computational elements. The use of SOC in multi-agent systems is endorsed by the proposal of achieving the agent goals by means of the invocation and composition of a set of services that are available within the multi-agent system.

Dickinson and Wooldridge discuss at [1] different ways to consider the relationship between multi-agent systems and service architectures. As it is summarized in that work, some authors propose that there is no conceptual distinction between agents and services: *both are active building blocks in a loosely coupled architecture* [2]. Another approach considers a bi-directional integration where agents and services interoperate by communicating one to each other [3]. Finally, a third approach considers that agents are who invoke services [4]. In this proposal, agents mediate between services and users.

Since agents are intelligent entities and have social capabilities, they fit properly in a service-

based framework[5] where the goal-oriented computing approach is used. This approach is based on finding solutions to problems through composition and execution of various services offered by different agents.

This goal-oriented computing paradigm suggests that agents provide services in a ubiquitous environment and users only need to express their goals. Thereby users can reach a solution by finding a plan which achieves the selected goal with very limited and simplified user interaction.

This functionality should be provided to agents through a specific framework that supports service composition and their subsequent execution. Agents are providers and consumers of services in this framework, where agents use their social capabilities to find a way to fulfill their own goals. These capabilities should be provided to agents through a specific architecture that supports service composition and their subsequent execution. This framework is presented in this work as an execution module for a Goal-oriented Operating System. Current operating systems (OS) are based on abstractions that have not evolved too much since their first designs. However, the evolution of software engineering poses the possibility of addressing the OS design from other points of view. The Distributed Goal-Oriented Computing paradigm offers new ideas for the development of more intelligent and effective OS's, which would benefit the end-user due to the advantages of both technologies.

In this work the Distributed Goal-Oriented Computing paradigm is presented. It is also presented an execution module for a Goal-oriented Operating System which gives support to this paradigm following the requirements defined in this work. Some of this requirements comprise how to define the properties of a goal and the parameters that define how good is a plan. Some of the parameters that involve the creation and selection of a plan are *time* and *trust*.

This work is structured as follows: Section 2 presents a related work about Operating System designs and trends. Section 3 presents the model and the architecture of the Distributed Goal-Oriented paradigm used in this work. In Section 4 we talk about the operating system deliberation engine and the various components that comprise this engine. Section 5 presents the execution module that interacts with the deliberation engine to develop the presented paradigm. This module is the Runtime Engine. Section 7 presents a series of experiments to show the functionality of this Operating System. Finally, Section 8 presents the conclusions of this work.

2

Related Work

Operating Systems research is always trying to improve security, efficiency and reliability of Operating Systems. This is one of the great challenges of the current OS that remains to be overcome. Several studies have focused on improving certain OS aspects as data access or the input/output (I/O) abstractions, leading to propose new abstractions in this field (file, object, socket, ...)[6, 7, 8]. However, no significant progress has been made in implementing OS execution models.

2.1 An updated OS classification

Nowadays, *any* operating system has multiuser, interactive and multiprocessor skills due to the evolution of computers, which made indispensable that all the OS endure it. The interesting point is the vision of its purpose, due to the important differences that lie on an operating system's design depending on what it was created for. It depends on how the device is going to be used for or its specific functionality. The following is an updated classification that includes the different OS differentiated by their architecture or purpose. This is not an exclusive classification, since it represents the different approaches that can be taken during the design of an OS. Several options can be taken simultaneously. This makes possible the creation of, for example, a multiprocessor, extensible and general purpose operating system.

- **Mainframe operating systems:** These are systems oriented to large computers where the computing and the input/output (E/S) power are important.
- **Server operating systems:** they are oriented to bring services across the net as well as to process efficiently a large amount of requests per second.
- **General purpose operating systems:** they were created for mass consumption. Their only goal is to bring, with a simple and friendly interface, the most common tools for the daily use of a personal computer.
- **Extensible operating systems:** Extensible OS give support to dynamic loading of new features in the system as required for its purpose. These new modules are loaded to *extend* the OS according to the needs of each moment.
- **Multiprocessor operating systems:** A special OS is needed to handle and share the jobs in computers with more than one CPU.
- **Parallel operating systems:** These are an extension to multiprocessor systems where the need to run different applications on multiple processors extends to a computer network or cluster.
- **Distributed operating systems:** Nowadays, there is a trend towards distribution of the different services that an OS offers among a number of computers making use of the network.
- **Grid operating systems:** These are an extension of distributed systems where there is access to geographically distributed resources by all network nodes in a heterogeneous grid.
- **Real Time operating systems:** These are OS for some very specific applications where not only the result of an operation is provided, but also the precise moment is important.
- **Embedded operating systems:** These systems run on control devices which are not generally though as real computers and which do not accept user-installed systems. Typical examples are microwave ovens, washing machines, televisions, cars, etc.

On this list of operating systems we can distinguish several groups that will outline the current trends in OS development. Thus, operating systems are characterized as *service-oriented* (such as servers), those aimed at enhancing *performance or availability* of the network (such as distributed systems, grid, parallel, etc.), those systems oriented to a *particular* purpose (such as real-time or embedded ones). Embedded systems become day to day more important due to the increasingly massive introduction of mobile devices. Finally, the ever-present *general purpose* operating systems, which are still very important given the high penetration of personal computers in homes.

One reason why no significant progress has been made in the OS execution model abstractions, such as the process, is that these abstractions are closely tied to current hardware. Processors are designed to work optimally with processes. Thereby, when adding improvements to the OS execution model, as well as defining new execution abstractions (as proposed in this work) would be interesting to start thinking about adapting the hardware to such abstractions.

2.2 Three Modern Operating Systems

In this section three modern Operating Systems are presented and analyzed in order to study the new trends in OS design and implementation. The OS analyzed are Singularity (an experimental OS where have been tested new techniques like code verification, contracts or modern VM-based languages), MINIX 3 (an evolution of the classic MINIX OS where the focus is on miniaturization of the microkernel and embedded systems), and finally XtreemOS (a distributed OS based on organizations and built on top of Linux). Some ideas about these three OS are presented below.

2.2.1 Singularity

Singularity[9, 10] is an experimental OS developed by Microsoft Research in 2003. Its main objective is to achieve **high reliability**. For this reason they have started the development of the OS from scratch. It has been possible to experiment with new technologies and high-level languages to build the architecture of the OS, this way they've achieved a very robust and reliable system. Therefore, one of its most critical abstractions are *Software-Isolated Processes* (SIP), which represent Singularity processes. Any code running outside the kernel is running in a SIP. The SIP is a way of encapsulating software into separate and fault tolerant components.

2.2.2 MINIX 3

MINIX is one of the most popular microkernel OS still in development. Originally designed by Andrew S. Tanenbaum as a study Operating System for his students. Its design was a model for the construction of other Operating Systems, while it has continued its own evolution, reaching in 2006 the third version of the OS: MINIX 3[11].

The main objective of the third version of MINIX is **reliability**, devising for it a **self-reparable** system. They have followed the design philosophy of microkernel, leaving in protected-mode the minimal functionality and placing in user-mode all the remaining functionality. Thus, the user-mode failures are not critical for the system and also, due to a system called *Reincarnation Server*, failing processes are self-reparable and can be relaunched in the same state they have failed.

2.2.3 XtreemOS

XtreemOS[12, 13] is a Grid Operating System. The development of this system is based on the Linux OS and its objectives are transparency and scalability. Transparency is offered to both the user and the application, since the great advantage of XtreemOS is still offering a *Linux* interface despite the availability of certain services and resources distributed on the network. Furthermore, this transparency allows heterogeneity among the classic applications of Linux and those found in the Grid.

XtreemOS uses virtual organizations (VO) to encapsulate the services and resources in the Grid. A VO administrator is responsible for its creation, management and completion, whether it is static and dynamic.

2.3 Discussion

The biggest innovation in the field of operating systems has probably been the introduction and expansion of the network. The leap from single centralized computing in a distributed computing in all computers across the net, which is called *cloud*, has emerged a complete redesign of the operating systems to adapt themselves to this new technology.

The functionality that is demanded today from an OS has changed from what was being demanded lately. Factors such as cross-platform, multi-processor support or concurrency ability do not pose a technological challenge today, as we discussed earlier, and are in the vast majority of new developments in operating systems. Key factors that make a difference in the new OS are related to the network (such as being distributed, single image, access to services, transparency ...) and those related to security and integrity of information. Two other important factors remain the efficiency of the system (as much as you increase the speed of the hardware it is still important that the OS interferes as little as possible in the response time of applications), and one factor which becomes more important every day because the increasing complexity of applications: reliability.

Our proposal is to focus on major current challenges of computing science that are not solved by existing OS: the presence in the network, service-orientation and, of course, the three major design factors inherited from the evolution of old OS: performance, security and reliability. For all this, our proposal is oriented to increase the level of the abstractions provided by the operating system and their services. This makes possible to offer an OS layer integrated to the network, and security and reliability mechanisms not available in lower levels of the architecture of the OS.

These changes begin by replacing the paradigm that is used. Changing the abstractions that an OS uses is linked to the paradigm used. This new computing paradigm is presented next.

3

Distributed Goal-Oriented Computing

In our work we define the concept of Distributed Goal-Oriented Computing as the paradigm where heterogeneous agents can express their desires by using goals. These agents can also fulfill the goals by using automatic composition of services that are available in the cloud. In this section the Distributed Goal-Oriented Computing paradigm is presented by means of showing the model that defines it and the architecture that gives support to the related model.

3.1 Goal-Oriented Execution Model

The Goal-Oriented Execution model is inspired by the classic BDI agent model presented in [14]. In this model there are included the abstraction of *agent*, *knowledge base*, *services*, *goals* and *plans* (which are the service compositions) [15]. Its purpose is to define an execution support based on a different computation paradigm that provides the features presented earlier in this document. The execution model operates on an operating system kernel, which provides the other necessary functionality of a common OS, such as memory management, security, etc.

In the Goal-Oriented Execution model, an agent A is defined through the following tuple:

$$A = \{KB, SS, CP, GS\} \tag{3.1}$$

where:

- KB represents the agent Knowledge Base.
- SS represents a Set of Services offered by the agent. This services are used by the agent to perform its goals, but they can also be offered to other agents to help to achieve their own goals.
- CP represents a set of Compiled Plans provided by the agent to meet its goals.
- SG represents the Set of Goals that the agent wants to achieve.

The services that the agent can offer in the Goal-Oriented Execution model are OWL-S services. An OWL-S service is defined by the tuple:

$$S_i = \{SP, GR, PM\} \tag{3.2}$$

where SP is the Service Profile, GR the Grounding and PM the Process Model of the service. The service profile defines *what* the service does. The grounding defines *how to interact* with the service and the process model defines *how* the service is used.

Moreover, OWL-S service process model can be *composite processes* and *atomic processes*. A composite process is a set of atomic processes (which have no internal structure and run in a single step) with an internal structure built up by composite and atomic processes and a few control constructs (`sequence`, `if-then-else`, `choice`, etc).

This kind of OWL-S service is a well-defined standard which provides this model enough power to construct all the functionality provided by an agent. The services that make up a plan are the real executable part of a plan. A service S_i is also composed of a *pre-condition* P, a *post-condition* Q and a set of *inputs* and *outputs*. The pre-condition P is a **prerequisite** for the execution of a service. The post-condition Q is the **impact** that will drive the execution of the service S and it represents the **Goal** entity that the agent wants to achieve. Both P and Q are defined in the *functional aspect* of the service profile.

3.2 Goal-Oriented Execution Architecture

Since a composition of OWL-S services is a composition of services, which include both atomic and composite processes and control structs, we define a *Plan* as a process model composed by one or more composite process models (again, including composite services, atomic services and control structs). A Plan defines the way to achieve some results or post-conditions by joining different OWL-S services which can be connected. Composite services or even atomic services can be seen as very simple plans, but we also define a plan as the result of joining different composite services in order to achieve a goal.

To give support to the model presented, a Goal-Oriented Execution architecture has been developed. The architecture is composed by the next components (Figure 3.1):

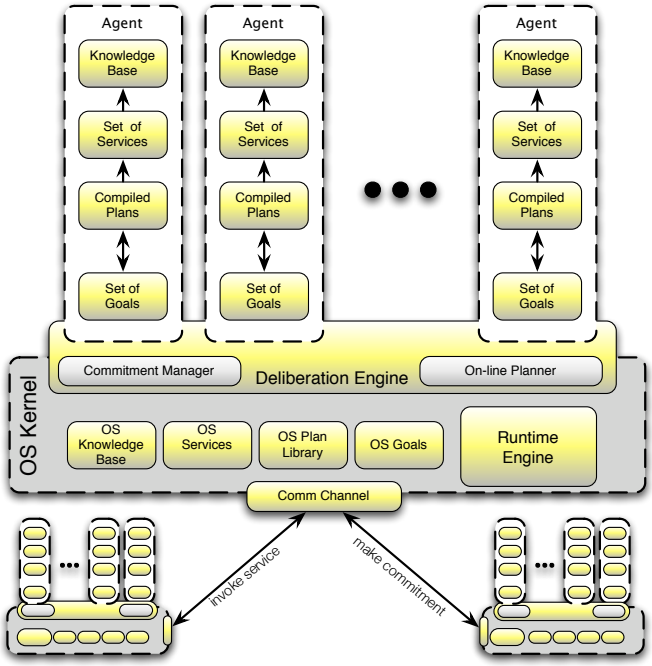

Figure 3.1: Agent and execution module components

- **Runtime Engine**: The Runtime Engine takes the plans provided by their planners and manages their execution by transferring the service execution to the OS kernel. It uses distributed services provided by agents in other hosts if necessary.
- **Deliberation Engine**: It is responsible for deciding how and in what order plans are executed. This engine negotiates with the agents which provide a service for a current plan. This engine is permanently running in background and evaluating the goals that are activated in the agents to be achieved and selecting them for its completion. This component interacts concurrently with the Runtime Engine, the Commitment Manager and the On-line Planner.
  - **Commitment Manager**: Service provider agents negotiate with the commitment manager their availability and, if so, quality and security parameters like their execution within a time window or the required encryption algorithm in transactions. The security parameters can be defined by both the client or the provider. The Commitment Manager will always try to reach these minimum security parameters by negotiating with all the available agents. However, if there are no required security parameters, the CM will always try to get the best deal for a transaction as secure

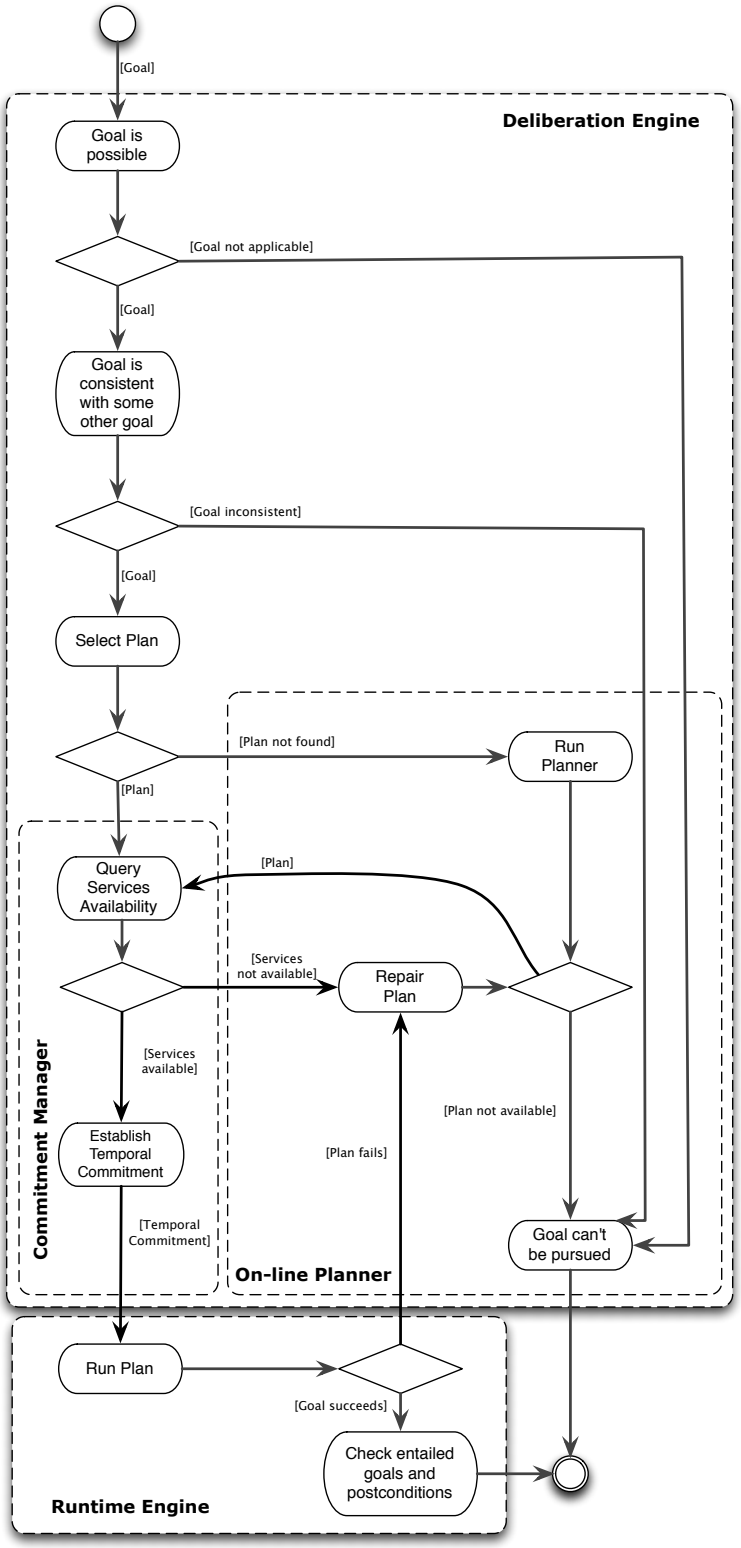

Figure 3.2: Goal-oriented executive model

as possible. To calculate the execution within a temporal bound the agent needs to take into account some points like: (i) the current workload, (ii) the availability of the service at the time of the request and (iii) the availability of the needed hardware and software resources to be able to run. For this work the agent needs the help of the OS. The OS can help the agent to predict if it is going to be able to satisfy the request in the defined temporal bounds, and if so, to establish a commitment with the Deliberation Engine. This functionality is offered by the Commitment Manager.

  - **On-line Planner**: it is able to compose new plans on-the-fly. It also repairs and refines running plans. This planner is executed concurrently inside the Deliberation Engine. Its task is to help the agents to reach a goal when the agent has no pre-compiled plans to guide it to the goal completion. This is done by composing or repairing plans. The On-line Planner uses a TB-CBP (Temporal Bounded Case Based Planner) to generate the plans at runtime. It uses past cases from the same service or similar services to generate a plan with a time prediction inside the established temporal bounds.

- **OS Goals Set**: The OS has its own goals to perform the corresponding tasks of an operating system. This set of goals includes all the maintenance tasks and non-critical functionality.

- **OS Knowledge Base**: This is the knowledge that the OS has. The operating system uses this knowledge base to perform their goals by means of the services that can invoke.

- **OS Services Set**: The set of basic services provided by the OS. This set of services is used by the OS to provide the basic low-level functionality to the system agents. It includes all the necessary stuff to manage the system and to access to restricted features only available through the OS for security and stability reasons. Some of these features are the communication of system drivers with the hardware, as well as other features that allow the correct interaction among agents, service providers and the operating system.

- **OS Plan Library**: It provides pre-compiled plans for their execution from a set of goals. This component is created in the design phase of the OS and its motivation is to provide pre-compiled plans for critical goals that cannot wait for a different composition or cannot vary their execution flow due to security and efficiency reasons.

Under the Distributed Goal-Oriented Computing paradigm the goals that the agents have are sent to the execution module for their achieve. Then, the deliberative engine chooses the appropriate plan to meet each goal. Note that the agent model preserves its desirable features like autonomy and pro-activity since the agent is who activates its own goals when he decides he wants to achieve them. The deliberative engine provides the needed resources to help to achieve the goals. Plans may be provided by the agent itself or can be compounded *on-line*. These plans are a sequence of **services** offered by the agents both locally and remotely. It is also

an agent choice to share its pre-compiled plans with other agents. The basic running elements are the services that make the plans. Plans are provided to the module in two different ways: the *off-line* generation of the plan or the *on-line* generation of the plan by the On-Line Planner module. Once the plan that meets the active goal is selected, the Runtime Engine activates the services that comprise the selected plan. In Figure 3.2 the deliberation and execution processes are shown.

Once the Goal-Oriented Execution model and architecture have been presented, in next section we are going to show the deliberation process that is used to fulfill the agent's goals.

4

Deliberation Engine

The Deliberation Engine is the *brain* of the execution module. This component is in charge of analyzing the current active goals and helping their achievement. The Deliberation Engine is the root node which manages all the main flow of the execution process. Its main task is to get a plan that fits properly with the activated goals. If the plan does not exist, the Deliberation Engine will compose a new plan using its component called On-line Planner. The On-line Planner returns a set of plans that guide the agent to the fulfillment of the goal. The Deliberation Engine uses two classifiers that help the agent to select the most proper plan. Both classifications are performed by the two components included in the Deliberation Engine: the On-line Planner and the Commitment Manager. The On-line Planner makes a first classification using the information retrieved from past executions. A second classification is done by the Commitment Manager. It finds the best providers which offer the required services to complete the plan. This classification establishes commitments with the provider agents to complete the service taking into account security and temporal constraints. Temporal commitments have different parameters like when the service must be run, when will the service end and the probability of finishing the service in that deadline. Security commitments present some constraints about three security concerns: authentication, intrusion detection and encryption. This two components are shown in more detail in Sections 4.1 and 4.2. Once a plan is selected, the Deliberation Engine sends it to the Runtime Engine to be run.

Since a goal is activated by an agent until it is achieved, the Deliberation Engine goes

through different steps which involve the different components of the execution module. These steps are:

1. Checking if it is possible to activate the goal.
2. Checking if the goal is consistent and there are no conflicts.
3. Asking the On-Line Planner for a set of plans to achieve the goal.
4. Querying the Commitment Manager for temporal and security commitments for each service of the plan.
5. If there is no available commitment, asking the On-Line Planner for a new plan or setting the goal as unreachable.
6. Selecting the best plan from the set of plans using the temporal and security commitments and the information retrieved from past executions.
7. Sending the plan to the Runtime Engine to be executed.
8. If the plan fails, asking the On-Line Planner for a new plan or setting the goal as unreachable.
9. When the plan ends, updating the case-base with the results of the commitments, penalizing or rewarding the providers if necessary.
10. Checking entailed goals and postconditions and setting the goal as reached.

Next the components used by the Deliberation Engine to determine the service composition are presented in more detail.

4.1 On-line Planner

Within the execution module, the responsible entity for providing plans that fulfill the agents' goals is the *On-Line Planner*. This component generates plans composed on-the-fly that achieve the goals that are activated by the agents. This generated plans complement the static precompiled plans provided by the agent Plan Library. The On-line Planner is based on a CBP (Case-Based Planning) methodology [16] that has been modified for giving a temporal bounded response. This new model (called Temporal Bounded CBP) is composed by the same phases as the classic CBP, but these phases have been treated to bound their execution time. Thus, the execution time of the service composition process is known and is taken into account when the *On-line Planner* is building a plan within a maximum time. This work has not as purpose

to introduce in detail the characteristics of a TB-CBP. A comprehensive description of this approach can be found in [17]. Anyway, a general description of the functioning of the *TB-CBP on-line planner* is shown below.

The case structure used in the TB-CBP is defined as: ¡$Postcondition$, $Precondition$, {$Service$}, $SuccessRate$, $ExecutionTime$¿, where $Postcondition$ is the goal wanted to be achieved. $Precondition$ are the initial conditions that must be given to start the execution of services necessary to fulfill the goal. $Service$ is the list of services that must be executed from the state $Precondition$ to reach the state $Postcondition$. $SuccessRate$ indicates the percentage of executions successfully completed in the past. This term represents implicitly the confidence that the system has about this composition. Finally, $ExecutionTime$ is the time required for the execution of the services. This value is obtained by calculating the worst-case execution times of each of the services included in the composition and combining them following the process model of the composition. This $ExecutionTime$ term is performed to get a temporal estimation of the execution of the whole composition and to use it in temporal commitments. Temporal commitments will be shown in Section 4.2. An example of the used case-base is presented in Table 4.1, where P is $Precondition$, Q is $Postcondition$, SR is $SuccessRate$ and ET is $ExecutionTime$.

Table 4.1: Example of Case-Base of the TB-CBP

Q	P	Services	SR	ET
B	A	{S1}	1	4t
C	A	{S1,S3}	0.85	10t
C	B	{S3}	0.85	6t
D	C	{S6}	0.9	7t
E	B	{S7,S10,S11}	0.76	11t
E	D	{S8}	0.99	3t
E	D	{S4,S12}	0.98	7t
F	C	{S5,S9}	0.81	7t
F	E	{S13,S14}	0.98	10t
...	...	...	...	...

To complete the search of a service composition, the agent will inform about the activated goal ($Postcondition$) and its knowledge base ($Precondition$). With this information the *On-line Planner* can fulfill a service composition. To do it, the planner extracts cases from the case-base and composes a path from the goal to be achieved until it reaches any of the beliefs that are stored in the agent.

Let's imagine the following situation using the information in Table 4.1. An agent has as requirement the fulfillment of the goal F, and has in its knowledge base the items {A,B}. These items can be used as preconditions for the fulfillment of the goal. The *On-line Planner* will extract from the case-base all cases that have the goal F as $Postcondition$. For every

extracted case the algorithm will come to search in the case-base, but now $Postcondition$ is the set of preconditions of all the extracted cases ($Precondition$ parameter). This process will follow until it extracts a case whose $Precondition$ is either defined in the agent's knowledge base ($Precondition = A \vee B$). In Figure 4.1 we can see the progress of the search from F to A or B. In this case, several plans are possible. In response to the needs of both the agent or the Operating System just one plan will be chosen. In order to get a result with the best success rate, any of the plans marked as (3) is picked. If it is required to get a plan that reaches the goal as soon as possible, the plan marked as (1) will be chosen. Finally, if a plan that meets within a specified temporal bound (e.g. before 22 time units) and with the highest success rate is required, then the option (2) will be selected. As shown, the execution module can choose a plan taking into account the agent requirements, making the system more adaptable.

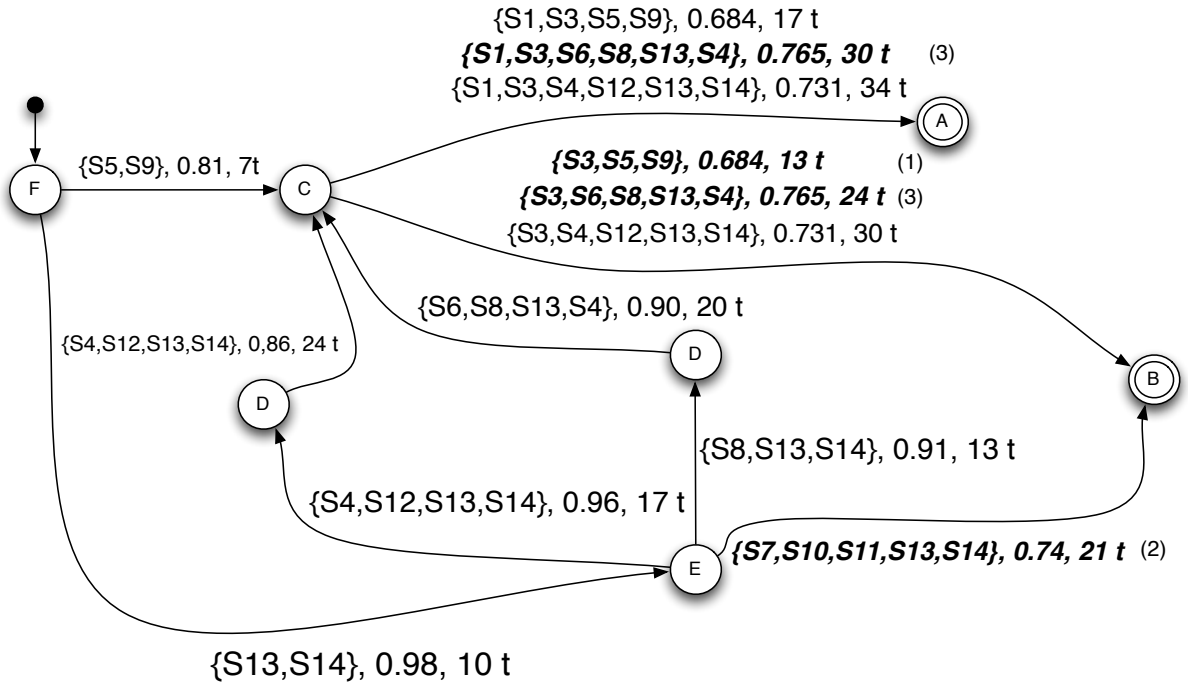

Figure 4.1: Search sequence in the case-base

This component applies a first classification to select a plan that fulfills the active goal within the agent requirements. This classification has into account past executions of the cases that have been retrieved from the TB-CBP. The plan selection is done by using the success rate and the execution time of the retrieved compositions. Thus, the On-line Planner uses static knowledge to select the plan but it has not into account the environment conditions and the agents workload at the current moment. Since a plan is composed by single services and each service can be provided by different agents at the same moment, a second classification must be done to be able to select the best providers for each single service of the plan. This classification

does take into account the security mechanisms of the provider host and the agents workload at the current moment. This function is performed by the Commitment Manager which is presented below.

4.2 Commitment Manager

This component is designed to select the best provider agents that offer the single services of a plan that has been selected by the On-line Planner. The Commitment Manager is related to a framework called *SAES* [18] which allows us to compose services and to ensure their fulfillment on time. The main difference with the SAES approach is that, by introducing the service framework as part of the operating system, it has more information to make better security commitments and temporal predictions.

The main function of the Commitment Manager is to check if the set of services offered as a plan by the On-Line Planner will be available to fulfill the request and to establish two kinds of commitments with the agents that provide the selected services: security and temporal commitments.

Security Commitments

One of the main purposes of this work is to establish a secure environment where services can be invoked with some grants of privacy, integrity and access control. If we focus on the current challenges in infrastructure security we can organize them in three levels: the Network level, the Host level and the Application level [19]. Since there are different network topologies, both in public and private clouds, and the Commitment Manager has no management abilities over the network, the Network level is out of the scope of this work. As the Commitment Manager is integrated into the OS we can afford security concerns at the Host Level. At this level we can apply Host-based Intrusion Detection Systems (HIDS) to keep data integrity. At this level it is also possible to apply audit mechanisms and server virtualization. The access control mechanism can be also placed at this level. Finally, the Application level depends exclusively on the application program, this is, the agents. At this level the agent can manage encryption mechanisms, application authentication and authorization and secure coding.

The Commitment Manager classifies the security level of a service in three categories: authentication[20], detection[21] and encryption[22]. The better the security level is in each category, the more confidence will be deposited at the service provider. Each CM has a table to prioritize different mechanisms for each category: cryptographic algorithms for the encryption level, intrusion detection systems and firewalls for the detection level and access control systems for the authentication level. As an example, Table 4.2 shows a little set of authentication mechanisms and how the Commitment Manager would rate them.

Table 4.2: Example of authentication mechanisms and their rating

Auth mechanism	Rating
One-time password	0.95
Time-based authentication	0.87
Two factor authentication	0.91
Closed-loop authentication	0.73
Username and password	0.1
Digest Access authentication	0.4
...	...

The CM asks for security information to each agent that is providing any of the services included in the selected plan. Each agent answers notifying its security level with the tuple $< A, D, E >$, where A stands for authentication and authorization mechanisms, D stands for detection and auditing mechanisms and finally E stands for encryption mechanisms. The CM assigns a value to the security commitments by adding the values of the tables that represent each of the A, D, E categories (like Table 4.2) and applies a weight with the confidence that the CM has in the provider agent. If security has failed too much in the past and the agent has been penalized, its confidence will be low and, therefore, the applied weight will be low.

Temporal Commitments

To establish this kind of commitments the Commitment Manager sends a call for proposals to all agents that can offer the services involved in the composed service (see Figure 4.2). Each agent analyses when the service can completed, and then each agent returns a proposal to the Commitment Manager. The proposal consists of a tuple $< T_{start}, T_{duration}, P >$ where T_{start} indicates the moment when the service can start its execution, $T_{duration}$ indicates the necessary time to complete the service and P is the probability, as said by the provider agent, of finishing the service successfully. This value represents the probability of reaching a successful execution and is extracted from its success rate of executions.

On one hand, the On-line Planner obtains a quality measure (the success rate) which is used to estimate the best plan. On the other hand, the Commitment Manager calculates a probability that indicates if the composition can be completed in time, taking into account the service provider agents workload. This information refines the success rate obtained by the On-line Planner because it takes into account the current situation of the agent that offers the service and the real conditions of the environment. With all this information, a pre-commitment between the agent and the Commitment Manager is established.

When all agents have answered to the Commitment Manager, the CM must calculate the success probability associated to the whole service composition. To do that, the Commitment Manager uses the P value, which was sent by all agents. This probability is weighted with

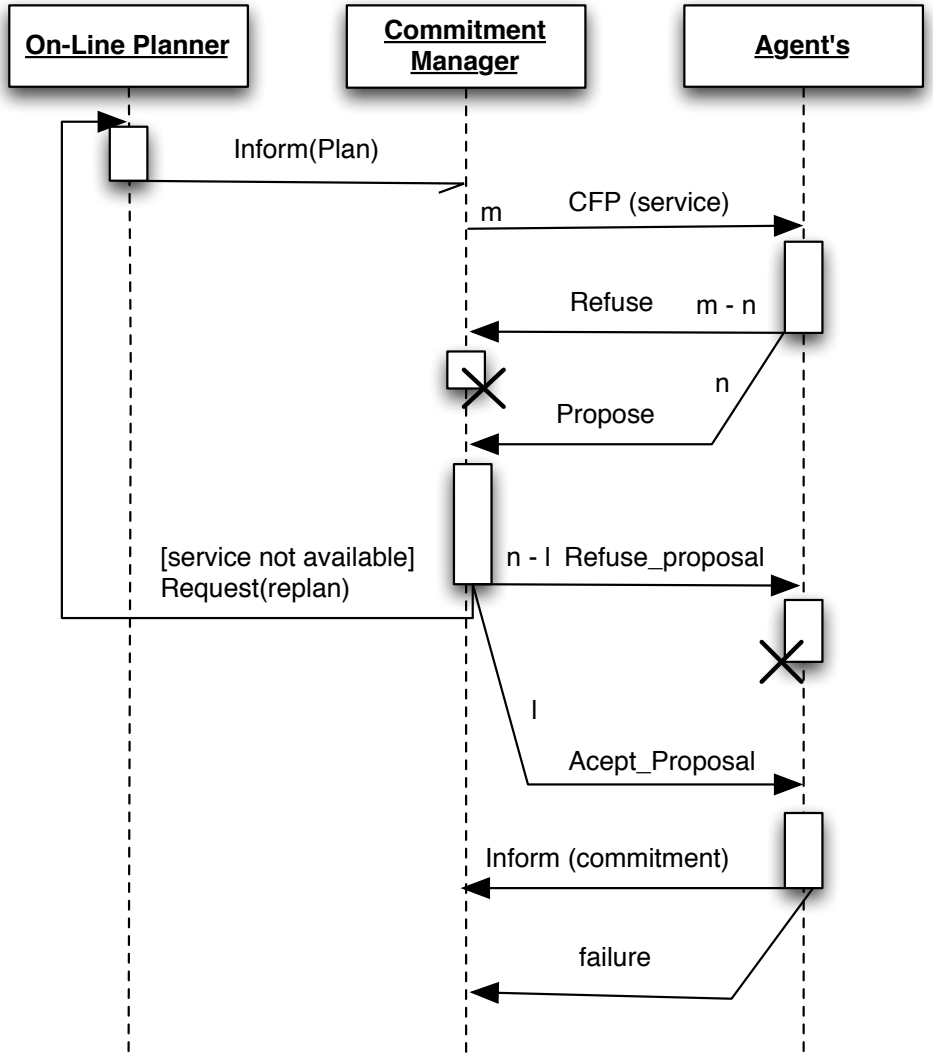

Figure 4.2: Services Availability Query interaction protocol

the confidence that the Commitment Manager has on these agents. The service composition success probability ($SCSP$) is calculated as follows:

$$SCSP = \prod_{i=0}^{N} P_i * \omega_i$$

where $\omega_i \in [0, 1]$ is the weight associated to the agent that provides the service. This weight is related to the previously fulfilled commitments and represents the confidence that the Commitment Manager has on this agent; e.g. an agent that has many unfulfilled commitments will have a low confidence.

Once the Commitment Manager calculates the security level and the service composition success probability, it sends the composed service and the temporal and security commitments to the deliberation engine. The deliberation engine analyses if it is a suitable composition. If

it agrees with the service composition, it communicates to the Runtime Engine that the service executions can start. When this is the case, the pre-commitments established with the agents are confirmed by the Commitment Manager. If the deliberation engine does not agree with the service composition, the Commitment Manager breaks the pre-commitments, freeing the slack reserved by the agents.

The Commitment Manager is also in charge of ensuring that the acquired commitments are fulfilled. In case where a commitment cannot be fulfilled, the Commitment Manager penalizes the agent which provides the service. This penalty is captured through the confidence weights that are applied when the Commitment Manager updates the service composition success probability.

5

Runtime Engine

The Runtime Engine is the component in charge of managing the entities that are running in the system. This includes driving the execution of the process model of the active plans and scheduling the services that are invoked by a plan, both the local and the remote invocations.

The execution of an atomic *service* is much like a traditional operating system's process abstraction. These services are scheduled and executed by the Runtime Engine with a proper context. These services have also a life cycle inherited from traditional processes[23]. The states of the service life cycle are: (i) `ready to run`, (ii) `running` and (iii) `sleeping`.

As stated before, the Runtime Engine also manages the life cycle of *plans*. The execution of plans is made in collaboration with the Deliberation Engine's Commitment Manager. While the Commitment Manager is in charge of ensuring that the temporal commitments are achieved, the Runtime Engine checks that every step of the plan is properly executed. This includes to ensure that, before executing a service, all its preconditions are true and that, after executing the service, all their postconditions have been achieved. This part is carried out by following the OWL process model (PM) at each step, following the logical flow that determines its preconditions and postconditions. The task of visiting the process model of each active plan and check the preconditions and postconditions of each node belongs exclusively to the Runtime Engine.

Algorithm 1 shows the steps followed by the Runtime Engine:

1. The Runtime Engine (RE) extracts a plan from the list of selected plans created by the Deliberation Engine.
2. The first action is to check that the plan's precondition is valid and can be executed.

```
0.1  foreach Plan in selectedPlans() do
0.2      if checkPreCondition(Plan) == True then
0.3          ServiceQueue = emptyQueue()
0.4          n = selectFirstNode(Plan)
0.5          append(ServiceQueue, n)
0.6          while hasNodes(ServiceQueue) do
0.7              n = getNode(ServiceQueue)
0.8              if checkPreCondition(n) == True then
0.9                  invoke(n)
0.10                 if checkPostCondition(n)== True then
0.11                     foreach Node in neighbors(n) do
0.12                         append(ServiceQueue,Node)
0.13                     end
0.14                 end
0.15             end
0.16             remove(ServiceQueue, n)
0.17         end
0.18         if checkPostCondition(Plan) == True then
0.19             return True
0.20         end
0.21         else
0.22             replanning()
0.23         end
0.24     end
0.25 end
```

Algorithm 1: The Runtime Engine algorithm

3. At this moment the plan is selected as a running plan. The RE selects the first node of the plan from its service graph and invokes the service by appending it to the scheduler's ready queue.

4. Before executing a service the Runtime Engine previously checks its precondition and, after the service execution is finished, it checks the service postcondition. If the postcondition is valid the execution of the plan can continue.

5. Once the service finishes its execution, the RE extracts from the process model all its neighbors and checks their preconditions. These neighbors are all the nodes that are directly accessible from the given node through a control construct.

6. This process continues until the service process model reaches a final node or their ser-

vices fail and a plan reparation is needed (using the On-line Planner).

7. When the plan finishes, the Runtime Engine checks its postcondition. If it is valid, the goal that has motivated the execution of the plan is marked as pursued. Otherwise, a new plan is requested to the On-line Planner.

The *Agent* is the main entity that motivates this execution model. Agents can flow through different states, depending on their current role:

- **Applicant**: The agent has goals to pursue and does not offer any service.
- **Provider**: The agent offers services to other agents but has no current goal.
- **Provider-Applicant**: The agent has goals to pursue and also provides some services for both its own use and for other applicant agents use.
- **Inert**: The agent has neither current goals nor provided services. This is the case when the agent is ready to leave the system.

Once the Runtime Engine executes a plan it notifies the On-Line Planner in order to perform the *retain* step, this is, to store the new case (whether it is successful or not) to keep the case-base updated.

6

Execution trace

This chapter will expose a sample trace where the different steps that this execution module follows to achieve a goal are shown. For simplicity we have prepared a simple scenario with a few elements and a single goal to achieve. To show the flexibility of the system we will simulate an error in the trace, showing the fault tolerance of the module.

In this example there is an agent that acts as an interface of the user (the client agent) and a set of services distributed around the different nodes of the network. Each of these services is provided by an agent and is hosted in a node which is connected to the node where the client agent is hosted. The prepared scenario is designed to perform a very common task: *saving a song in an iPod*. In this scenario the client agent just expresses its goal (**`Song in Ipod`**), and has some previous knowledge in its knowledge base: the audio he wants to save (**`PCM Audio`**) and some metadata (title, author, genre,...) about the song (**`Song Metadata`**). These knowledge items will act as the preconditions of the plan that is going to be executed.

When the client agent activates the goal the Deliberation Engine looks for a plan to fulfill the goal. Since there is not a plan that is able to perform the goal expressed by the client agent, the On-line planner generates a plan that is able to perform the desired goal starting from the known KB items as preconditions. This plan is shown in Figure 6.1. The iPod only works with MP3 encoded audio (and the precondition expressed by the agent is encoded in raw format), so the generated plan will include the needed services to encode the raw audio to the MP3 format. The services that encode audio were previously executed in the system, that is why there is a Case included in the generated plan in Figure 6.1. The dashed box represents a service provided by the operating system of the client agent. For this reason the service is hosted in the same node than the client agent.

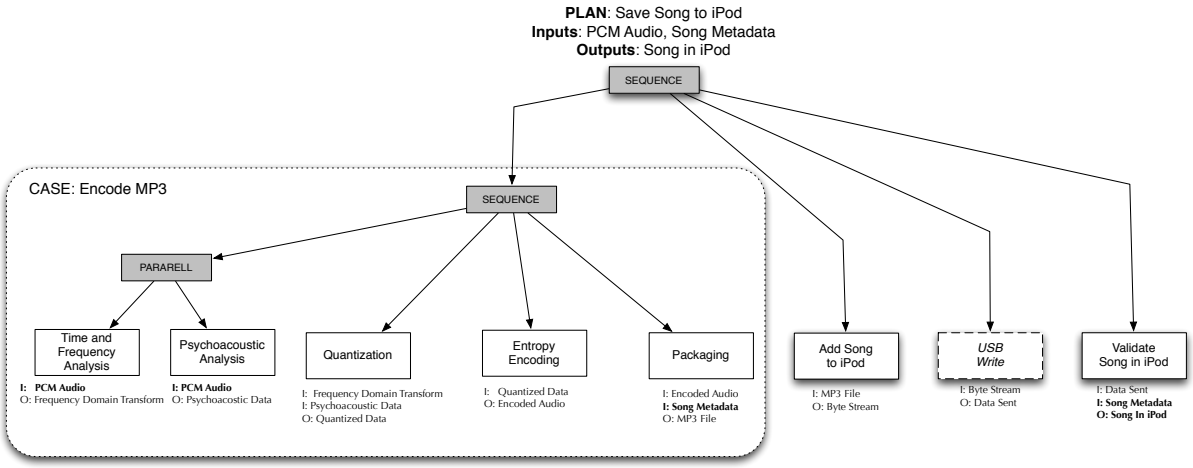

Figure 6.1: Process model of plan `Save Song to iPod`

As an example, we'll follow an execution trace using this plan:

1. Initially, the Deliberation Engine would select a goal of an agent. For simplicity there is only one goal, which is *Song in iPod*. Since there is only one goal, the Deliberation Engine selects it.

2. The On-line Planner generates a plan to fulfill the goal (Figure 6.1), as stated before.

3. As long as the plan meets the precondition (the agent *knows* `PCM Audio` and `Song Metadata`), the deliberative engine will select the plan for its execution since its post-condition is compatible with the desired goal (it generates `Song in iPod`).

4. The first services to be executed are `Psychoacoustic Analysis` and `Time and Frecuency Analysis`. Before running them, the Commitment Manager establishes temporal commitments with their hosts.

5. The Runtime Engine executes the services `Time and Frecuency Analysis` and `Psychoacoustic Analysis`, achieving as effects the values `Frequency Domain Transform` and `Psychoacoustic Data`. The Commitment Manager checks that the temporal commitments were accomplished, rewards the services and performs the retain stage in the Case-Base of the On-line Planner.

6. Next service is `Quantization`. After the establishment of the temporal commitments, the Runtime Engine executes the service `Quantization`, achieving as effect the value `Quantized Data`. Once again the Commitment Manager rewards the service and retains the case.

7. To show the advantages of running this model, we introduce an error at this point. Let us assume that the service `Entropy Encoding` is unavailable (the agent that provides

the service is not connected, the service is saturated, or maybe the output is not a real MP3 file). This situation generates that the Commitment Manager punishes the case representing the service.

8. At this time, the Runtime Engine would ask the On-line Planner a repair of the running plan to continue the execution of this agent.

9. The planner would return the plan shown in Figure 6.2. This repaired plan continues where the other plan has failed its execution and replaces the failed service with other structure thanks to other services found in the distributed system. The new plan has a very similar structure but replaces the encoding service with a choice for other three time domain encoding services (`PCM Encoding`, `Differential PCM Encoding` and `Adaptive PCM Encoding`).

10. At this moment the Commitment Manager needs to establish a commitment with the service which ensures a lower execution time and offers a better trust value. To do this, the CM asks the case-base for old trust stored values and asks the providers hosts about their temporal commitments. With this information the Runtime Engine selects for execution the `Adaptive PCM Encoding` service.

11. Finally the execution of the plan is ongoing through the services `Packaging`, `Add Song to iPod`, `USB Write` and `Validate Song in iPod`. At each step a temporal commitment is established and the service executed is punished or rewarded depending on the case.

12. When the execution of the service `Validate Song in iPod` has finished, the client agent has in its knowledge base the fact `Song in iPod`, so the goal has been achieved and it can be removed from the agent set of goals.

A remarkable aspect of the client agent is that despite the selected plan has failed, it has been able to achieve its goal on a completely transparent way to the agent through the ability of replanning of the execution module. With this module the success degree of goal achievement is higher than on classic BDI systems. This module has also the ability of providing system services for the plan composition, allowing the OS to work with this paradigm, as is the case of the `USB Write` service.

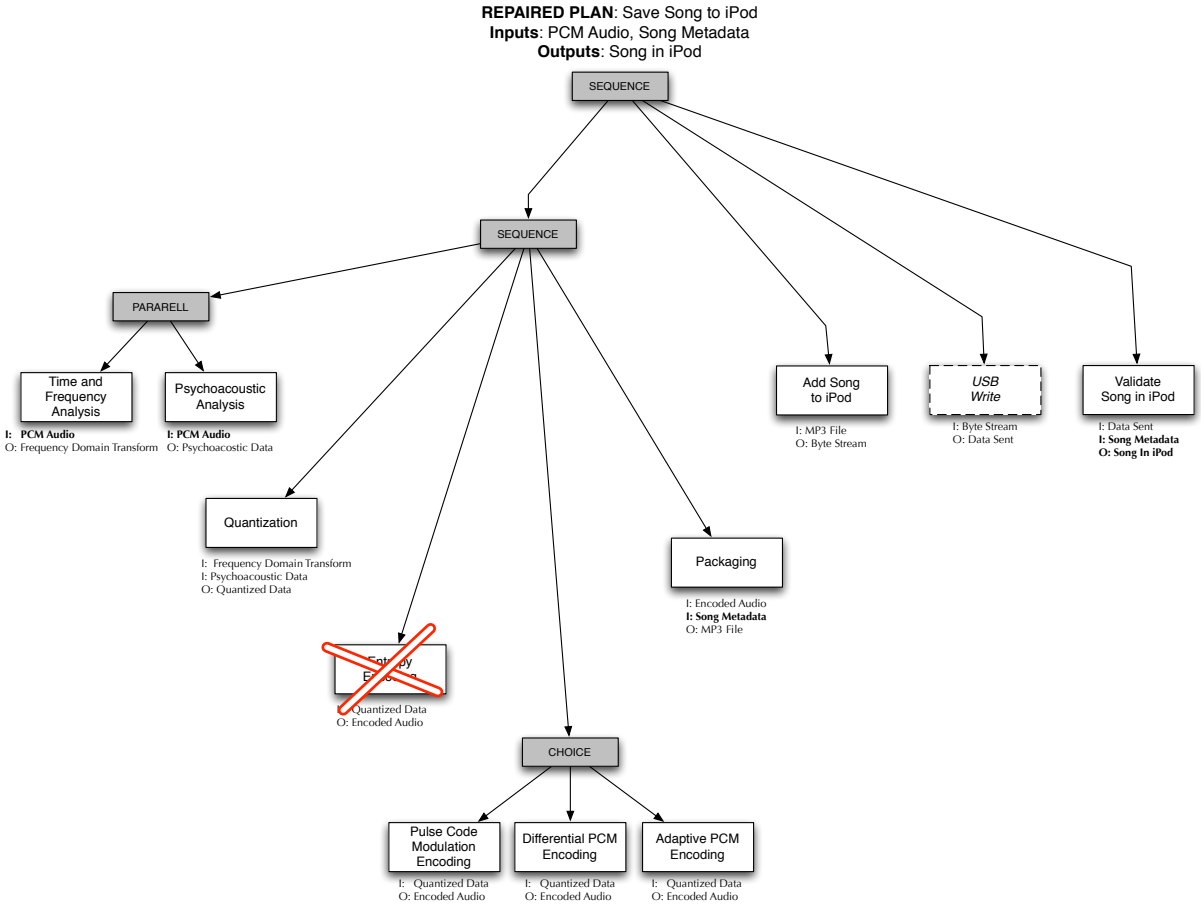

Figure 6.2: Repaired plan `Save Song to iPod`

7

Implementation and Results

In order to evaluate the architecture presented here for the development of goal-oriented operating systems, this work presents a set of tests and results that validate the proposal. A discrete simulator has been developed to test all the features and advantages provided by an Operating System implementing the Distributed Goal-Oriented paradigm. In this section we present how the simulator works and how the different tests that have been done by analyzing what the different components of the proposal (runtime engine and deliberation engine components: commitment manager and on-line planner) contribute to the system.

7.1 The simulator

The operating system simulator allows us to test the provided functionality proposed by this work, but avoiding the complexity of developing the full operating system low-level abstractions. This simulator implements the main components of the goal-oriented operating system execution module that are needed for our purposes. This is mainly the execution module, which comprises the runtime engine and the deliberation engine (including the on-line planner and the commitment manager). The runtime engine is in charge of executing the services that are invoked by any running plan. The deliberation engine selects the goals that are acti-

vated and finds a plan which performs the goal within a temporal commitment.

The simulator also supports the representation of a distributed environment, where there are several goal-oriented operating systems which offer their services in a shared network using a common publish-subscribe protocol (like ZeroConf[24] or XMPP[25]). Thanks to this protocol, when an agent registers or unregisters a new service every OS in the same network receives a notification of this event and updates its case-base.

The environment also simulates a *global time service* which synchronizes the clock in every OS in the network. If a new OS is added to the environment it gets automatically synchronized with the rest of the system. This global time service is very useful for establishing proper temporal commitments and uses known solutions for clock synchronization in distributed real-time systems[26].

Every operating system in the simulation environment has also a communication module that is in charge of managing communications between the nodes of the network. A node is a representation of a goal-oriented operating system. This message passing system simulates a time-bounded environment which allows us for predictability of end to end operations.

Since this is an ad-hoc simulator developed for our OS testing purposes, it allows us to change some parameters in order to explore some interesting behaviors. We can parameterize architectural issues, such as the number of nodes in the network, the number of agents per node, or the number of services or goals that an agent has. The simulator has a scripting system that loads a configuration for the desired environment. The script can define the initial configuration of the environment (number of nodes, agents, distribution of the services by agent, goals, preconditions, etc), setting up the scenario that is desired for the simulation. It can also schedule different events that will be processed during the simulation in order to change the environment at runtime.

To compare the different behaviors, a set of internal parameters can be changed. The probability that a service fails during its execution is parameterizable in the simulator. This way we can check how the operating system behaves in a fault tolerant environment. We can also modify the precision of calculating a temporal commitment in the simulator. Changing the prediction algorithm or the quality of the algorithm itself we can compare different nodes having responses that are not equal for a same request. This is a good way of detecting how the system adapts itself to a changing environment. This kind of tests will be presented in next sections.

Below all the tests have been conducted using the same methodology and with at least 20 repetitions to extract a *statistically significant* mean and standard deviation. This *test of significance* ensures with great confidence that the *null hypothesis* was avoided.

Next, the set of tests performed in this work are presented. They have been divided into two main test suites: Deliberation Engine Tests, where its main components have been tested

(Commitment Manager and On-line Planner), and Performance Tests, where some advantages of this distributed system are presented.

7.2 Deliberation engine Tests

The deliberation engine is the component that introduces a reasoning process in the proposed operating system. It is in charge of selecting the best available services that can fulfill the activated goals and with the best possible conditions. The deliberation engine components that perform this functionality are the On-line Planner and the Commitment Manager. In this work we have developed a set of tests in order to validate their expected functionality.

7.2.1 Commitment Manager

The main aim of the Commitment Manager (CM) is to establish temporal commitments between a service provider and a client. When the client invokes a service he needs to communicate to the service's Commitment Manager to get a proper time prediction of **when** the service response is going to be ready. This prediction is not an easy estimation, since there are lots of factors than can influence in the results (mainly, the workload of the system). The CM must work side by side with the Runtime Engine, which schedules all the running services in the resource (the microprocessor). The scheduling algorithm is very important for the prediction task, since it has to be able to accomplish the established temporal commitments. At the same time, it should get a good performance and a high degree of interactivity in the system.

The way to do this is through resource reservation. The Runtime Engine reserves *at least* the 50% of the current remaining processor using first-come priorities. This scheduling algorithm ensures that each service will have a minimum of resources allocated for its execution. This means that if a service has a 50% of processor and another service has a 25% of processor, since the first service has twice the allocated resources, it will work twice faster. This is a pessimistic case because the slack time is shared by all the running services.

This algorithm ensures that a running service has a percentage of processor assigned, so it is easy for the Commitment Manager to calculate the response time and establish a temporal commitment with the client. Since the priority is assigned using first-come preferences, and it always assigns the half of the remaining processor time, the Commitment Manager can calculate the response time ($\Re$) using the equation showed in 7.1.

$$\Re_P = 2^P * WCET_P \tag{7.1}$$

Where, P is the priority of the service and $WCET_P$ is the worst-case execution time of the service, which is provided by the service provider agent. This is a pessimistic approach since it

ignores the slack time that is gained when the processor is idle. Improvements to this algorithm are being prepared and will be proposed in future work, including priority promotions when a service finishes and estimation of time gained when there are priority promotions.

Below are the tests that check the proper functionality of the Commitment Manager. These tests show how the system tries to select always the best services that are available to perform the agents goals.

Test 1: Trust evolution for different deadline predictions

In this experiment we are going to show how the trust that a client node has in different provider nodes evolves as time passes, focusing their requests on the more reliable nodes. The trust will be changing due that not all the nodes in the distributed system have the same accuracy when calculating the deadline predictions.

The first experiment has being designed using the following scenario at the initial state:

- There are 3 agents registered in the system: 1 client agent and 2 provider agents, which offer the *same* service with the same precondition P and postcondition Q.
- The network is composed by 3 nodes: Each agent is hosted in one of the three nodes in the same network.
- The client agent has the necessary knowledge to run the service (P) and activates the goal G which is the same as the two services postconditions (this is, G=Q).
- The nodes that host the service have different accuracies to calculate the response time:
  - First node has an accuracy of 90% calculating the deadline prediction of the response time (called *GoodProviderHost*).
  - Second node has an accuracy of 20% calculating the deadline prediction of the response time (called *BadProviderHost*).

Each experiment makes a request to any of the available services every time step. This is, the client agent activates its goal and selects a plan to perform its goal. After the execution of the service, the agent resets its knowledge base and re-activates the goal once more. As time passes, the case-base acquires more experience about the nodes confidence. Figure 7.1 shows the results of running this test. The X axis represents time (in simulator steps) and the Y axis represents the cumulative sum of services provided by each node, which is a good representation of the trust that the client has in each node. At the start time, the trust in each node is equal. This is because there are no previous known experiences and the case-base of the client is empty, so the client has the same trust on each provider. As time passes, the number of invocations to each node varies due to the accuracy of the *BadProviderHost* is not very good

and he fails continuously when calculating a proper response time. This makes his trust value going down and, therefore, most of the invocations are done to the *GoodProviderHost*, as is presented in the related figure.

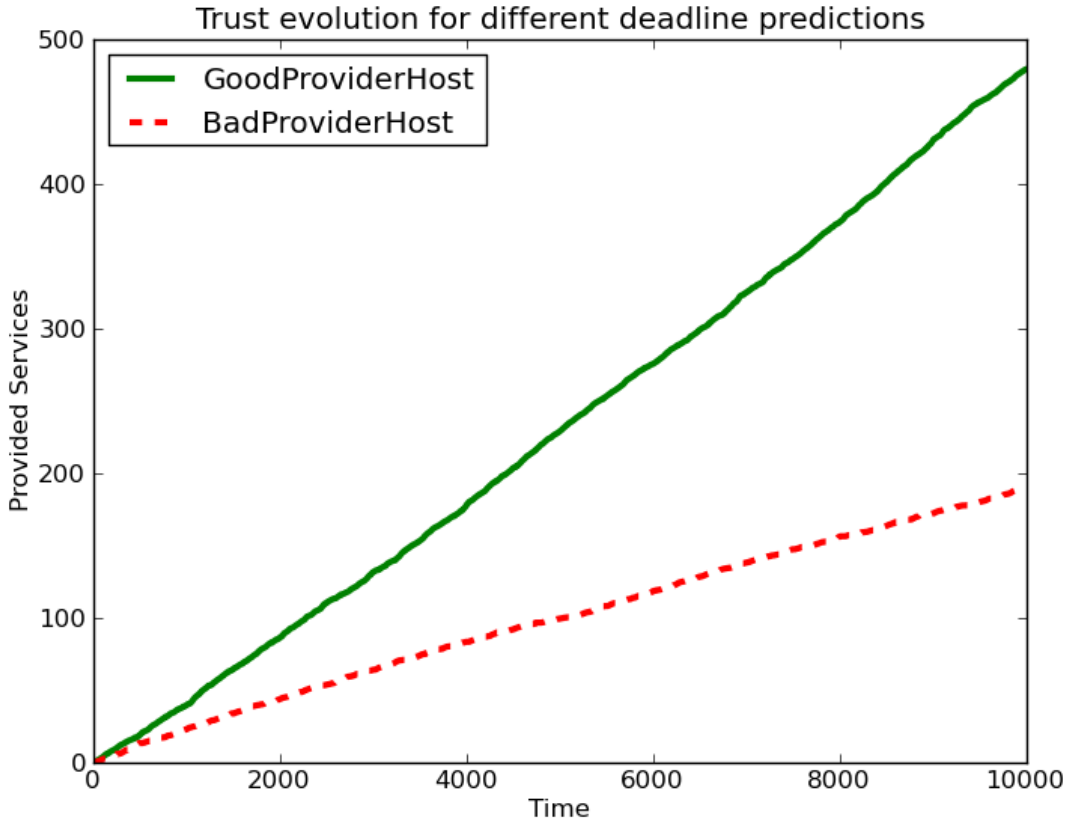

Figure 7.1: Test 1: Trust evolution for different deadline predictions

Note that the Deliberation Engine is not *only* using the trust value (extracted from the case-base) to determine which service provider to choose. The Deliberation Engine gives a chance to other providers by using an on-line learning algorithm[27] which decides to explore or exploit its solutions. This is done by adjusting a threshold value during the execution of the operating system. That is why the *BadProviderHost* provided services are not stuck. They grow more slowly than the *GoodProviderHost* ones, but sometimes have a new chance.

Test 2: Trust evolution in a bigger scenario

This experiment shows a similar approach to the previous study (Test 1). The main difference of this test is the size of the agents and nodes sets, which is bigger than in Test 1. This test shows

how the trust in the nodes with a good deadline accuracy grows while the system learns about the environment. The scenario is designed with the following elements:

- There are 51 agents registered: 1 client agent and 50 provider agents which provide the same service.
- There are 51 nodes in the network: Each agent is distributed in one node. Only one agent per node.
- The client activates the goal that invokes the service offered by the providers.
- The deadline accuracy of the nodes is distributed equally in four groups: 5%, 33%, 67% and 100%.

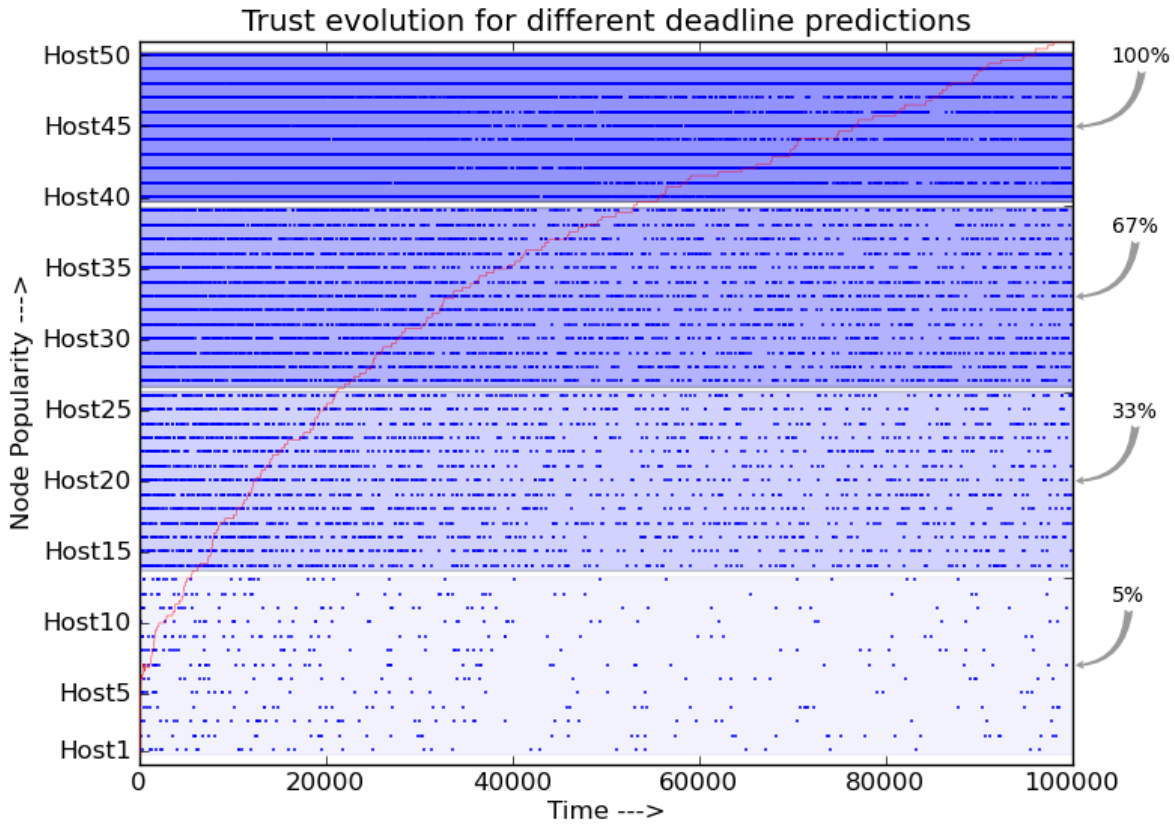

Figure 7.2: Test 2: Trust evolution in a bigger scenario

The execution of this experiment is equal to the execution of Test 1. The client agent resets its knowledge base and re-activates the goal every time step. Figure 7.2 shows the results of

the experiment. The X axis represents once more the time. The Y axis represents if a service was requested by the client at each time instant. Each dot represents a request to the node that is represented in the Y axis. So, the density of the dot cloud shows how *popular* is a group of nodes. When the density is large enough and the dots are very close, the representation becomes a straight line.

The middle line is just a mark to divide the dot cloud in the dense area (top-left) and the sparse area (bottom-right). Both areas show how the vast majority of the service requests are in the dense zone. This is again because the OS case-base learns, as time passes, which hosts are more reliable. These results demonstrate that the behavior of the system is what was expected. As we increase the time, the number of requests to the less confident nodes gets decreased.

Test 3: Adaptive Operating System

This experiment shows how the Operating System is able to adapt itself to changes in the environment. The adaptation of the system is very important, since it allows the system to have a dynamic behavior which is able to re-configure itself to take full advantage of current circumstances. For this test we have designed an scenario formed by the following elements:

- There are 5 agents registered: one client agent and four provider agents, offering the same service.
- There are 5 nodes in the network: each agent is distributed in a different node.
- The client activates the goal that invokes the service offered by the providers.
- The accuracy of the nodes at the initial step is distributed as follows:
  - Host1: 100%
  - Host2: 75%
  - Host3: 50%
  - Host4: 25%

In this experiment we are going to change the accuracy of some of the nodes to show how the system adapts itself on changing environments. We are going to activate 3 events to change the environment. Specifically, the following events have been scheduled:

- Step 50000: Host 1 accuracy decreases to 20%
- Step 300000: Host 3 accuracy increases to 80%
- Step 600000: Host 4 increases to 90% and Host 2 decreases to 20%

Figure 7.3 shows how the trust of the nodes (Y axis) changes when the environment undergoes these major changes (marked with the vertical bounding boxes). This trust value represents the trust that the client node has in the other nodes. Adaptation takes time to occur due to the learning algorithm that the deliberation engine is applying. In step 50000 we can see how the *Host 1* stops increasing its trust due to the first event. Note that this change takes some time to occur. When the second event occurs (step 300000), the trust value of Host 3 begins to increase (its deadline prediction is improved by 80%). Meanwhile the Host 1 trust continues decreasing and the other two hosts maintain their trust value. This third event changes again the system behavior, giving more trust to the *Host 4*, which has increased its accuracy to 90%. Its trust is growing quickly since its new accuracy is quite good. Parallel to this, Host 2 begins decreasing its trust value.

These results show how the operating system adapts itself when unexpected events change the known environment. In this experiment the client agent changes its trust in the different nodes of the network, changing consequently the number of requests done to each one of the nodes.

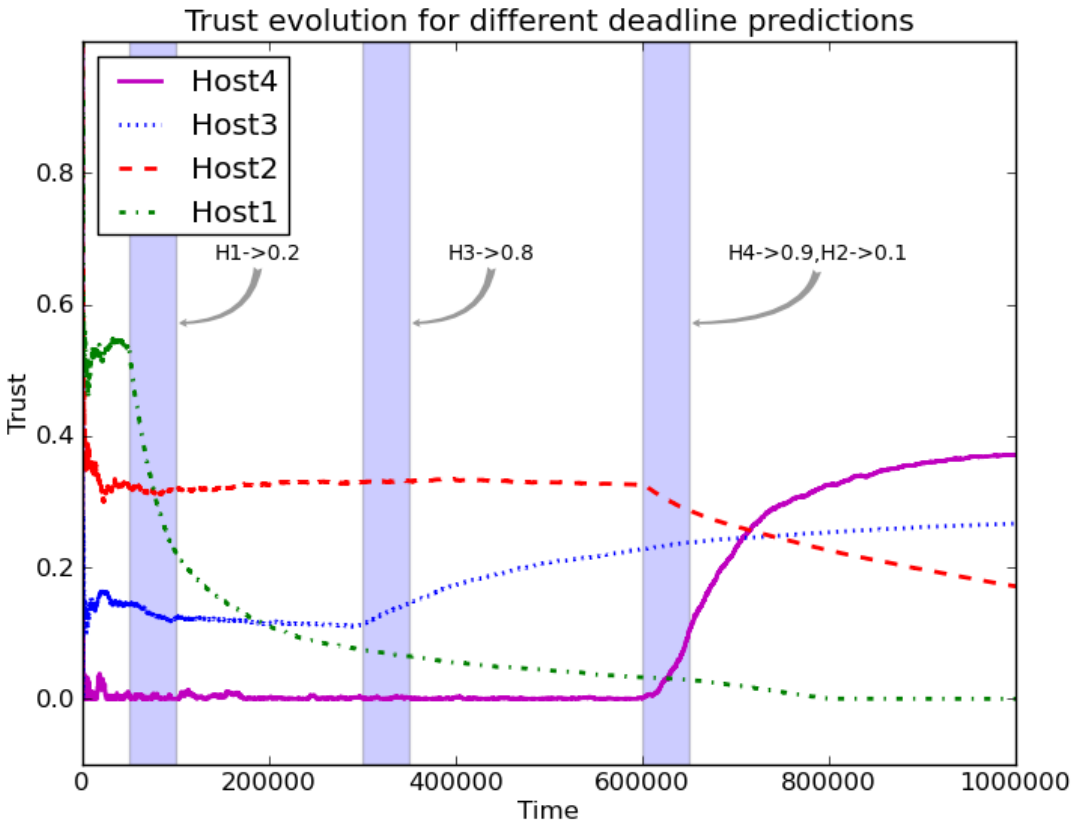

Figure 7.3: Test 3: Adaptive Operating System

Test 4: Accepted Plans Ratio

These tests have the purpose of showing the adaptability of the system in different situations of the workload and user preferences. To simulate this, during the tests two variables have been parametrized. These variables are established by the Deliberation Engine to manage the quality and quantity of plans that are accepted to be executed.

This two variables are the maximum slack time that the deliberation engine gives to run the plan and the minimum quality that the deliberation engine requires (this quality value is the SCSP provided by the Commitment Manager). Also, the number of agents has been continuously increased at each iteration to get a bigger number of active goals, which increases the system's workload.

The maximum slack time (**MaxTime**) represents the amount of time that the deliberation engine is prepared to give for the execution of a plan. Any plan whose time execution prediction exceeds this parameter will be excluded. The minimum quality accepted (**MinQ**) by the deliberation engine represents the lower limit that is accepted among all the plans proposed by the On-Line Planner. The quality of every plan is obtained by the Commitment Manager and represents the probability of success in the execution of the plan (SCSP).

Both parameters, the MaxTime and the MinQ, are dynamically adjusted by the deliberation engine to adapt itself to the current requirements of the system. It will always try to minimize the execution time and maximize the quality obtained. But in these tests we are going to study how both parameters modify the ratio of *accepted plans*, because we do not want a system that is so strict that does not accept new jobs, since its main purpose is to run plans. This is the reason why these parameters (MaxTime and MinQ) are dynamic, to adapt them to the current demanded workload of the system.

The relationship between the time estimated by the deliberation engine and the percentage of plans *accepted and executed before their deadline* is studied. Different qualities have been analyzed with the purpose of seeing how this parameter affects the result.

The first test set was executed for a static number of agents. These agents provide services and express static goals. We have run the simulation with different values of *MaxTime* and *MinQ* for the deliberation engine. In Figure 7.4 the *accepted and achieved* plan ratio for different qualities is shown. For each MaxTime value (from 1 to 7 time units), that represent the maximum time to execute the services, three bars represent the minimum qualities (20%, 50% and 90%) accepted. The figure shows that the trend in the percentage of accepted plans increases as time increases. When the MaxTime parameter is increased the set of plans that meet the specified time is bigger. Similarly, the proportion of achieved plans is always greater for high quality values. This is because the number of accepted plans is lower but with higher quality, so their success chances are higher.

Figure 7.5 allows us to validate this last result. This figure shows the percentage of plans

accepted and executed before their deadline, but this case comparing it to quality. For each quality value (from 30% to 90%) three maximum times (1, 4 and 7 time units) have been analyzed. Here the trend is to decrease the accepted ratio when the quality is increased. This is because the deliberation engine is more strict with the plans it accepts for higher values of required quality. In this figure we can appreciate that the quality parameter is more relevant than the time parameter when the system workload is not critical. We can also appreciate that for high values of quality (i.e. 90%), the accepted plans ratio is very low, but *almost* all of them were successfully achieved, regardless of the allowed execution time.

Test 5: Stress Test

To see how the system workload affects to the number of accepted plans a stress test has been performed. The method used in this test set to stress the system has been putting a high load by increasing the number of agents. The more agents in the system, more activated goals, which results in more running plans. Figure 7.6 shows the variation of the number of accepted plans when the number of agents is increased, parametrizing the maximum time (MaxTime) and the

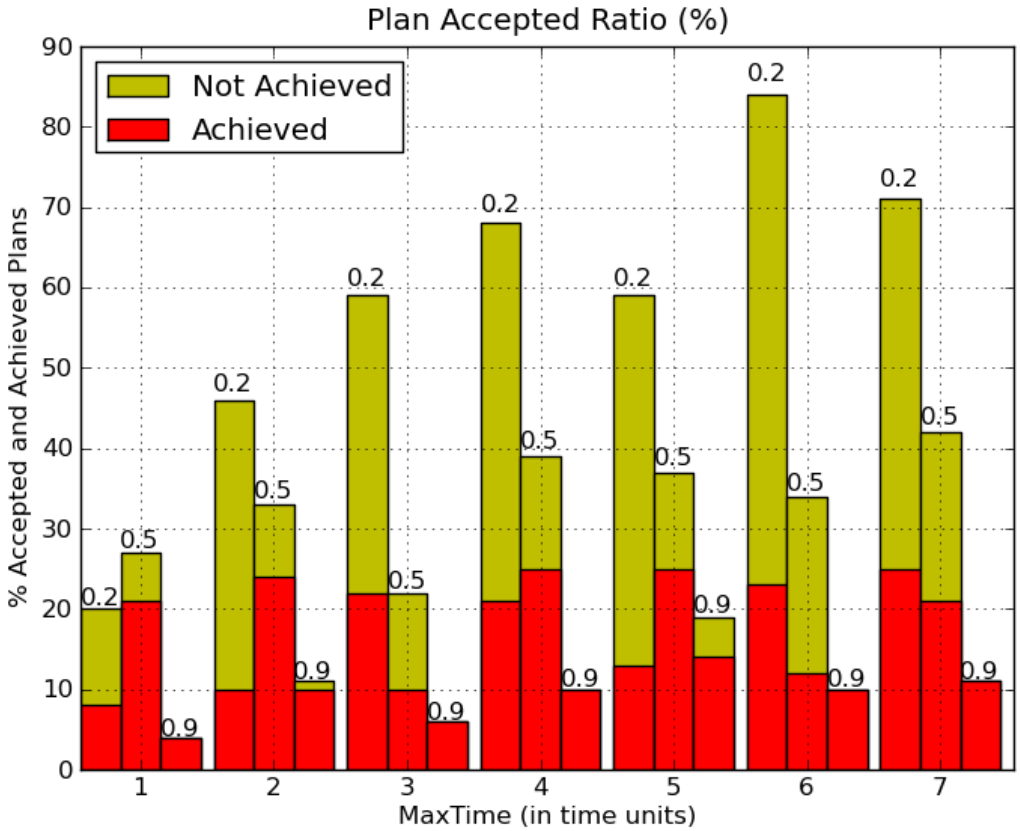

Figure 7.4: Test 4: Plan Accepted Ratio by time

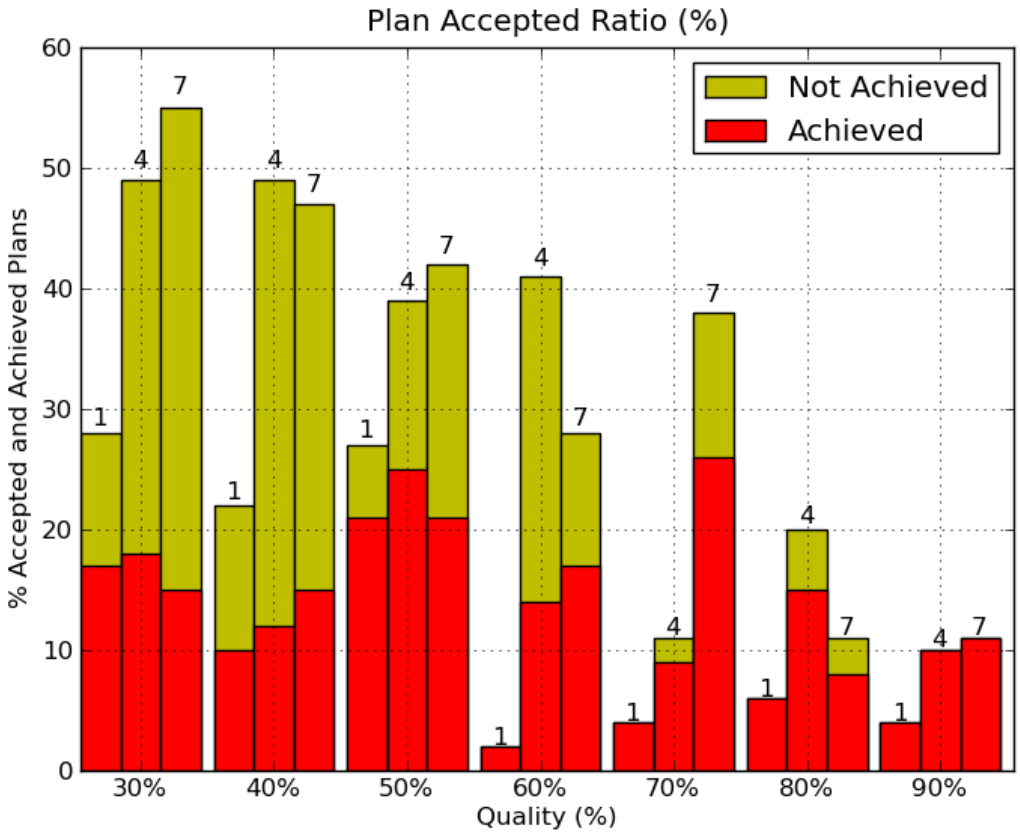

Figure 7.5: Test 4: Plan Accepted Ratio by quality

minimum quality (MinQ). In this work we are showing the evolution of the accepted plans ratio when the number of agents is increased for sets of 20, 30, 50, 70, 90 and 100 agents. In these figures it is appreciated that the behavior is what is expected and desirable. This means that the accepted ratio is decreased when the workload is increased. Comparing each graph, shows a progressive decrease in the number of plans accepted when the number of agents is larger and the system is more stressed. The higher ratios occur for the worst time and quality parameters. Therefore is the responsibility of the deliberation engine to balance the parameters of time and quality to maximize the utilization of the system. Meanwhile the execution time parameter will be minimized and the quality parameter will be maximized to improve the results of each executed plan.

To analyze with more detail how these parameters modify the goodness of selected plans we have designed another test. This test (Figure 7.7) presents the relationship between the maximum time to achieve the plan, the minimum requested quality and how this affects to the number of plans that finish before their deadline. For lower time values the percentage of achieved plans is higher because the number of accepted plans is very low. This is because there is not enough time to execute the most of the compiled plans. We can also see that for

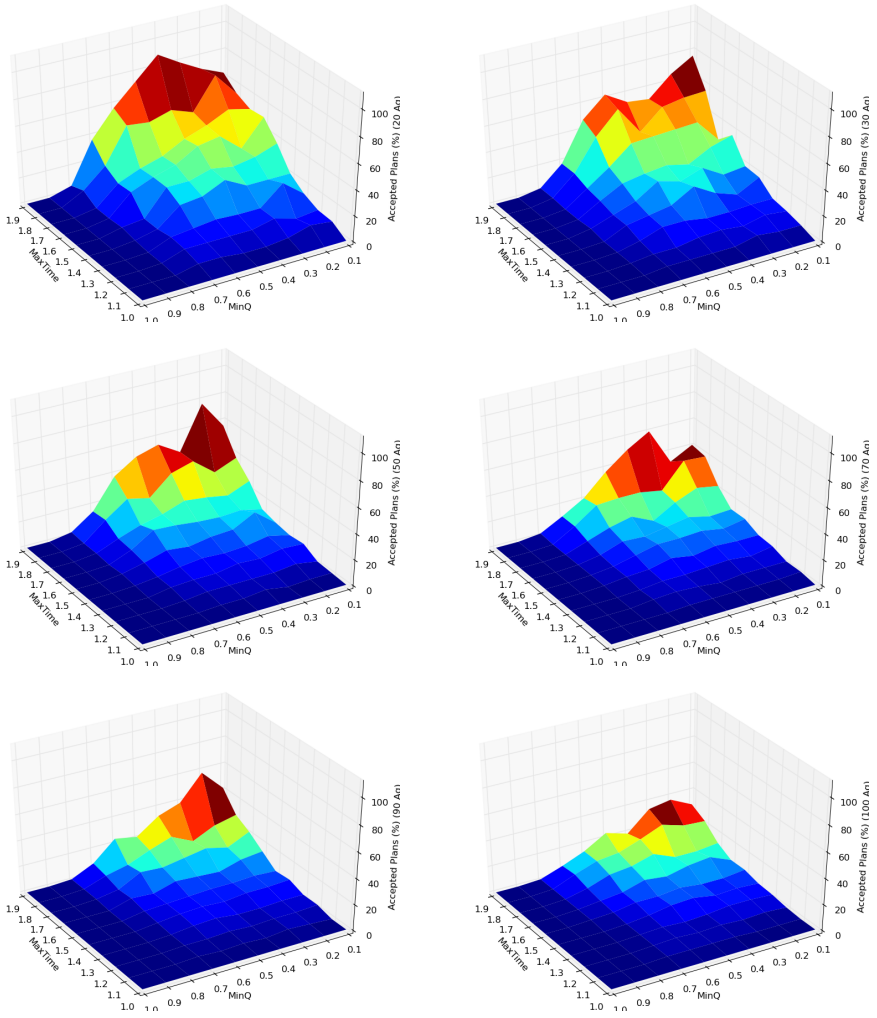

Figure 7.6: Test 5: Plan Accepted Ratio by time (20 to 100 agents)

high values of quality the number of plans that finish in time is always high. Therefore, it is observed the correct behavior of the deliberation engine, which is able to predict the execution of the services to fulfill the required quality parameters and select *only* the plans that will fulfill their commitments.

Test 6: Security Commitments

An specific test has been done to check the relevance of the security commitments. This test is intended to check how important is selecting a provider that applies the proper security policies to transactions. It is also very important to trust your provider, so your providers confidence level must be dynamic and well calculated. For this test we have prepared a scenario where a client agent activates a goal that is fulfilled by a set of agents, each one of them with a different security level. This test is run with three different OS configurations: an OS

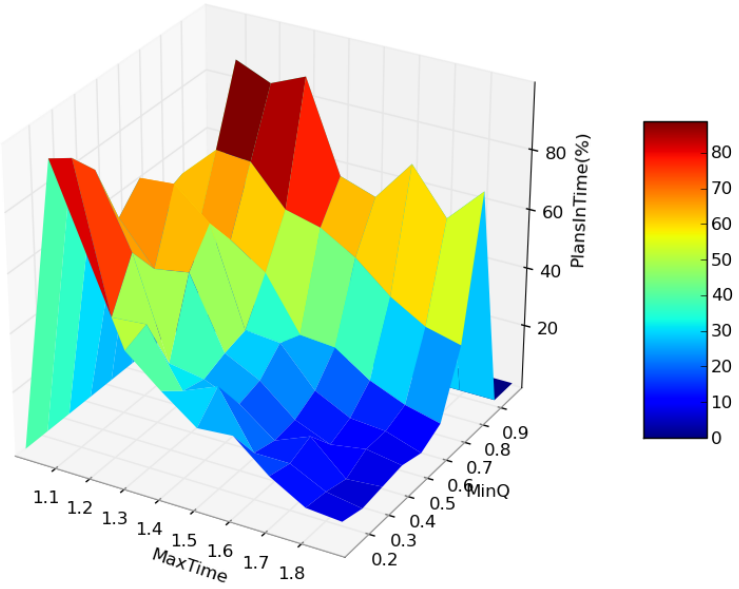

Figure 7.7: Test 5: Percentage of plans executed in time

without Commitment Manager that does not apply any kind of commitment when selecting a service (the selection is random); an OS with a CM that only performs temporal commitments (t-commitments), however, no security commitments are done. That is, it selects the service with a lower execution time and a higher success rate; Finally, a third OS with a CM that performs t-commitments and s-commitments (security commitments). In this case security and temporal parameters are taken into account. To test the robustness of the system we have introduced a sniffer in the simulator that spies the communications between all the agents, trying to steal passwords that are exchanged when invoking services. If the security level of the service is high, the password is encrypted with a strong algorithm. If the security level is very high the access control uses one-time passwords, this makes very difficult to steal a useful password. When the security level is low the password is very easy to be decrypted. Finally, when the security level is very low the password is not encrypted. As shown in Figure 7.8, the percentage of stolen passwords for a system with s-commitments is very low. As long as the services that are selected for execution are those that have the highest security levels, passwords are rarely stolen. It improves with time as the confidence is better calculated, based on experience. Nevertheless, the percentage of stolen passwords is very similar when the CM performs only t-commitments and when there is no CM in the OS. This is because the selection of services based on security parameters is completely random. We can even see that, in some cases, the percentage of stolen passwords is higher with t-commitments than without commitments. This is because security algorithms are usually big time consumers, so the selected services tend to be those with the lowest security level.

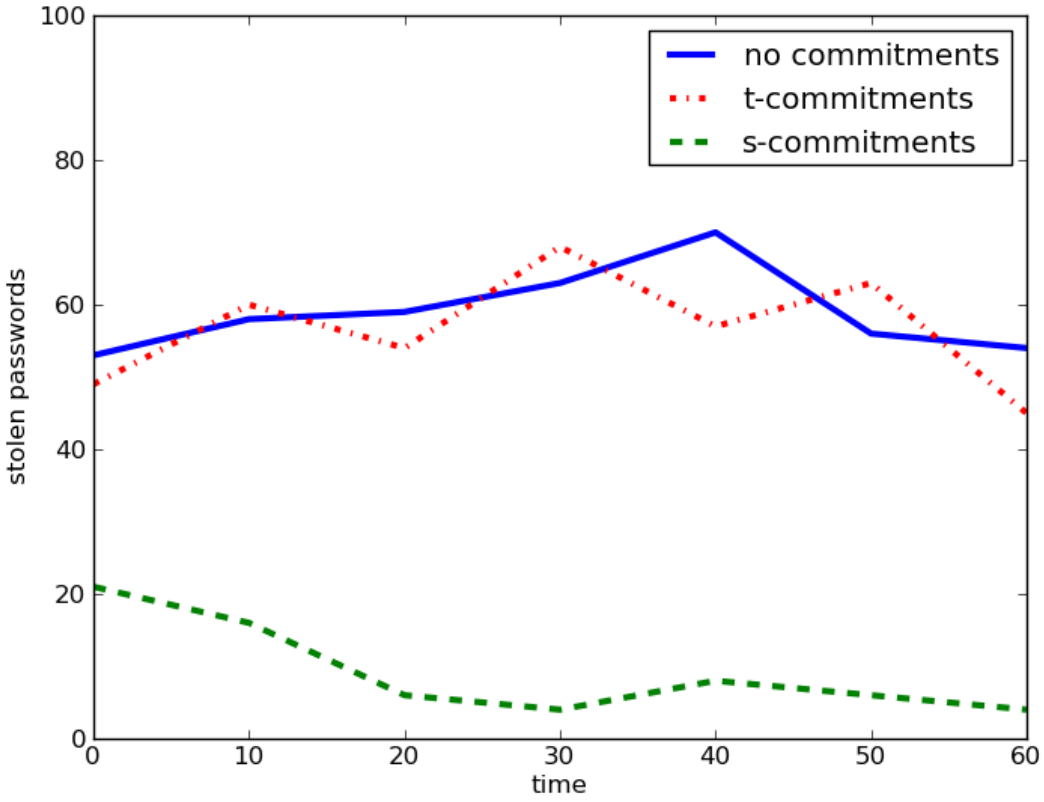

Figure 7.8: Test 6: Security test

7.2.2 On-line Planner

The On-line Planner is the component that allows to compose plans that fulfill the agents goals. This planner uses a time-bounded case based planner (TB-CBP) to create the requested plans by reasoning about past cases. Using a planner to achieve the active goals provides agents an interesting feature: plan repairing, which makes the operating system fault tolerant. This is the functionality that we are going to check with these tests. How the system increases its fault tolerance in unreliable environments and how this affects to the rate of completed goals.

Test 7: Fault-tolerant operating system

This test has as objective to check how the operating system is able to complete the goals that are active, even if a service execution fails and the plan becomes unuseful. In order to do that, the simulator can be parametrized with an **error probability**, which defines the probability of a service to fail. This test is defined with the following elements:

- Only 1 host is created, there is no need of distributing the test in this case.
- There are 50 registered agents, each of which has 50 goals to activate.
- There are 300 services equally distributed throughout all agents.

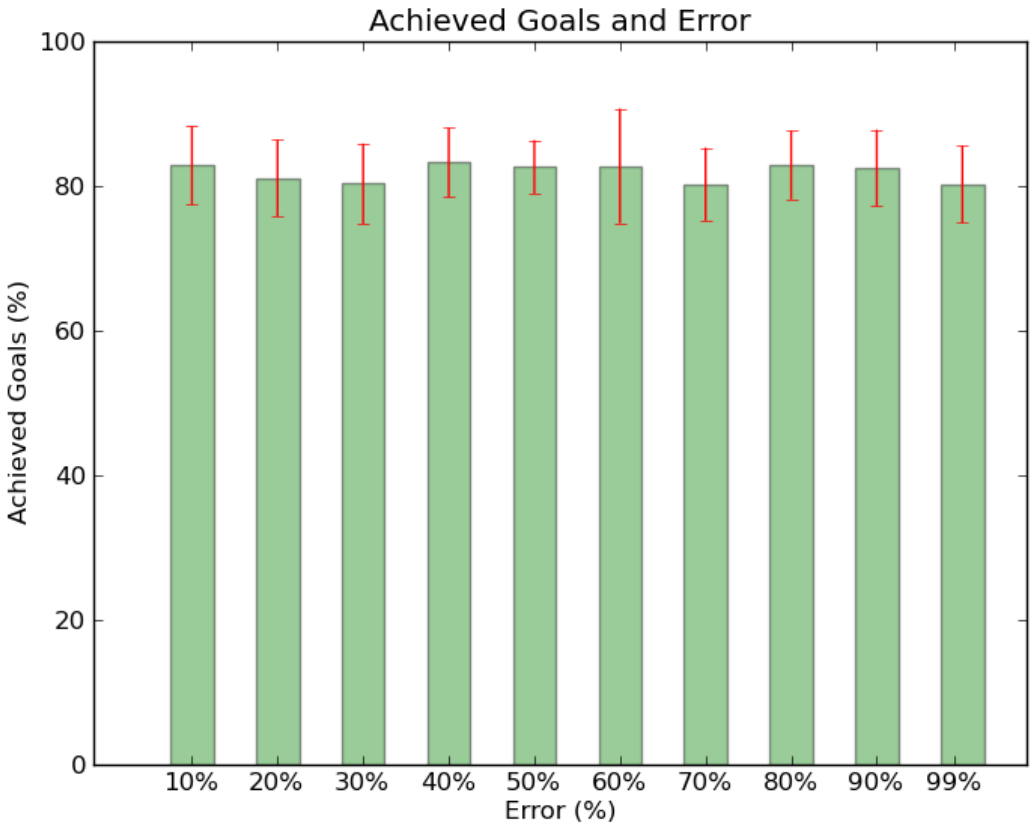

Figure 7.9: Test 7: Fault-tolerant operating system

In this experiment all the goals, services and agents knowledge items are randomly generated. There is only one parameter that will be changed during the test, the error probability. This parameter will be changed from 10% to 99% in steps of 10. Figure 7.9 shows the results of this experiment. The X axis shows the error probability assigned to the services. The Y axis shows the percentage of success for all the goals activated. Note that the percentage of success is not 100%, since the data is randomly generated and there is not always a path from the preconditions to the goals. What is shown in Figure 7.9 is that the percentage of success of the goals is constant, despite the error probability that the services have. These results are so relevant because they conclude that the proposed operating system is highly fault-tolerant.

Test 8: Trust evolution and multiple errors

This experiment shows how the combination of previous experiments can affect to the trust of the nodes of the system. This experiment combines the error probability of the running services and the accuracy of the response time calculated by the Commitment Manager.

This experiment has the following scenario:

- There are 5 agents registered in the system: 1 client agent and 4 provider agents.
- The network is composed by 5 nodes: Each agent is hosted in one of the five nodes.
- The client agent has the necessary knowledge to run the service (P) and activates the goal G which is the same that the two services postconditions (G=Q).
- All services have the same behavior **but** the nodes that host the service have different accuracies to calculate the response time and different error probabilities for the services:
  - *Host1* has an accuracy of 90% calculating the deadline prediction and a service error probability of 10%.
  - *Host2* has an accuracy of 90% calculating the deadline prediction and a service error probability of 90%.
  - *Host3* has an accuracy of 10% calculating the deadline prediction and a service error probability of 10%.
  - *Host4* has an accuracy of 10% calculating the deadline prediction and a service error probability of 90%.

Figure 7.10 shows the results of this test. These results show that there is no relevant difference between nodes with different configurations. The client does not discriminate on the basis of the situation that generated an error (a bad deadline prediction or a service error). What the client can see is that the service has not been provided conveniently (maybe its execution failed or was not provided in time), so the provider is punished. The figure shows how Host 1, which is the most reliable overall, has the higher number of requests. On the other hand, Host 4 is probabilistically the less reliable node, thus it has the lower number of requests.

7.3 Test 9: Distributed Computing Performance Tests

Finally, a performance test has been done to check how this computing paradigm can improve the execution of goals. The Operating System implementing the Distributed Goal-Oriented Computing paradigm has a great impact in the performance of the system. Having an Operating System that not only helps agents to perform their goals, but also searches services to compose the plans on other hosts, largely increases the concurrence of the distributed system.

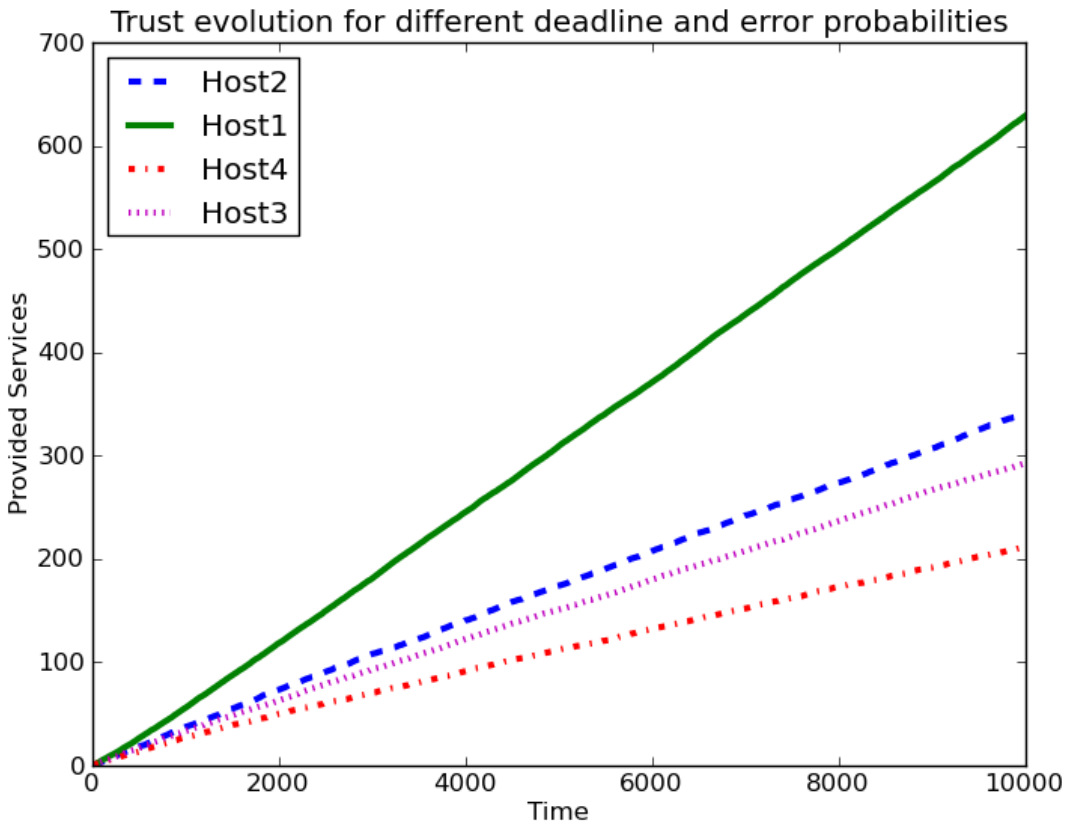

Figure 7.10: Test 8: Trust evolution and multiple errors

Test 6 (Figure 7.11) shows how increasing the number of nodes that offer services (X axis) decreases the mean time for achieving goals (Y axis). This behavior is highly significant as nodes are added to the network. To run this test, a large enough set of goals has been activated at every experiment. Each experiment has a different number of nodes (1 to 50) and the agents are distributed equally around the nodes. This way we can perform the activated goals with a higher degree of concurrency and, accordingly, with less time.

We can see in this experiment how important is to increase the amount of nodes in the network. Specifically, the first ten nodes contribute with a great impact to decrease the time needed to fulfill the activated goals. Experiments with 10 or more nodes do not have as much impact as the first experiments, but are always decreasing. In conclusion, the ability of distributing the execution in an automatic and transparent way increases reasonably the performance of the system.

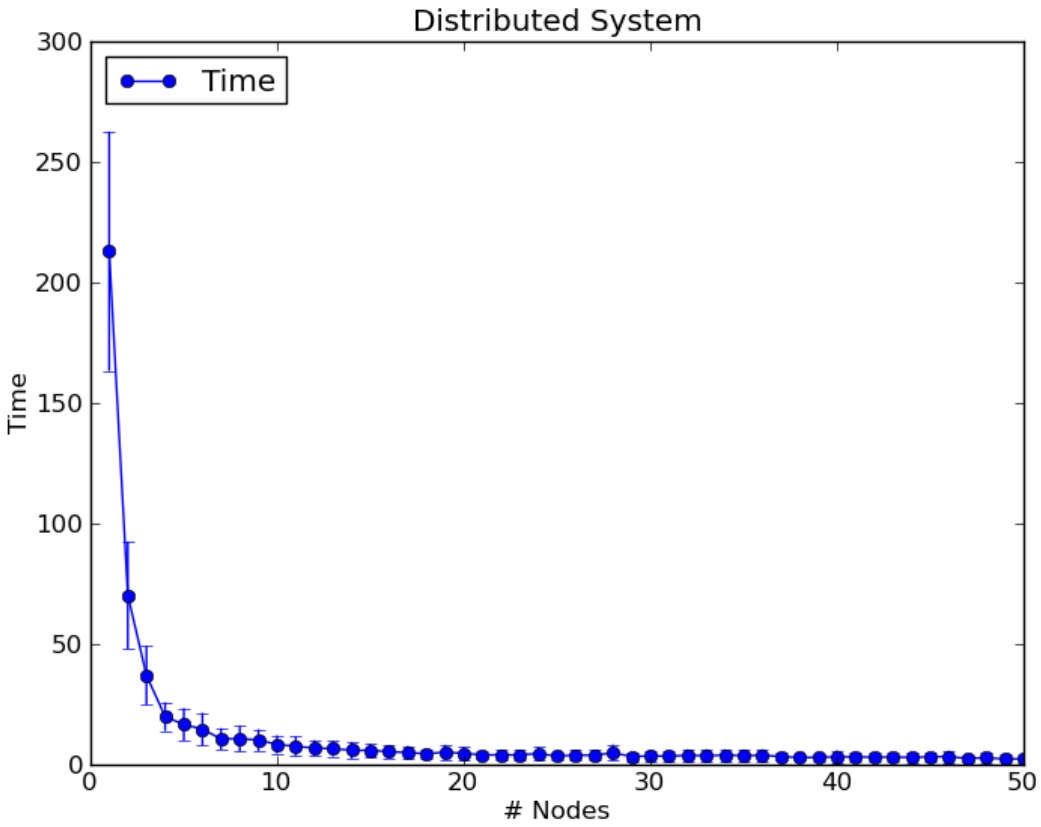

Figure 7.11: Test 9: Distributed Computing

8

Conclusions

We have presented in this work a Distributed Goal-Oriented Computing paradigm based on the automatic composition of plans. These plans are formed by distributed services provided by agents. Agents are also the entities who express their own goals and try to fulfill them by means of the plans. To implement this paradigm an execution module for a Goal-oriented Operating System has been designed. The OS purpose is to help agents to achieve their goals by means of a service-oriented approach.

The OS execution module is divided in two components which are in charge of performing this paradigm. The Deliberation Engine obtains the services needed to achieve the agents goals and stores them in a case base to reason about past cases. This component also takes time and trust constraints into account. This is done either to obtain a result before a deadline, or just to improve the quality of the result.

The case base introduced in the Deliberation Engine uses a Temporal Bounded CBP algorithm to obtain plans that guarantee their execution before a deadline (using the temporal commitments given by the Commitment Manager) and that have a high success degree (reasoning about the trust stored in the case base). This TB-CBP has allowed us to compose on-line plans that give solutions to the goals of the agents following temporal constraints. To guarantee that the agents execute their services before their deadline, the Deliberation Engine provides a Commitment Manager which is in charge of analyzing the workload and establishing a temporal commitment between the agents and the Deliberation Engine.

The results of this work have shown how the Operating System adapts itself to the environment where it is deployed. It selects the providers which offer better temporal commitments and trust values and distributes the workload around these providers proportionally. Also,

having an On-line Planner in the OS makes it more reliable and fault-tolerant. This is because, even if a service execution fails, the OS will look for a new plan transparently and without user interaction. In fact, the user is not even aware of this.

This proposal opens the possibility of designing service-based operating systems directed by goals using this paradigm. These OS can be extended continuously with new services and plans driven by the user needs. These plans are added by means of the services offered by other users and by their composition, thanks to the new goals defined by the users. The OS architecture defined in this work allows us to use this computing paradigm, since there are some capabilities that only the OS can provide (like soft real-time constraints and temporal commitments).

Bibliography

[1] M Wooldridge and I Dickinson. Agents are not (just) web services: considering bdi agents and web services. *Proc. of SOCABE'2005*, Jan 2005.

[2] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *In Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI98)*, pages 43–52. Morgan Kaufmann, 1998.

[3] Dominic Greenwood and Monique Calisti. An automatic, bi-directional service integration gateway. In *Proc. Workshop on Web Services and Agent-Based Engineering (WSABE'2004)*, 2004.

[4] E. Sirin and B. Parsia. Planning for semantic web services. In *Proc. Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, 2004.

[5] Javier Palanca, Vicente Julian, and Ana García-Fornes. A goal-oriented execution module based on agents. In *44th Hawaiian International Conference on System Sciences*, page 277, 2011.

[6] R Pike, D Presotto, K Thompson, and H Trickey. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221, Jan 1995.

[7] A Montz, D Mosberger, S O'Malley, L Peterson, and et al. Scout: A communications-oriented operating system. *Hot OS*, Jan 1995.

[8] R Rashid, D Julin, D Orr, R Sanzi, R Baron, A Forin, D Golub, and M Jones. Mach: a system software kernel. *COMPCON Spring '89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers*, pages 176—178, 1989.

[9] Galen C Hunt, James R Larus, D Tarditi, and T Wobber. Broad New OS Research: Challenges and Opportunities. *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Jan 2005.

[10] Galen C Hunt, James R Larus, M Abadi, Mark Aiken, P Barham, and et al. An overview of the singularity project. *MSR-TR-2005-135*, Jan 2005.

[11] Jorrit N Herder, H Bos, B Gras, P Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *Operating System Review*, Jan 2006.

[12] T Cortes, C Franke, Y Jégou, and T Kielmann. Xtreemos: a vision for a grid operating system. *White paper*, Jan 2008.

[13] I Johnson, B Matthews, and C Morin. Xtreemos: Towards a grid operating system with virtual organisation support. *UK eScience All Hands Meeting*, Jan 2007.

[14] A Rao and M Georgeff. BDI agents: From theory to practice. *Proceedings of the first international conference on multi-agent systems (ICMAS95)*, pages 312—319, Jan 1995.

[15] L de Silva and L Padgham. Planning as needed in BDI systems. *International Conference on Automated Planning and Scheduling*, 2005.

[16] Luca Spalzzi. A survey on case-based planning. *Artif. Intell. Rev.*, 16(1):3–36, 2001.

[17] M. Navarro, S. Heras, V. Julian, and V. Botti. Incorporating Temporal-Bounded CBR techniques in Real-Time Agents. *Expert Systems with Applications*, 38(3):2783–2796, 2011.

[18] Elena Del Val, Marti Navarro, Vicente Julian, and Miguel Rebollo. Ensuring time in service composition. In *SERVICES 2009. 2009 IEEE Congress on Services*, volume 1, pages 376–383. IEEE Computer Society, 2009.

[19] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance*. O'Reilly Media, Inc., 2009.

[20] M. Zviran and W.J. Haga. A comparison of password techniques for multilevel authentication mechanisms. *The Computer Journal*, 36(3):227, 1993.

[21] J.S. Balasubramaniyan, J.O. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni. An architecture for intrusion detection using autonomous agents. In *Computer Security Applications Conference, 1998, Proceedings., 14th Annual*, pages 13–24. IEEE, 1998.

[22] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[23] D RITCHIE and K Thompson. The unix time-sharing system. *Communications of the ACM*, Jan 1973.

[24] Zero Configuration Networking: http://www.zeroconf.org.

[25] XMPP Pub-Sub: http://www.xmpp.org/extensions/xep-0060.html.

[26] Hermann Kopetz and Wilhelm Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(8):933 –940, aug. 1987.

[27] R.S. Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine learning: An artificial intelligence approach*, volume 1. Morgan Kaufmann, 1985.