

# AgentService: a framework to develop distributed multi-agent systems

## Christian Vecchiola\*

DIST - Department of Communication Computer and System Sciences,  
University of Genova, Genova, Italy  
E-mail: christian@dist.unige.it

\*Corresponding author

## Alberto Grosso

DIST - Department of Communication Computer and System Sciences,  
University of Genova, Genova, Italy  
E-mail: agrosso@dist.unige.it

## Antonio Boccalatte

DIST - Department of Communication Computer and System Sciences,  
University of Genova, Genova, Italy  
E-mail: nino@dist.unige.it

**Abstract:** This paper presents AgentService, a framework for developing multi-agent systems. The key features introduced with AgentService are a flexible agent model and a modular platform. Agents can perform multi-task activities as well as move among different platforms. The MAS is hosted in a modular environment so that the platform can be customized. Several tools come with the programming suite: a visual protocol designer, a set of language extensions integrated into a professional development environment to ease development, and monitoring tools to simplify the control over the MAS.

**Keywords:** agent-oriented software engineering; multi-agent system design and programming; distributed systems.

**Reference** to this paper should be made as follows: Vecchiola, C., Grosso, A., and Boccalatte, A. (2006) 'AgentService: a framework to develop distributed multi-agent systems.', *Int. J. Agent-Oriented Software Engineering*, Vol. 1, No. x, pp. xx-xx.

**Biographical notes:** C. Vecchiola is pursuing his PhD in Computer Science from University of Genova. He received his MSc in Computer Science from the University of Genova in 2003. His current research interests include agent-oriented software engineering, programming language design and compiler implementation.

A. Grosso received his PhD in Computer Science from the University of Genova in 2006. He currently works at DIST and his primary research interests are in the field of multi-agent systems, multi-robot systems, and agent oriented software engineering.

A. Boccalatte is graduated in Electronic Engineering at the University of Genova (1976). In 1987 won a chair at the University of Genova, and actually is Professor of «Data Base Management Systems» at the Faculty of Engineering. He is author of more than 70 scientific papers presented at international congresses or published on international journals. He is involved in research activity on System Architecture and Artificial Intelligence.

---

## 1 INTRODUCTION

---

Rapidly evolving network and computer technology, coupled with the exponential growth of the services and

information available on the Internet, make people request software more and more distributed. The Internet or intranets are viewed as the global environment in which computations take place [Mattern, 2000]. Hence the

application of distributed system grows every day and cover from internet based multi-player games to distributed information system for enterprises.

From the developer point of view, the complexity of a distributed system requires a high level of abstraction. Agents and multi-agent systems (MAS) can provide such level of abstraction and seem to be a suitable solution for developing distributed system. The area of multi-agent systems addresses distributed tasks, it promises to cope more efficiently and elegantly with a dynamic, heterogeneous, and open environment which is characteristic for today's Internet.

Developing multi-agent systems is a complex task; it implies the implementation of concurrent software environment constituted by autonomous entities. These entities should rely on communication and directory services, offered by the hosting environment in order to easily interoperate. Agent programming frameworks commonly address these issues: an agent framework can support developers in the entire MAS life-cycle, providing them with tools for analysis, design, implementation, and deployment (Nwana et al., 1999; Poslad et al., 2000). Jade (Bellifemmine et al., 1999), developed by the TILab, is one of the most used and diffused.

The use of agent technology is increasingly explored by an expanding industry, and first commercial systems (e.g., Jack (Busetta et al., 1999), Aglets (Lange et al., 1998), Voyager (Glass, 1998), Concordia (Mitsubishi Electric ITA Horizon Systems Laboratory, 1997) are now available; but only a restricted number of agent applications are developed by industry. This can be due to the lack of some important features in agent frameworks. Scalability, quality of service, and robustness with respect to partial component failures are key issues that can make agent technology suitable for developing real enterprise systems. Hence multi-agent systems and in particular agent frameworks are still an open field for researchers.

This paper presents AgentService, a software environment designed to deliver distributed multi-agent systems. It provides software engineers with a library for agent development, a software environment hosting multi-agent systems, and a set of tools supporting developers at design-time and during the deployment. The core features of the framework are its innovative agent model and the agent platform. The agent model allows a real multi-tasking activity for agents and is flexible enough to implement different agent architectures. The agent platform provides a flexible and modular software environment able to evolve and adapt to different application contexts.

In this paper we give a brief overview of the framework, its services, and the features provided with them. The next section describes the features of AgentService by underlining the main design goals which drove the definition and the implementation of the framework. Section three analyzes the modular architecture of the agent platform describing the components that constitute it. In section four we introduce the agent model and describe its interactions with the hosting environment. Section five

discusses about the development of distributed multi-agent systems with AgentService: this task strongly relies on the inter-platform communication services and the mobility infrastructure. In section six we present the remaining components of the framework which support software developers and make their work more effective. Section seven provides a brief overview of the related works and presents a comparison with some of the most important software projects in the field. A summary of the features of AgentService, followed by the conclusions, terminates the dissertation.

---

## 2 AGENTSERVICE FRAMEWORK

---

Wikipedia defines the concept of software framework as: *"..a defined support structure in which another software project can be organized and developed"*. Moreover, *".. a framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project"* ([www.wikipedia.org](http://www.wikipedia.org)). Designing and developing a software solution based on agent technology is a complex activity since it requires different skills and involves many tasks. First, it is important to identify the different kinds of agents which better represent the problem domain; then we need to carefully detail the roles played by the agents and describe the interactions among them. Once we have obtained a model of the system we need to implement it by using a set of libraries allowing us to define agents and create the multi-agent system. High level abstractions and automated tools integrated within professional IDEs can help us developing multi-agent systems. Finally, after deploying the multi-agent system, we need to monitor its activity: in particular, graphical user interfaces can dramatically simplify the interaction with the MAS. All the tasks previously introduced are interrelated and concur in the development of a single project that is the multi-agent system; hence an agent programming framework providing all these features is the solution to our problem.

AgentService is a framework designed to develop and deploy agent-oriented applications; it provides a rich set of tools which supports users in designing, implementing, and deploying multi-agent systems. AgentService differs from the others agent programming frameworks for its architecture which allows designers and developers to tune it for different scenarios. In particular, the key points of our framework are the agent scheduler and the agent platform which can be easily customized and extended with new functionalities. The main features of the framework are the following:

- a flexible agent model through which different agent architectures can be implemented;
- a library which defines the core of the system and the basic services of the framework;
- a software environment that hosts multi-agent systems and controls their life-cycle;

- a set of programming language extensions simplifying the implementation of software agents;
- a collection of tools supporting users in designing, implementing multi-agent systems;
- complete support for ontology definition and development;
- automatic code generation for interaction protocols with ontology integration;
- a software infrastructure allowing agents to migrate among different instances of the AgentService platform;
- a set of support programs through which users can maintain and monitor multi-agent systems.

AgentService has been designed, and implemented, with the aim of delivering to end users a *flexible software infrastructure*. By the term flexible we mean the ability to evolve along with the changing requirements of the users, and to be easily extended or customized without a great effort. Such goal has been achieved by defining an extremely modular architecture: we can either easily change the base components of the system core or extend the whole system with additional services that seamlessly integrate with it.

Flexibility has been obtained without renouncing to simplicity: end users are supported with design modellers and high level abstractions either in the design or in the development phase. These elements can automate many decisions and simplify the tasks, but it is always possible to switch to a more detailed and powerful interaction. The ability to provide two different options in order to perform a task makes the framework suitable to different software development scenarios.

AgentService is based on the .NET technology and fully exploits the features introduced by such component oriented software platform. The core of the system relies on the Common Language Infrastructure (hereafter CLI) which makes the framework portable over different implementations of this specification like Mono, Rotor, and .NET.

The following sections describe the components of the frameworks and show how some key concept of the CLI have been smartly used to implement a flexible agent programming framework.

### 3 PLATFORM ARCHITECTURE

The agent platform along with the agent model represent the core of the framework: these two components identify the minimal subset required to implement a multi-agent system with AgentService. In particular, the platform constitutes the software environment which hosts the multi-agent system and provides it with all the required services. In other words, it acts as virtual machine for agents: it controls their instantiation, schedules their activities, gives them communication and localization services, and maintains their state. The platform also constitutes the main access

point to the MAS since it gives access to all the services offered by the multi-agent system.

The platform is designed according to the reference implementation proposed by FIPA ([www.fipa.org](http://www.fipa.org)) and by keeping modularity in mind. The adoption of this reference model provides a well known and accepted model among the community of researchers, while defining a modular architecture allows us to give a high degree of customization to the system. Modularity plays a key role in this case since the agent platform is one of the most important components of the framework and then is the one which mostly concurs in defining a flexible software infrastructure.

In this section we describe the three components constituting the software platform: the system core, the modules, and the service components. These three software layers cooperate and provide the multi-agent system and end users with all the features expected by a powerful and flexible software environment.

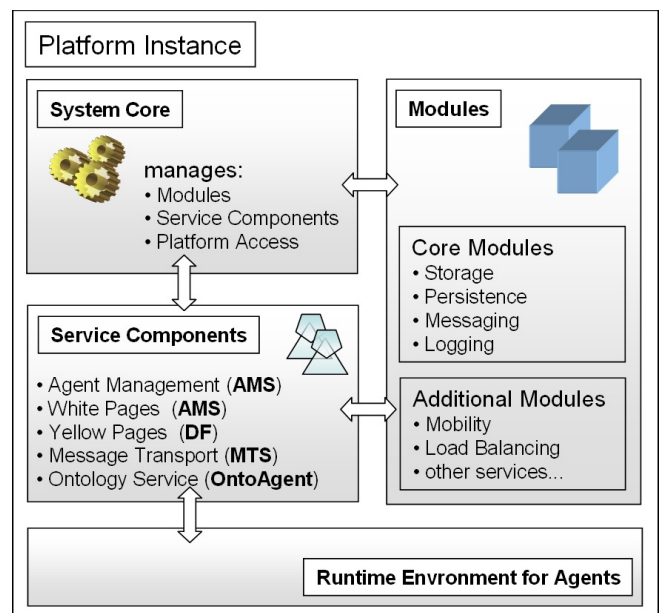


Figure 1 Platform instance internal structure

#### 3.1 System core

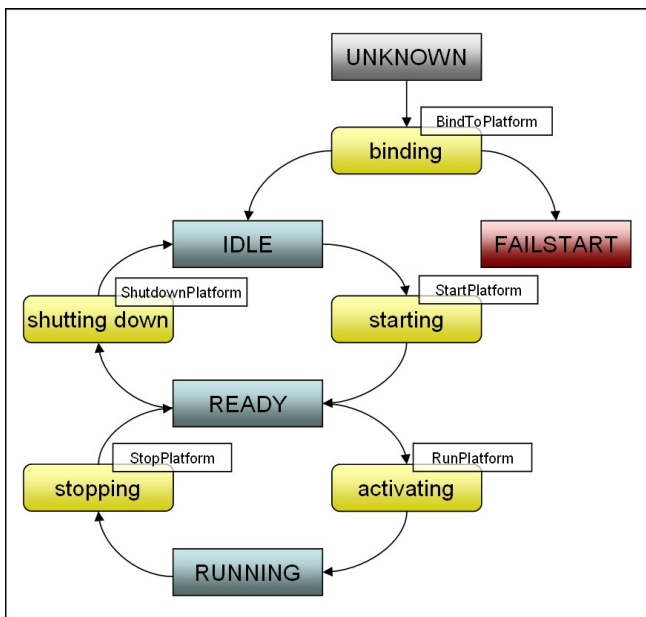
The system core provides the basic services to all the other software layers which compose the agent platform. It is responsible of setting-up a platform instance and controlling its activities. By the term *platform instance* we mean a single installation of the agent platform on a machine. The system core coordinates the activity of modules and service components and exposes their services to end users.

The system core is responsible of managing the life-cycle of a platform instance and customizes the setup of the platform by choosing:

- the type of agent-factory used: the agent factory is an implementation of the factory pattern (Gamma et al., 1995) and is responsible of creating the agent instances or resuming them from a persisted storage;

- the type of agent scheduler used: the agent scheduler is responsible of controlling the life cycle of the agent and the activities it performs. Agent schedulers control the number of threads and system resources assigned to agents as well as the type of scheduling algorithm used;
- which modules are activated during platform start-up: we will see in the next paragraph that modules are software components through which the platform can be extended and customized;
- the types of the service components that will be activated in the platform: service components are specific software agents suggested by FIPA that should be deployed on every FIPA compliant implementation. FIPA service components are responsible of agent communication, localization, execution and provide a standard to simplify interoperation among different agent platforms.

Moreover, a set of additional properties controlling other aspects of the platform, like remote access, can be added to the configuration settings. This set is open-ended: new properties can be added and used by other software layers in a transparent manner. For example, modules could require additional configuration settings or specific installation preconditions; by using the system properties a module can easily detect all what it needs to work properly.



**Figure 2** Platform instance life cycle

Figure 2 describes the life-cycle of every platform instance: the system core executes the state machine represented in the picture. By using a state machine as execution model, we are able to track more efficiently bugs and detect illegal conditions which could lead the platform instance to an inconsistent state. It is worth noticing that the platform goes through three main states during its life cycle: *idle*, *ready*, and *running*. When the platform process starts, the system core reads the configuration settings and takes the platform into the *idle* state. This state identifies a successful setup

and the system core has been properly configured to serve all other components. The next step of the process moves the platform into the *ready* state after which all the pending components are installed and modules activated. The *running* state activates all the service components and starts the activity of the multi-agent system hosted by the platform. We can notice that while the multi-agent system is running the control of the system is mainly delegated to modules and service components. In the next two paragraphs we will show how these two software layers exploit the system core to perform their tasks.

### 3.2 Modules

The platform accomplishes many of its activities by delegating them to modules. The term module refers to a software component that provides a certain type of service within the context of the AgentService platform. Modules are the common way to add features to the agent platform or to customize it in order to meet the requirements of a specific scenario. The platform system core basically starts the engine that loads and configures modules; service components are configured to use the installed modules in a transparent manner. Specific modules are required by the platform core in order to bootstrap an instance of the platform. Hence the concept of module is a fundamental part of the platform architecture. The system core provides the modules with a rich set of services: modules can inspect all the configuration data of the platform, register handlers for events, register new event sources, and query for installed components. Such architecture allows modules either to fully exploit the services offered by the system or to extend it in a simple way.

Modules are classified into two different classes: core modules and additional modules. Core modules are required by the platform core in order to create a working agent platform since they perform critical activities. The following sub-systems are implemented through core modules:

- storage management system: it handles the installation of the assemblies where agent templates, behaviour objects, and knowledge objects are defined. Every time a new agent type is deployed on the platform, all the dependent assemblies need to be placed in the storage. When an agent is instantiated, all the necessary information for its creation and its execution have to be found in the storage;
- persistence system: it is responsible to persist the state of the agents and to restore it after a system crash. Agents can be instantiated either as persistent or as not persistent. Persistent agents are registered with the system that saves a copy of their state according to the defined policies. After a system crash, the Agent Management System (AMS) queries the persistence system and recreates the agent community by restoring all the agents whose state was persisted;
- messaging system: it provides the agents with a communication channel based on message exchange. The messaging system is transparent to the agents that

use the Message Transport System (MTS) to communicate with peers. The MTS routes all the messages to the messaging system which is in charge of delivering. Messaging modules can provide two different communication models: simple message exchange and conversations that are connected communication channels between two agents;

- logging system: it provides the agents with logging facilities and allows different levels of logging. Even if the logging module is a core module it is not essential for the platform life-cycle and the activity of agents. Hence an instance of the platform can run even without this module.

The framework provides a standard implementation of these components relying only on the services of the Common Language Infrastructure: the storage uses the file system in order to store the assemblies; the persistence system stores the state of the agents by serializing the knowledge objects and the messaging system relies on *Remoting*<sup>1</sup>. Nonetheless if specific requirements need to be met a custom implementation for these modules can be provided at installation time or afterwards.

Additional features can be provided by installing and running additional modules. By defining and integrating additional modules we can provide more complex services which rely on the platform infrastructure and the core modules. Additional modules are loaded after core modules and can be notified of events that occur during the platform life cycle: by registering with the platform events modules are able to interact with the platform while performing their tasks. The mobility infrastructure (see section 5) has been implemented by using an additional module and the integration with the system has only required a specific implementation of the agent factory. Such component is dynamically configured and created by the system core during the platform setup.

The plug-in architecture provided with modules is an efficient technique to obtain a flexible hosting environment that can seamlessly evolve according to the changing requirements of multi-agent systems: module installation, configuration, and execution are transparent to the agent activity. Agents or platform administrators can exploit the new features as they are installed into the system. As a proof of concept we can notice that during the development of the framework we used three different kinds of messaging modules: one using Microsoft Messaging Queue as storage for messages, the current one which provides a complete CLI compliant implementation, and another which allows inter-platform communication by using web services. The use of different messaging modules has been completely transparent to rest of the system.

### 3.3 Service components

The platform constitutes the runtime environment for agents and it has to provide services for make them work properly. FIPA defines a set of basic features which each agent platform should have and that should be offered through dedicated service agents. AgentService is compliant to the FIPA abstract specifications and offers the following services:

- Agent Management System (AMS): it is the supervisor and the controller of the MAS. It provides facilities for controlling the life-cycle of the agents and it offers the white pages service to the community.
- Directory Facilitator (DF): it offers directory services (i.e. yellow pages service) to the agents of the platform and its external clients. The DF maintains a registry of all services offered by the agents and can be queried to retrieve all the agents exposing a specific capability. DFs deployed on different platforms can join together into a federation.
- Message Transport System (MTS): it handles the messaging system. All the messages, exchanged within the platform, are routed by the MTS that also forwards messages to other instances of AgentService. It makes agents interoperate.

In addition to these standard features the platform provides an ontology service in order to ensure that the agents ascribe the same meaning to the symbols used within the message contents. The ontology service, as indicated by FIPA ([www.fipa.org](http://www.fipa.org)), is offered by a dedicated agent who manages all the ontology adopted by the agent community and takes trace of the capability of each single agent.

---

## 4 AGENT MODELING AND DEPLOYMENT

---

The models and software artefacts used to define and execute software agents constitute the second fundamental component of the framework. These elements define the agent model adopted within AgentService. In this section we point out the definition of software agent which the model is based on, we present the features of such model and describe how the model is translated into a software artefact managed by the platform. We also cover in detail the scheduling system used to implement the multi-behavioural activity of software agents. Finally we discuss the agent deployment process and how software agents are integrated with the MAS and perform their activities.

### 4.1 Agent Model

In order to introduce the agent model, we define the abstraction which better represent the concept of software agent within the framework: an agent is an *autonomous software entity whose activity is constituted by a set of concurrent tasks and whose state is defined by a set of*

---

<sup>1</sup> Remoting is a software infrastructure provided by the CLI which enables instances to call methods on objects distributed over a network.

*shared objects*. Concurrent tasks are referred as *behaviour objects* while the term *knowledge object* is used to identify the components of the agent state.

If we examine the previous definition, we can notice that agents are represented as autonomous entities; this could suggest that software agents control their life-cycle. In AgentService this is partially true, since the agent platform which provides basic services to software agents implicitly controls the activity of agents while the AMS, as suggested by FIPA, explicitly manages the agents hosted in the platform. Hence, autonomy refers to the fact that software agents can be conceived and then designed and implemented as simple programs which have their own execution stack; other software components (except the platform through the AMS) can not control the activity of agents. The autonomy model described here is common to almost all the others frameworks (Belifemmine et al., 1999; Nwana et al., 1999; Poslad et al., 2000) that want to be compliant with the FIPA reference model (www.fipa.org). Such model does not tell anything about the implementation of agent activities, which are only said to be concurrent; this gives more freedom to framework designers. We decided to keep separate the state of the agent from its activities in order to easily manage, persist, and move the knowledge acquired by it; for these reasons we introduced the concepts of behaviour objects and knowledge objects.

Behaviour objects contain all the agent computational logic; they are used to model capabilities and services offered by the agents to the community. The overall activity of an agent is defined by the concurrent execution of behaviour objects and it is responsibility of the agent runtime system to schedule behaviour instances in a concurrent fashion. The key concept behind behaviour objects is that they have to be considered like little simple programs having their own entry point. Developers implement behaviour objects by extending a special class of the framework, by overriding the method that constitutes the entry point, and by adding any other features they need.

Knowledge objects define the knowledge base of the agent. They are shared between behaviour objects which, by modifying their data, can change the state of the agent. Knowledge objects are data structures defined like the Pascal *record* or the C *struct*: any CLI serializable type can be a field of a knowledge object. The agent runtime system ensures exclusive access to the knowledge objects and to their fields, in addition it takes care about all the concurrency issues. The framework offers a reliable service for the agent life-cycle and knowledge objects are handled with different levels of persistence. The agent state can be saved upon each knowledge object change or on a specific user request. Persistence is transparent to programmers that just have to define the fields that compose the knowledge and to specify the persistence policy.

The distinction between activities and data allows a clear decomposition of the agent definition providing the developer with a simple customization of the agent template. New behaviour and knowledge types can be

programmed, or they can be chosen from ready-to-use libraries.

The model described above (Figure 3) defines an agent either statically or dynamically, but two different views are provided and the framework makes a clear separation between the agent definition and its instance scheduled at runtime. The first one is called agent template and it extends the *AgentTemplate* class, while the other one is the “real agent” and it is an instance of the *Agent* class. The relationship between the two entities is similar to the relationship between the class type and its instances: each agent instance acts according to the relevant agent template. It is worth noting that agent instances do not strictly have the type of their template, but complete different classes are instantiated at runtime. By means of reflection, the agent template is inspected and the agent instance is created according to that model. The running instance of the agent has all the necessary stubs to exploit the platform services and to interoperate with other agents, while the agent template and the behaviour objects have only access to proxies of these services.

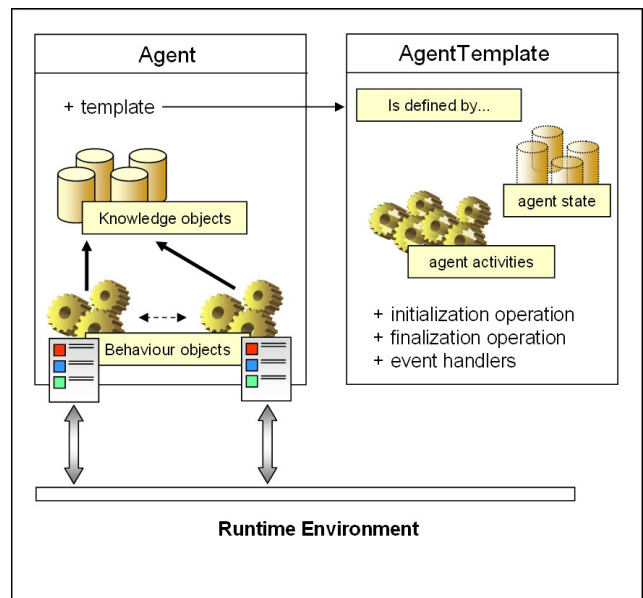


Figure 3 Agent model

The separation between the agent definition and its running instance gives many benefits. Common implementations of the agent model use an abstract agent class specialized by inheritance in order to define new agent types. The types used at runtime are the same defined by the agent designers: such a model leads to high coupling and implicitly binds the definition of software agents to their runtime instances. The model proposed with AgentService completely isolates the definition of agents from the corresponding runtime instances: user implementing software agents are exposed only to APIs relevant to the agent definition while the framework internals are maintained into the *Agent* class. The new templates are used like prototypes for agent instances and the runtime environment uses them to create agent instances in a

transparent manner. Hence the runtime support can be easily changed without breaking the contract defined by the *AgentTemplate* class.

## 4.2 Runtime execution

The agent model represents the essence of a software agent from a design and an implementation point of view. In order to have a complete picture of what software agents really are, we need to investigate the behaviour at runtime. We mostly concentrate on the agent deployment process and on the execution of its multi-behavioural activity. These two tasks heavily rely on the concepts defined with the CLI that are *assembly packaging* and *Application Domain*: we show how these features have been exploited in AgentService in order to deliver a robust and flexible agent runtime environment.

Agent instantiation and execution takes place after the corresponding template is registered with the platform. Template registration is automatically performed by the storage module when a new assembly is uploaded: by means of the Reflection API the assembly is inspected and all the types defining agent templates, behaviour and knowledge objects are registered. The platform can be configured in order to accept only signed assemblies which univocally identify the types they contain. After the registration process, templates become available to create new agent instances. The creation of an agent instance involves the retrieval of the corresponding template, the actual creation of the object instance, and its further customization according to the agent template. In particular the agent instance performs the following operations:

- creates all the required knowledge objects and behaviour objects declared in the template;
- connects the behaviour instances with the selected knowledge objects;
- binds the agents to the runtime that makes available to the behaviour objects all the services offered by the platform;
- initializes the agent instance.

At the end of these operations, the agent instance is ready to run and the platform configures the scheduling engine which starts the agent activity. This process is completely transparent to end users who, while defining new templates, just have to compose them with behaviour and knowledge instances and provide some initialization code. The overall process is delegated to the runtime environment and to the scheduling engine.

## 4.3 Agent Scheduling

Agents scheduling is a critical issue for every agent programming framework, since it dramatically influences the overall system performance. Agents scheduling mainly concerns the use of threads to execute the concurrent activities of agents. Threads are limited resources, especially in case of virtual execution environments (like

Java and CLI) where often the abstraction of logical threads is used in place of physical threads. A common requirement for software agents is the ability to perform multiple tasks in the same time; hence, agent programming frameworks have necessarily to cope with the problem of defining and implementing a scheduling engine. There are mainly two scheduling models adopted as reference model by the community of agent researchers:

- agent-process model: for each agent a new process is created and the multiple activities characterizing the agent have a proper execution thread; agents use IPC to interact. The creation of one process for each agent ensures isolation: each agent has a separate address space and resource sharing among processes is achieved by explicit cooperation. Processes cannot be easily managed from other processes and only the operative system (OS) scheduler has complete control over their execution;
- agent-thread model: for each agent only a single thread, hosted into the process of the platform, is created and it is responsibility of programmers to simulate multi tasking. Interaction is fast and efficient because it does not cross the process boundaries. Threads allow an easy management of agents: by using common APIs agents can be started, stopped, resumed, and terminated. Isolation is not guaranteed since threads within a process share the same address space. Furthermore, threads cannot be given different execution permissions and they normally run with the same privileges of the owning process. Nonetheless, the common adopted solution for agent scheduling is the use of threads. A common extension to this model consists in assigning a variable (or fixed) number of threads to each agent in order to increase the degree of parallelism.

We can notice that the use of a particular scheduling model affects the performance of other features like interaction and basically communication. Communication is fundamental for software agents: the messaging infrastructure dramatically changes if object instances have to cross the boundary of a system process. The same considerations apply to the software infrastructure required to expose platform services. These are some of the reasons which shifted the community of researchers to prefer solutions based on the agent-thread model (Fonseca et al., 2002). Solutions based on the agent-process model are more appealing for systems based on single software agents or where the infrastructure defining the MAS is very simple and lightweight. In these scenarios agents can really take benefits from the real autonomy which is assigned to every OS process.

The scheduling model proposed within AgentService tries to take the advantages of the two models by defining a flexible scheduling infrastructure which can be tuned according to different scenarios and resource requirements. In defining the scheduling model, *Application Domains*, a new software abstraction introduced with the CLI (Standard ISO, 2003), and the *Remoting* communication infrastructure

play an important role. Application Domains are a sort of light weight processes contained within CLI process and constitute the unit of execution for the CLI virtual machine (hereafter managed runtime). Application Domains can run with different security privileges from the owning process and have a separate address space. They also provide a unit of isolation for code execution and the only way to communicate with them is the use of *Remoting*. *Remoting* provides an explicit communication channel between objects which reside in different Application Domains either they are in the same process, or on different processes on the same machine, or on different process on different machines.

The solution implemented can be considered as a variation of the agent-thread model which gives the possibility to use a wide range of scheduling models that can be dynamically selected at platform start-up. The elements defining the scheduling engine are mainly constituted by the *agent factory* and the *agent scheduler*: the former is responsible of creating agent instances and the latter manages the execution of behaviour objects belonging to each agent. Both of the two elements can be configured at platform start-up and the framework provides some ready-to-use agent factories and agent schedulers. By separating these functionalities from the system core, the scheduling engine can implement both of the solutions previously presented. In particular, we can classify the different available solutions by examining the number of Application Domains and the number of thread used; it is interesting to analyze the effects of combining the different factories and schedulers provided by the framework. The system is open ended since third parties can implement their own schedulers and factories if the options provided by the framework do not meet the given requirements.

The framework provides two different agent factories: *LightAgentFactory* and *AppDomainAgentFactory*. The former emulates the behaviour given by the agent-thread solution: in this case all the agent instances are created within the same Application Domain. The latter gives a smarter control over the mapping between the Application Domains used and the agents created; such mapping is controlled by a policy which selects, end eventually creates, the Application Domain to be used for the next agent instance. Software developers can define their own custom policies and instruct the platform to use them, but, at the end, the decisions taken with the policies can always be overridden by platform administrators who can define restrictions on the usage of system resources. By using the *AppDomainAgentFactory* and defining custom policies we can:

- emulate the behaviour of the *LightAgentFactory*: we just have to define a policy which use only one Application Domain;
- emulate the behaviour of the agent-process model with better performances: we simply create a new Application Domain for each agent. The communication between Application Domains within

the same process is relatively fast if compared to inter-process communication;

- group a set of closely related agents within the same Application Domain in order to control them as a single unit: Application Domains can be unloaded and all the dependent resources released while the hosting process is running;

These three classes can meet the requirement of many different scenarios. We can observe that, even if Application Domains are extremely powerful and can be considered as lightweight processes, they have a cost for the managed runtime since they require additional data structures inside the virtual machine and need additional OS resources.

While agent factories are responsible of setting up agent instances, agent schedulers control the execution of behaviour objects within a single agent. In order to understand the features of the different options provided by the framework we can have to spend some words on the different types of behaviour objects which can be defined. Behaviour objects can expose the *thread-less* property which means that they do not require a separate thread to be executed. Thread-less behaviour objects are executed sequentially by using a round-robin strategy and can be processed with a single thread, while each non thread-less behaviour object requires a thread.

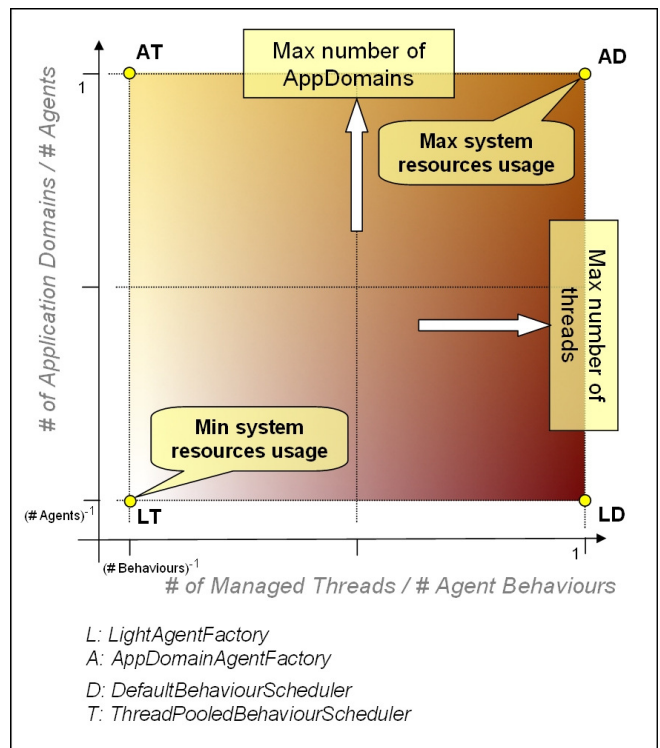


Figure 4 System resource usage graph

In order to make AgentService suitable for different application scenarios, the framework provides platform users with three different agent schedulers: *DefaultBehaviourScheduler*, *ThreadPooledBehaviourScheduler*, and *CompositeBehaviourScheduler*. The first one must be used for agents which have non thread-less

behaviour objects, this is the most expensive scheduler since requires the highest number of threads. The *ThreadPooledBehaviourScheduler* is a dedicated scheduler for agents with thread-less behaviour objects; it implements a custom thread pool which allows configuring the number of threads to use. The *CompositeBehaviourScheduler* is a composition of the previous schedulers and should be used when software agents have either thread-less or non thread-less behaviour objects; it simply dispatches the management of behaviour objects to the scheduler required by their type.

We can observe that non-thread-less behaviour simplifies the development but requires more resources at runtime while thread-less behaviour requires more attention in the development phase but can be managed more efficiently at runtime. The combination of the agent factory and the agent scheduler results an extremely flexible scheduling engine which can be extended with more complex and powerful implementations of these components. Figure 4 shows the advantages and the drawbacks of the possible combinations that can be selected for a given multi-agent system.

---

## 5 DISTRIBUTED MULTI-AGENT SYSTEM SUPPORT IN AGENTSERVICE

---

Multi-agent systems are particularly attractive for creating software operating in environments that are distributed and open, such as multi-robot systems, manufacturing automation systems, and of course the internet; commonly agents and MAS are used as a metaphor to model complex distributed processes. Nowadays, distributed systems, particularly the Web and other Internet-based applications and services, are of unprecedented interest and importance. Tanenbaum defines a distributed system as a collection of independent computers that appear to the users of the system as a single computer (Tanenbaum and Van Steen, 2002).

We can define a distributed multi-agent system as a multi-agent system composed by software agents running on different nodes of a network. Hence, the area of distributed agent computing is the area in which distributed systems enable, or facilitate, multi-agent systems, and multi-agent systems are a special kind of distributed application.

In order to be distributed AgentService provides a communication infrastructure for inter-platform messages exchange based on Web Service technology along with a federation mechanism for directory services. In addition, the framework offers a support for agent mobility that is a general solution for distributed systems. A mobile agent can carry out complex tasks which require the agent to migrate from a network place to another one. In the next sections we present a brief description of the distributed communication system and analyze the mobility service of the framework.

### 5.1 Inter-platform communication

As previously described, the messaging subsystem of AgentService is managed by a dedicated module which

drives the communication service based on message exchange. AgentService provides agents with a communication service that is transparent from the point of view of the physical agent location. The reference scenario considers the presence of many federated platforms deployed among different network nodes; the message service is able to deliver messages to agents distributed over this network. The framework adopts the Web Service technology (Chinnici et al., 2003) for inter-platform message transport which is tightly coupled with the Messaging Module. Hence, agent messages travel within SOAP messages and have to be serialized in XML format. Since messages in AgentService can contain instances of any serializable type of the Common Language Infrastructure, we choose to serialize in XML format the envelope and the description of the message content while the specific instances of the body items are serialized in base64 format. In this way the Web Service that receives the message can deserialize the envelope and then analyze the information about message content in order to verify it before the deserialization on that network node. The process of transferring messages among different platforms is completely transparent to agents and proceeds as follows:

- an agent sends a message to its peer;
- the message is checked by the MTS which sends it through the Messaging Module;
- Messaging Module realizes that the message receiver is deployed on a different platform, hence it checks the federated platforms list and finds the address of the right one;
- Messaging Module works as client of the Web Service on the target platform and invokes the web methods passing the agent message as parameter;
- the Web Service on the target machine deserializes the received message and passes it to the local Messaging Module which takes care of the delivery to the queue of the target agent.

The adoption of Web Service technology ensures standards compliance for network communication; it allows AgentService agents to easily interact with external software components and vice versa. It can be noticed that the implementation of this communication service is the one offered by the default installation of AgentService but, as the others platform modules, can be substituted with another implementation.

### 5.2 Agent mobility infrastructure

Mobility is an important agent property, as it is for objects, as well. Mobile agents have emerged as a paradigm for structuring distributed applications: software mobility can bring robustness, performance, scalability or expressiveness to systems (Karnik and Tripathi, 1998). A mobile agent must contain all of the following models: an agent model, a life-cycle model, a computational model, a security model, a

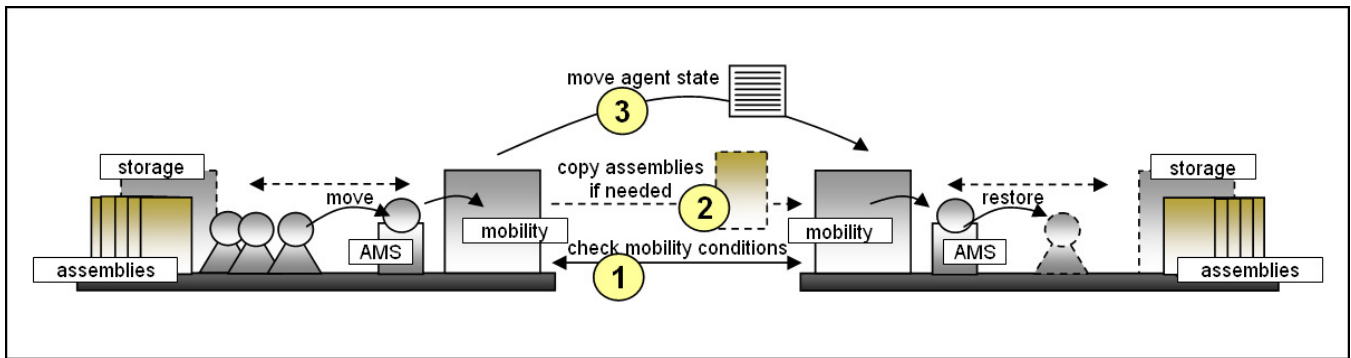


Figure 5 Agent Mobility Process

communication model and finally a navigation model (Nwana, 1996).

The agent model adopted by AgentService is compliant with the above definition. Concerning the navigation model we rely on the agent life-cycle described by FIPA. This specification extends the common life-cycle by adding the *transit* state and two actions to enter and leave that state (*move* and *execute*). This allows the current state of the agent to be represented within the AMS. The agent itself can require the move action while the platform, through the AMS, is responsible of completing the migration by performing the *execute* operation.

In literature there is a distinction between two different kinds of mobility based on whether the execution state is migrated along with the unit of computation or not (Cabri et al., 2000). Mobility in AgentService concerns the former option, commonly called “weak mobility”, but provides some additional features and can be done transparently for programmers. The main idea is to exploit the adopted agent model and move just the agent state (knowledge objects) among platforms. Within the target platform the agent activities can be restarted taking advantage of the persisted agent state. In addition, the framework provides developers with an entry point, the *Resume* method, for checking the state and the activities of the agent before it continues the execution.

In order to execute an agent the AgentService runtime environment needs the information defining the agent state and the assemblies containing the agent definition. Hence, moving an agent among AgentService installations requires moving its state and ensuring the presence, on the target platform, of the template defining the agent. The information about knowledge objects and the state of each behaviour object (ready, active, suspended) are all that we really need to accomplish agent migration.

The mobility service is implemented by an additional module which takes care of agent state migration and the transfer of assemblies if needed. When an agent requires to be moved, the AMSs of the interested platforms contract to allow mobility and then delegate to the mobility modules the transfer. The phases of the mobility process are the following:

- contract with the target platform the mobility of the agent getting information about target storage;
- stop the agent activity, persist the agent state, and put it into the transit state (*move* action);
- if necessary, transfer the assemblies defining the specific agent template;
- transfer the persistent agent (the state of the agent containing all the knowledge item, the AID<sup>2</sup>, and the state of each agent behaviour);
- create an instance of the agent on the target platform, restore the agent knowledge, create all the behaviour objects, and put them into the same state in which they were before migration;
- invoke the *Resume* method allowing the programmer to customize the agent activation;
- deploy the agent and put it in the active state (*execute* action).

The transfer process of an agent can be activated either by the agent itself, by sending a request to the AMS, or through the platform control interface: platform administrators can decide to move agents among different installations. By means of the platform APIs other software applications can manage the mobility of agents and apply load balancing algorithms.

### 5.3 Load balancing services

Agent mobility can be a solution to the problem of distributing the load in a network of computational entities: multi-agent systems can decentralize the distribution of the computational load. In fact, a complex application can be divided into autonomous parts, each of which delegated to a mobile agent. Each mobile agent is in charge of searching for the most convenient node/platform of the network, where to execute its own part of code. During execution, agents can move to other nodes where more computational resources are available, in order to better distribute the load.

<sup>2</sup> AID is the acronym of Agent Identifier adopted by FIPA to univocally identify a software agent.

Load balancing in AgentService is managed by the Load Balancing Policy (LBP) module. It provides a service that federates platform instances and creates a unique environment in which agents can move. By default, LBP comes with two policies: one based on the number of agents and one based on the number of exchanged messages. The first policy balances the number of agent between platforms, while the second moves in the same platform the agents interacting more frequently.

## 6 DEVELOPMENT SUPPORT

The AgentService framework comes with a set of software components that can help agent programmers and speed up the development process of MASs.

A multi-agent system is a complex and powerful system able to expose an intelligent behaviour thanks to the high volume of interactions among agents. Hence the design of agent interaction protocols is a crucial task that can take a lot of time for developers; AgentService provides a tool for modelling communication protocols which are exportable and easily integrated into behaviour objects.

The development support provided by the framework does not only cover the visual definition of interactions but also takes care of the coding phase. Commonly, programming multi-agent systems leads developers to deal with languages that are not agent-oriented; in addition many agent frameworks impose some programming patterns. Hence the introduction of agent oriented features within standard object oriented language can enhance development activity making programmer work more effective.

In this section we speak about tools and features for supporting the development of MAS targeting AgentService.

### 6.1 Interaction protocol design and ontology support

A focal point in a multi-agent system is communication: agents do their best cooperating and negotiating each others. The interoperation is message based and the semantic content can be defined by using ontologies. The ontology provides a set of elements for creating sentences within a dialogue and a set of rules for its proper use. However ontology does not define the structure of a conversation but only the content of each single phrase. Hence it is important to have a tool for graphically modelling agent interaction protocols and verify the correctness of its formal specification. We developed a graphical tool for the definition of agent communication protocols which are integrated with the ontology system and can be exported to the AgentService object model. AgentService is integrated with Protégé (Noy et al, 2000) and relies on it in order to visually model ontologies. Protégé projects are automatically converted into a collection of classes representing them into the AgentService object model. Such types are compiled and made available to the protocol designer. Figure 6 describes the development process.

In the field of communication protocol engineering different representation techniques are proposed (Huget and Koning, 2003) but a standard specification for agent protocols is not yet defined. We adopt the Agent Unified Modelling Language proposed by Odell (Odell et al., 2001) and then reviewed by FIPA (www.fipa.org) which can be considered the most authoritative. In addition to the elements proposed by AUML specification, we introduced some new features in order to better map the programming model of AgentService. The AUML model is defined as an XSD schema therefore the protocols can be exported in xml format.

In particular by using protocol designer developers can:

- design protocol steps for each agent role involved;
- adopt previously defined ontology for message content;
- export the protocol in a format usable by AgentService;
- design agents interpreting the roles through dedicated behaviours.

The designer only defines the structure of the protocol and creates classes which represent the state machine executing the protocol. It is up to the agent playing a specific role to complete the state machine with all the information it needs. This contract is defined by an interface which behaviour objects have to implement in order to participate in the protocol. Such interface gives access to all the messages exchanged in the protocol, manages exceptions, and allows the implementation of choices to drive the execution of the protocol. The tool can generate an assembly containing the code that defines the state machine of each role so agents can execute the state machine within their behaviours.

Figure 7 shows the process that starting from a graphical representation of the protocol generates agents interpreting it. The use of the protocol designer along with the

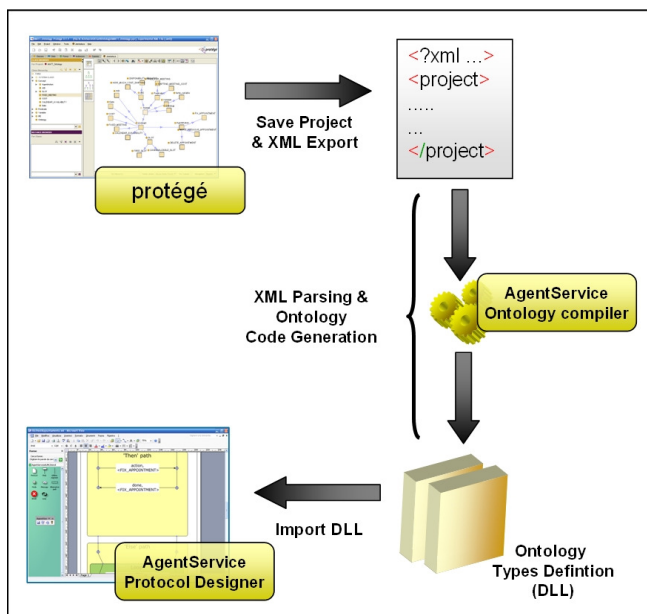


Figure 6 Ontology Development Process

customization of the agent behaviour allows programmers to obtain a reliable and accurate system in a quick way.

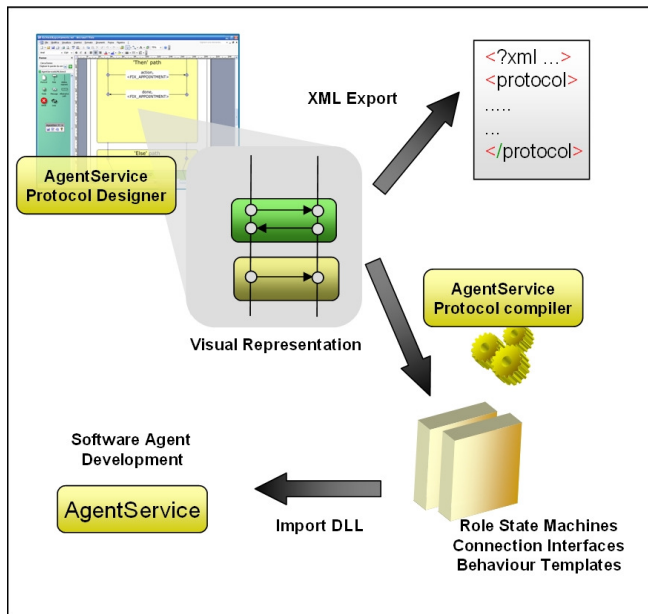


Figure 7 Protocol design and implementation

## 6.2 Language support and IDE integration

The development of agents with the AgentService framework requires programmers to follow some programming patterns ensuring the production of safe code that takes advantage of the features of the adopted agent model. These programming patterns are mostly concerned with the definition of agents, knowledge and behaviour objects, and knowledge objects access. The wrong use of such patterns does not cause the platform instance to crash but only terminates the execution of the “ill” code.

Writing the code required by AgentService can be sometime annoying and exposes to developers some uninteresting framework internals. This task can be partially automated: the information needed to implement the software pattern required by AgentService can be inferred from the context and developers just have to specify some additional properties. In order to increase and simplify the implementation task we designed a set of extensions to the C# language which represent the key concepts of agent model adopted by AgentService. In particular new language constructs have been defined to represent agent templates, knowledge and behaviour types: in order to model these concepts within the language we added the keywords *agent*, *knowledge*, and *behaviour*. Such extensions (APX – Agent Programming eXtensions) are translated by a modified C# compiler into the programming patterns required by the framework. The compiler also performs additional checks in order to produce safe code: during compilation it verifies the proper use of the extensions and performs an additional semantic analysis in order to detect concurrency flaws when accessing knowledge instances. In particular the *use* keyword is introduced in order to control the scope of

knowledge objects making them visible only within thread-safe code blocks.

AgentService proposes a development model where agent, behaviour, and knowledge instances are completely managed by the runtime environment. The extensions introduce into the language the concept of template that is a type that can be defined but cannot be directly managed (i.e. instantiated) which are used to define agent, knowledge, and behaviour types. Such templates expose just the properties required for their definition; the compiler translates them into classes by adding all the boiler-plate code required by the framework and verifying the right implementation of patterns, thus reducing the number of possible bugs.

APX does not define a new language but just some extensions integrated into the C# language: developers can take advantage of these features while they define agents, knowledge, and behaviour objects, without renouncing to the strength of C#. APX introduce some “syntactic sugar” and additional type checks that speed up and simplify the development of code targeting the AgentService framework; this is a technique already adopted by other compilers like the C# compiler itself for some high-level constructs (i.e. properties and the *lock* statement). The work performed by the APX compiler frees users of taking care about the details and the patterns imposed by the framework.

In order to completely support the development process, the framework provides a software package integrating the Agent Programming eXtensions into the Visual Studio .NET development environment. The package allows users to write code either with APX or with the C# language and exploits all the services offered to other languages like C#, VB.NET, and C++:

- project management;
- wizards and templates;
- source code control, syntax highlighting, and on-line parsing;
- package deployment.

These features allow programmers to organize all the elements of a software project, to detect many bugs during the implementation phase, and to give a complete support to code management and production. Thanks to the package the APX compiler is integrated into the environment and it is used side by side with the compilers available for the other installed languages.

The package has been development using the features provided by Visual Studio Integration Program (VSIP) that allows third party software vendors to integrate their software into Visual Studio .NET.

## 7 RELATED WORK

In the last decade, the creation of software tools and environments supporting the development of agent oriented applications has been an active field of research. Featured

	<i>AgentService</i>	<i>JADE</i>	<i>LEAP</i>	<i>AgentFactory</i>	<i>JACK</i>
Technology	CLI (.NET)	Java	Java	Java	Java
FIPA Service Components	v	v	v	v	
FIPA Compliant		v	v		
Agent Model	Knowledge-Behaviour	Behaviour	Behaviour	BDI	BDI
Native BDI				v	v
Customizable Scheduler	v				
Modular Architecture	v				v
Mobility	v	v	v	v	
Interaction Designer	v			v	v
Runtime Tools		v	v	v	v
Ontology Support	v	v	v	v	v
Language Supports	v			v	v
Methodology Support		v	v	v	v
Mobile Devices Support			v	v	

**Table 1** Frameworks Comparison

and authoritative agent community web sites, such as agentlink ([www.agentlink.org](http://www.agentlink.org)), multiagent systems ([www.multiagent.com](http://www.multiagent.com)), and agents portal ([aose.ift.ulaval.ca](http://aose.ift.ulaval.ca)), list a plethora of software tools, libraries, and toolkits to design, build, and deploy multi-agent systems. Among them just a few are still active and obtained a wide interest from the community of researchers or the industry field.

At now, much of the effort in supporting agent systems development is oriented to maintaining and extending with new features the most successful solutions. According to the definition introduced in section 2, not all of these projects can be considered frameworks. In this section we will briefly overlook those projects which, according to our opinion, are the most distinctive and important then we will sketch a comparison with the work presented in this paper. Some important projects, like FIPA-OS (Poslad et al., 2000), are not still maintained, we will focus our attention on those actually used and supported by the agent researchers community, in particular Jade, Jack, LEAP, and Agent Factory.

JADE (Bellifemine et al., 2001) is one of the most known and used agent programming frameworks; the development of JADE has been started at CSELT (now TILab) and has been supported and extended (i.e. JADEx (Pokahr et al., 2005) by a wide community of researchers. The core of JADE is constituted by a FIPA compliant middleware hosting agents, a library of classes that developers can use or extend while creating agents, and a set of graphical tools for runtime monitoring. The agent model proposed by JADE is based on the concept of tasks constituting the activities performed by the agent. JADE agents are maintained in containers which can be easily managed and moved among different JVMs. The set of connected containers define a distributed JADE agent platform, which can be easily managed through the GUI delivered with the framework.

LEAP (Berger et al., 2001) is the synthesis of two earlier projects: JADE and ZEUS (Nwana et al., 1999). The JADE component delivers a FIPA-compliant agent platform

constituting the target of LEAP agents, which can be easily developed with the tools shipped with the ZEUS component. Specifically, LEAP includes a re-factored version of the JADE code base that is compliant with J2ME and extends this code base to include the additional ZEUS functionality. Users can take advantages of the ZEUS development environment and deploy agents on a well established agent platform that is JADE.

JACK Intelligent Agents (Busetta et al., 1999) is an industry strength product aiming to integrate the agent-oriented technology into large legacy software systems. JACK agents are designed as components that can interact with pre-existing object oriented systems. The main components of the framework are a set of syntactical additions to the Java language, along with a customized compiler, which simplify the development of JACK agents. Such extensions rely on a kernel of classes which constitute the run-time support to the generated code. Moreover, the plug-in architecture allows developers to extend the framework with new features like support for additional agent models, different from the native BDI support, as well as new communication channel and language extensions.

AgentFactory (Collier et al., 2003) is a framework promoting a structured approach to the development and the deployment of agent oriented applications. It delivers extensive support for the creation of BDI agents and puts the emphasis on reusable *agent designs* which are the starting point to define new agents: through inheritance *agent designs* are extended with new features. The framework is organized into two core environments: the Agent Factory Development Environment and the Agent Factory Run-time Environment. The former delivers an integrated toolset giving support from design to deployment, while the latter provides facilities to deploy agent-oriented applications over a wide range of network-enabled Java compliant devices. The rich set of CASE tools integrated into the Agent Factory framework makes it an interesting solution for rapidly prototyping reusable agent components.

Along with the previously described frameworks there are many software projects which target specific applications or

focus on a particular aspect of MASs: the tracy toolkit (Braun and Rossak, 2005) mainly addresses the support of mobile agents, while TuCSoN (Omicini and Zambonelli, 1998) allows defining MASs as a set of concurrent Prolog programs which interact and modify the environment constituted by artifacts (non agent, reactive software components).

Table 1 compares the projects previously described with AgentService. We chose some of the most representative features that range from architectural aspects of the frameworks, compliance with FIPA specifications, presence of CASE tools, and methodologies. As we can see from the comparison, all the frameworks except AgentService rely on the Java technology. Almost all of them are FIPA compliant but such conformance does not guarantee them to naturally interoperate among each other. JADE and Agent Factory are the closest solutions to AgentService. Class inheritance is the common way for defining new agent prototypes and instantiation of these classes is the method for creating agents. On the contrary, AgentService uses class inheritance for agent prototyping and composition for agent instantiation: new templates are defined by inheriting from the AgentTemplate class, agent instances belonging to the Agent class refers to a specific template. This model ensures agent autonomy and high decoupling between runtime environment and the agent model.

While many projects rely on the well known BDI architecture, JADE and AgentService propose more general and flexible agent model based on the notion of concurrent activities. Such model implies the presence of an internal agent task scheduler. Almost all the frameworks provide a fixed scheduler while AgentService presents a flexible scheduling engine able to target different application and resource scenarios.

AgentService and JACK seem to offer the higher level of extendibility through the adoption of a modular architecture: this is an important point for project maintenance. By adding new modules or providing a different implementation for an old one it is easy to comply with very different application requirements.

The proposed agent development environment seems to be able to provide the most flexible and open solution from different points of view (agent model, scheduler engine, architecture), this can be a great advantage for AgentService developers. Such degree of flexibility obviously implies some drawbacks: the best platform configuration often requires users to deep understand some framework internals.

For nowadays agent applications, mobility and ubiquity are two fundamental features: the possibility to distribute a multi-agent system even over portable devices gives an added value to multi-agent programming frameworks. Even if the core components of AgentService can be run on the .NET Compact Framework<sup>3</sup>, we are working on a more effective solution which really takes in account the constraints of embedded contexts. In addition, our

framework lacks in AOSE methodologies and in advanced administrative and deployment tools which are now being implemented.

---

## 8 SUMMARY AND CONCLUSION

---

In this paper we presented AgentService, a framework to develop distributed multi-agent systems. The core components of the framework are a library of abstractions which easily allow implementing agents, and a software environment delivering all the services to develop and set up multi-agent systems. Moreover a collection of tools – designers, language extensions, and management interfaces – support end users during the life cycle of the MAS. Our aim is to simplify the development process of a multi-agent system without renouncing to a good support for high quality software production. AgentService uses simple abstractions – the agent model, the language extensions – which are integrated into a flexible software infrastructure. Such infrastructure provides a high degree of customization and extensibility: we have been able either to instrument the framework with different scheduling engines or to add the mobility service as an additional component with low integration costs. We think that customization and extensibility will play a key role for the future of AgentService because they contribute to make the framework open and evolvable. These features are a fundamental requirement to support the life-cycle of a distributed multi-agent system,

Design and monitoring tools complete the overview of the framework: these components offer a valuable aid which is complementary to the implementation phase. We can visually define or monitor many aspects of the life-cycle of a MAS, thus delivering to the end user a good support to software development. AgentService does not provide any facility to cover the analysis phase; it is our intention to integrate this functionality into the framework by providing support for most known analysis methodologies (Wooldridge and Jennings, 2001; Chella et al., 2004; Wood and DeLoach, 2000) and development processes, in particular Tropos (Giorgini et al., 2004) for what regarding the goal analysis (Giorgini et al., 2005). AgentService has been proven to be as flexible enough to tailor multi-agent systems for very different scenarios: we are using it in the MATT project (Grosso et al., 2005), a groupware application where agents have been used as personal assistants which share an agenda and collaborate to schedule meetings; we have also investigated the use of our framework in the field of grid computing (Vecchiola et al., 2006). The use of requirement analysis tools integrated into the framework would have been very useful during these projects, we think that augmenting the framework capabilities with such functionalities will make AgentService a simpler and more powerful tool for developing multi-agent systems.

More information can be found on the AgentService website ([agentservice.lido.dist.unige.it](http://agentservice.lido.dist.unige.it)).

---

<sup>3</sup> The Microsoft .Net Compact Framework is a version of the .NET Framework that is designed to run on mobile devices such as PDAs, mobile phones and set-top boxes.

---

**REFERENCES**


---

- Bauer, B., Muller, J. P., and Odell, J. (2001) 'Agent UML: A formalism for specifying multiagent software systems', *Int. Journal of Software Engineering and Knowledge Engineering*, Vol. 11 No. 3, pp. 207-230.
- Bellifemine, F. Poggi, A., Rimassa, G. (2001) 'Developing multi agent systems with a FIPA-compliant agent framework', *Software - Practice & Experience, John Wiley & Sons Ltd.*, Vol. 1, No. 31, pagg. 103-128.
- Berger, M., Bauer, B., Watzke, M. (2001) 'A Scalable Agent Infrastructure', *2nd Workshop on Infrastructure for Agents, MAS and Scalable MAS. Autonomous Agents'01*, Montreal.
- Braun, P. and Rossak, W. R. (2005) 'Mobile Agents - Basic Concepts, Mobility Models, and the Tracy Toolkit', *Morgan Kaufmann Publishers*.
- Busetta, P., Ronnquist, R., Hodgson, A., and Lucas, A. (1999) 'JACK Intelligent Agents - Components for Intelligent Agents in Java', *Technical Report TR990, AOS*, January.
- Cabri, G., Leonardi, L. and Zambonelli, F. (2000) 'Weak and Strong Mobility in Mobile Agent Applications', *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000)*, Manchester, U. K., April.
- Chella, A., Cossentino, M., and Sabatucci, L. (2004) 'Tools and patterns in designing multi-agent systems with passi', *WSEAS Transactions on Communications*, January, Vol. 3, No. 1, pp. 352-358.
- Chess, D., Harrison, C., and Kershenbaum, A., (1995) 'Mobile Agents: Are they a good idea?', *Technical Report of IBM T.J. Watson Research Center*, NY, March.
- Chinnici, R., Gudgin, M., Moreau, J., and Weerawarana, S. (2003) 'Web Services Description Language (WSDL) Version 1.2', *W3C Working Draft*, January.
- Collier, R., O'Hare, G.M.P., Lowen, T., and Rooney, C. (2003) 'Beyond prototyping in the factory of agents', *Proceedings of the Third Central and Eastern European Conference on Multi-Agent Systems - CEEMAS'03*, Prague, Czech Republic.
- Fonseca, S.P., Griss, M.L., Letsinger, R. (2002) 'Agent behavior architectures a MAS framework comparison', *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pp. 86-87
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) 'Design Patterns - Elements of Reusable Object-Oriented Software', *Addison-Wesley Longman*.
- Giorgini, P., Kolp, M., Mylopoulos, J., and Pistore, M. (2004) 'The Tropos Methodology: An Overview', *Proceeding of Methodologies and Software Engineering for Agent Systems*, Kluwer.
- Giorgini, P., Mylopoulos, J., and Sebastiani, R. (2005) 'Goal-Oriented Requirements Analysis and Reasoning in the Tropos Methodology', *Engineering Applications of Artificial Intelligence*, Elsevier, March, Vol. 18, No. 2.
- Glass, G. (1998) 'ObjectSpace Voyager - the Agent ORB for Java', *Proceedings of Worldwide Computing and its Applications (WWCA'98)*, Springer, Tsukuba, Japan, March, pp. 38-55.
- Grosso, A., Vecchiola, C., Coccoli, M., and Boccalatte, A. (2005) 'A Multiuser Groupware Calendar System Based on Agent Tools and Technology', *Proceedings of The IEEE 2005 International Symposium on Collaborative Technologies and Systems - CTS2005*, St. Louis, Missouri - U.S.A., May, pp. 144-151.
- Huget, M., Koning, J. (2003) 'Interaction Protocol Engineering in Multiagent Systems', *MATES 2003 Tutorial*.
- Karnik, N.M., and Tripathi, A.R. (1998), 'Design Issues in Mobile-Agent Programming Systems', *IEEE Concurrency*, July-September, Vol. 6, No. 3, pp. 52-61.
- Lange, D., and Oshima, M. (1998) 'Programming and Deploying Java Mobile Agents with Aglets', *Addison-Wesley*.
- Mattern, F. (2000) 'State of the Art and Future Trends in Distributed Systems and Ubiquitous Computing', *Vontobel TeKnoBase*, August.
- Mitsubishi Electric ITA Horizon Systems Laboratory (1997) 'Concordia: An infrastructure for collaborating mobile agents', *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, Berlin, April.
- Noy, N., Sintek, M., Decker, S., Crubezy, M., Ferguson, R., and Musen, M. (2000) 'Creating Semantic Web Contents with Protege-2000', *IEEE Intelligent Systems*, Vol. 16, No. 2, pp. 60-71.
- Nwana, H. (1996) 'Software agents: An Overview', *Knowledge and Engineering Review*, November, Vol. 11, No 3.
- Nwana, H., Ndumu, D., Lee, L., and Collis, J. (1999) 'ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems', *Applied Artificial Intelligence Journal*, Vol. 13, No. 1, pp. 129-186.
- Omicini, A. and Zambonelli, F. (1998) 'The TuCSon coordination model for mobile information agents', *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, June.
- Pokahr, A., Braubach, L., and Lamersdorf, W. (2005) 'Jadex: A BDI Reasoning Engine, in Multi-Agent Programming', *Springer Science+Business Media Inc.*, USA, pp. 149-174, September.
- Poslad, S., Buckle, P., and Hadingham, R. (2000) 'The FIPA-OS agent platform: Open Source for Open Standards', *Proceedings of PAAM2000*, Machester, UK, April.
- Standard ISO/IEC (2003) 'Common Language Infrastructure', March, 23271:2003.
- Tanenbaum, A.S., and Van Steen, M. (2002) 'Distributed Systems: Principles and Paradigms', *Prentice-Hall*, Englewood Cliffs, NJ, U.S.A.
- Vecchiola, C., Coccoli, M., and Boccalatte, A. (2003) 'Agent Programming Extensions relying on a component oriented infrastructure', *Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration (IRI - 2003)*, Las Vegas, NV, October, pp. 26-29.
- Vecchiola, C., Grosso, A., Podestà, R., and Boccalatte, A. (2006), 'Design and Implementation of a Grid Architecture over an Agent-Based Framework', *Proceeding of .NET Technologies 2006 - 4th International Conference*, Plzen, Czech Republic., May.
- Wood, M.F., and DeLoach, S.A. (2000) 'An Overview of the Multiagent Systems Engineering Methodology', *The First International Workshop on Agent-Oriented software Engineering (AOSE-2000)*.
- Zambonelli, F., Jennings, N., Wooldridge, M. (2003) 'Developing Multiagent Systems: the Gaia Methodology', *ACMTransactions on Software Engineering and Methodology*, Vol.12, No. 3.

---

**WEBSITES**


---

- AgentService, <http://agentservice.lido.dist.unige.it>
- Wikipedia, <http://www.wikipedia.org>.
- Foundation of Intelligent and Physical Agents: FIPA Abstract Architecture Specification. Available at <http://www.fipa.org/specs/fipa00001>
- AgentLink, <http://www.agentlink.org>
- Multiagent Systems, <http://www.multiagent.com>
- Agents Portal, <http://aose.ift.ulaval.ca>