Adaptive Agents to Heterogeneous Platforms, New Protocols & Evolving Organizations

Laurent Magnin* Viet Thang Pham* Arnaud Dury* Nicolas Besson*
Houari Sahraoui**

* CRIM, 550 Sherbrooke Street West, Suite 100
Montréal, Québec, Canada H3A 1B9
{firstname.lastname}@crim.ca
Tel. +1 (514) 840 1234 - Fax. +1 (514) 840 1244

** Université de Montréal, Québec, Canada
sahraouh@iro.umontreal.ca

Abstract

Adaptive agents provide a natural and clean way to deal with heterogeneous, distributed and unpredictably evolving environments, such as Internet. In this paper, we present a model of agents that are able to adapt themself to their environment by several ways. Using a middleware approach, our agents, called *Guest*, can run, communicate and move between different multiagent platforms, which are *a priori* incompatible. *Guest* agents can also change dynamically their capabilities, based on the concept of plug-ins. To build multiagent applications, *Guest* agents are able to organize themselves into centralized or distributed hierarchies. These agents can even change automatically their organizational models to adapt to their evolving environment, using metamodeling technique.

1 Introduction

Multiagent systems are probably one of the most suitable approaches to build applications in open, heterogeneous, evolving and distributed environments, such as the Internet (7) (8). Being autonomous, agents running in such complex environments for a long period of time need to be self-adaptive to different platforms and new protocols. To bring this important caracteristic to agents, instead of a top-down approach where the main focus is on functionalities or behavior adaptation, we follow a bottom-up approach: it is useless to provide learning capabilities to agents if they don't have first basic tools to deal with such environments.

This paper will present some of the means we are using to achieve this vision. First, we propose a model of agents that are able to run, communicate and move between different multiagent platforms. This model is based on a middleware between such agents and platforms (section 2). We then describe briefly the implementation of this model on a kind of agents called *Guest*, which can be used the same way regardless of the kind of servers they run on (section 3). Next, we introduce a framework called *plug-ins* that allows agents to dynamically add, remove or upgrade their capabilities, such as understanding of new communication language, reasoning algorithm, etc. (section 4). Third, we propose two kinds of dynamic agent hierarchies, one is based on a middleware for the centralized context, while the other is designed using plug-ins for the distributed scenario (section 5). Lastly, we present a metamodel of agent control mechanisms to allow such agents to automatically adapt to their environments using the tools we early described (section 6).

2 Adaptation to heterogeneous multiagent platforms

An agent could only run on its own platform. For example, an Aglets agent can only use an Aglets based server, not a Grasshopper based one. The opposite is also true. Considering limited multiagent applications, this is not a real problem. However, future agents running in Internet-based open applications will have to be able to adapt themselves to heterogeneous servers provided by different partners.

Three feasible solutions to this interoperability problem can be envisioned: the use of standards (3) (11), the use of converters between platforms (15) and the creation of a generic interface (middleware approach). As (2), we choose the last one: we implement our generic agents (9) by using interfaces (see figure 1), more precisely by providing an intermediate layer called *Guest* between our *Guest* agents and the targeted agent platforms (all Java

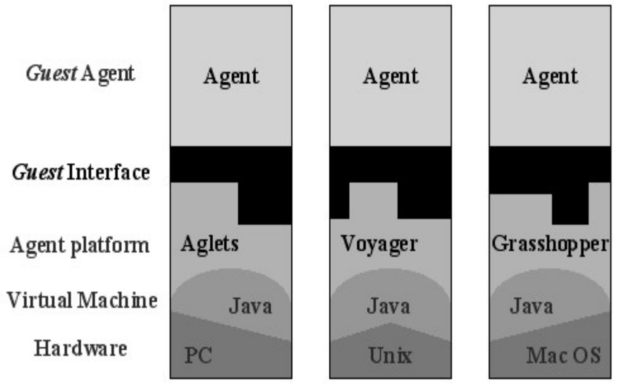

Figure 1: *Guest* interface

based). This layer is a two-sided entity: on one side, the *Guest* API that is visible to an application programmer, on the other side a platform-dependent layer, which we provide. In that way, our agent will be able to run and eventually talk to native agents on these different agent platforms while maintening the same functionality thanks to interfaces (one per platform). That does not imply that all of the platforms need to integrate these *Guest* interfaces: it is only when a *Guest* agent reaches an agent server that the *Guest* Java classes need to be downloaded from a specific server by the Java Virtual Machine (JVM) without modifying the server behavior. This method does not impose a new standard, or more precisely parties who want to use the genericity of the *Guest* API do not require that *Guest* becomes a standard. This point is essential to the success of our approach.

A *Guest* agent is made up of two facets: one is specific to the platform expected to carry the agent, the other one is independent of any platform. Moreover, in order for this *Guest* agent to move from one platform to another, it should be able to change dynamically its specific facet while maintaining its internal status. It is generally impossible to get access to the source code of agents associated with the targeted platforms. As a result, it is impossible to implement these generic agents by a modification of their code which is offered by the platforms. We must resort to interfacing with the public methods proposed by these agents. Therefore, our solution consists in designing a *Guest* agent modeled as the sum of two interconnected agents. The first one (so-called native agent) inherits from the agent class of the targeted agent platform. The second one (so-called generic agent with the purpose of implementing the agent function) inherits from the generic agent class *Guest*. It is the only part of the agent that will move between servers when the agent migrates [1]. Consequently, our agents are simultaneously perceived by platforms as being native and by their originators as being platform-independant *Guest* agents. Our model requires slightly more resources than a native model: memory consumption and CPU overhead are limited, but are far from being doubled.

A critical constraint has to be mentioned: all the selected platforms require using their own Java Virtual Machines. When a composite *Guest* agent migrates, the generic part will be handed over to the targeted JVM right after the native part is created on it. Upon completion, the composite agent will be removed from its originating platform, thus ending the migration process. One of the conditions for the agent migration is to ensure that the *Guest* agent is serializable.

As mentioned above, our *Guest* agents must have *Guest* interfaces adapted to specific multiagent platforms if they are intended to be operational. There are already *Guest* interfaces on those Java based platforms:

- ASDK, Aglets Software Development Kit (1), originally developed by the IBM Tokyo Research Laboratory.
- CorbaHost, our own Java server implementation on top of Corba (Orbix by IONA(4)) [2].
- Grasshopper (5), commercial product of German firm IKV++.
- Jade (6), an open source platform which is Fipa compliant [3].
- Voyager (12), commercial product of ObjectSpace.

3 Generic Guest Agents

Our goal is not only to have agents that can run on different platforms but also not to care about the platform they run on. In other words, the agent programmer will work with the uniform interface we provide, but will not have to deal with specific code for the different targeted platforms. For example, the agent programmer can use the same addressing schema to represent addresses of different servers, regardless of their platforms. We achieve this by providing a complete and operational set of agent features through the methods provided by the GuestAgent class. Of course, the API of such methods is the same regardless which platform the agent is running on.

3.1 How to deal with specific features of the platforms ?

By providing always the same interface to our *Guest* agents, do we restrict the features allowed to them to the most

[1]As is the case on many other agent platforms, the current state of the running agent will not been saved and restored when the agent migrates from one server to another. It is the duty of the agent programmer to solve this problem by using specific methods which are called just before and after the migration

[2]We developped this server to offer the bridge between our *Guest* agents and the world of CORBA.

[3]We specially choose this platform to provide Fipa communication capabilities to our *Guest* agents.

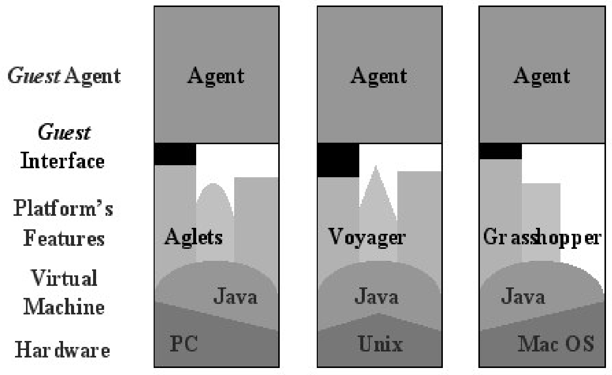

Figure 2: Equivalent features provided by native platforms

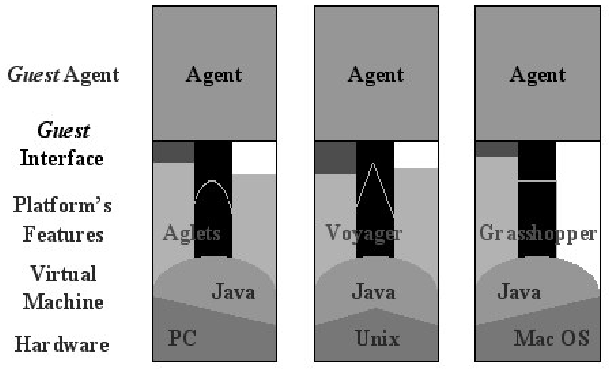

Figure 3: Different features provided by native platforms

common denominator of the different platforms they can run on? In other words, in order to be provided by *Guest* should a feature be available on all targeted platforms? If so, only limited functionalities would be given to the agents. Also, only platforms providing all of *Guest* current features would be able to be added to the targeted platforms.
Fortunately, this constraint does not hold. To explain how we deal with this problem, we have to divide the features in three sets. The first one contains features that are quite similar for all platforms. For example, Aglets, Grasshopper and Voyager provide the ability to send a message to an agent by calling a proxy.function(args) like method, so, the *Guest* interface can implement this functionality simply by calling the native method of the targeted platform. In figure 2, we show that the *Guest* code (the black square on the first column) is merely a simple function call. In the second case, a feature is different on the targeted platforms. For example, the naming services are different for Aglets, Grasshopper and Voyager. To solve this problem it is possible to implement this feature independently, inside the interface. In figure 3, we show that *Guest* implements the same functionality on the three differents platforms (the functionality being the black area on the second column of fig. 3). That way, *Guest* provides a uniform type of addresses for agents running on these three platforms.

Last case, a feature can be unavailable on some platforms. For example, Voyager does not provide a function that returns all of the agents running on a specific server. Such a function can be implemented from scratch for this platform. At the same time, this function can also be based on a call to the native functions provided by Aglets and Grasshopper. In figure 4, we show in the third column that the *Guest* implementation (the black area of the figure) varies from platform to platform, from a simple function call (on aglets and voyager) to a full implementation (on grasshopper).

In conclusion, our middleware approach allows us to provide a uniform set of functionalities to universal agents

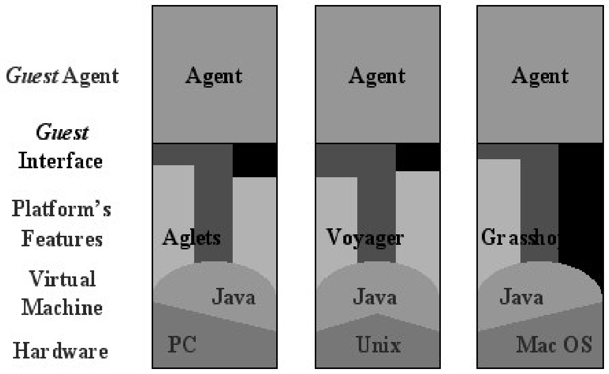

Figure 4: Lack of features provided by native platforms

despite the fact that the targeted platforms could have various features and goals.

3.2 Guest Agent Model

3.2.1 Agent's life cycle

No matter on what server a *Guest* agent executes, the agent can be in one of the following states:

- STATUS_CREATION : agent is built, but has not been initialized and can not communicate with other agents.
- STATUS_EXECUTION : agent finishes initialization, can communicate with other agents and can be fully used.
- STATUS_MIGRATION: a mobile agent is in this state when it is on the way to its destination. Such a moving agent can not directly receive and process messages, but all the messages sent to the agent during this period are buffered and later forwarded to it.
- STATUS_SAVE: when an agent stops temporarily its work and saves itself on disk, it is changed to

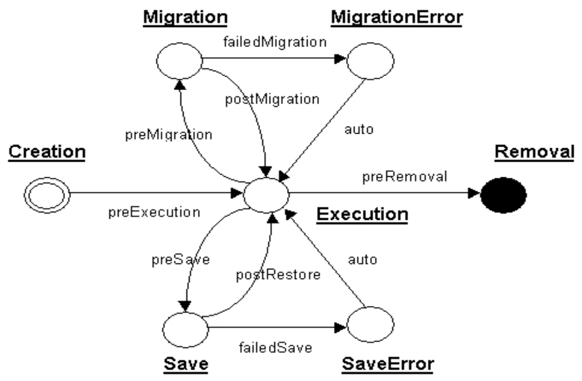

Figure 5: The agent life cycle

this state. An agent at this state can not receive messages and does not do anything. However, such an agent can be woke up by itself or by another agent to continue its task.

- STATUS_REMOVAL: agent enters this state when it prepares to die. An agent in this state can not be moved or saved.

The other two states: MIGRATION_ERROR and SAVE_ERROR are temporal states and are used internally by GuestAgent.

This life cycle model is the most common part of the different life cycles of the four targeted platforms. Moreover, it is sufficient to express different states and to support all the basically necessary functionalities of a universal agent. For example, states of a JADE agent, which strictly follows the Agent Platform Life Cycle in FIPA specification, can be easily mapped into states of our model: AP_INITIATED → STATUS_CREATION, AP_ACTIVE → STATUS_EXECUTE, AP_DELETED → STATUS_REMOVAL, AP_TRANSIT → STATUS_MIGRATION. AP_SUSPENDED and AP_WAIT should be represented as a sub-state of the STATUS_EXECUTE, because an agent in these states is still alive and the actions of SUSPEND and WAIT can only be activated by an agent on itself.

3.2.2 Agent URL

In the *Guest* platform, the address of an agent is uniformly represented by a GuestURL. A GuestURL has the syntax of a common URL: **platform_protocol://host_name — IP_address:port[/optional]**

- ***platform_protocol*** represents the protocol supporting the specific native platform. For example, *Guest* platform supports Grasshopper, Voyager, Aglets, CorbaHost [4] and JADE by offering the following protocols: GH, VY, AG, CH and JD, respectively. Based on this signature, agent can correctly activate the appropriate native part to accomplish its task ;

[4]CorbaHost is a small agents server on top of Corba we created.

- ***host_name*** is the domain name of the computer on which the agent executes. It can be replaced by the IP address of this computer ;
- ***port*** is an integer indicating the port on which the native server is listening ;
- ***optional*** is used to represent the platform-dependant extension of the GuestURL. For example, the Grasshopper agent needs to include Place and Region in its address, this extension is therefore necessary.

This form has proved to cover all the representative forms of agent addresses of the four targeted platforms. For example, an Aglets server uses the following string to represent its address: ”atp://halida.crim.ca:4434”. This address can be easily transformed into the GuestURL as ”AG:// halida.crim.ca:4434” and vice versa.

3.2.3 Agent communication

Any *Guest* agent can send and receive *Guest* messages through an uniform API. As the communication between Guest agents running on the same platform use the native layers provided by the platform, the communication between heterogenous platforms is based on Java RMI. This mechanism enables the delivery of *Guest* messages between heterogeneous platforms, but does not allow the processing of native messages exchanged between non-*Guest* agents. Such capability is nevertheless possible using direct access to the native agent, which is also supported by the uniform API. However, that code is platform-specific and may not be executed on all platforms. *Guest* platform supports three communication modes between agents: synchronous, asynchronous and future reply. While the first two modes use the communication service of the native platform, the last one is completly implemented in *Guest* based on the synchronous communication mode.

4 Adaptive agents based on plug-ins

In an open, evolving multiagent world, new algorithms and services will be put into use during the life-cycle of an agent or system. For example, a new compression algorithm may be used to compress messages, or a new kind of cryptographic signature may be released. To adapt to that kind of environments, our agents need to be able to apply ”on the fly” these new algorithmes and services, without having to restart any part of the system. More precisely, we want to be able to design ”new capabilities” in an agent-independent way, and then allow our *Guest* agents to use them *on demand*.

Guest agents are currently able to operate on heterogenoeous platforms, thanks to the interface layer on top of native agents. This interface layer encompass creation and destruction of the native agent, message sending and receiving, migration, etc and is therefore sufficient in terms of agents programming. But it is fixed at the runtime and

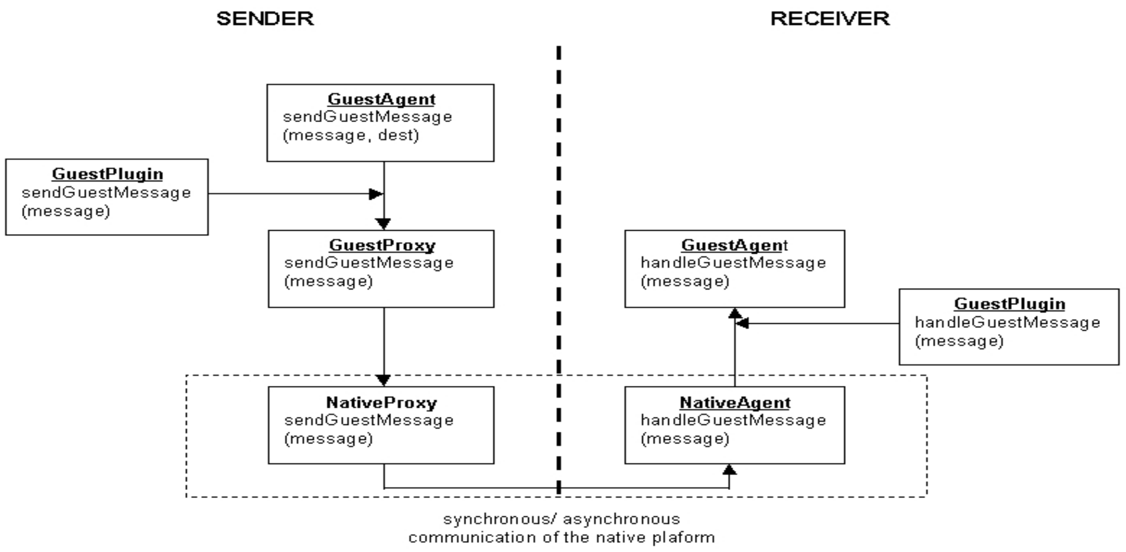

Figure 6: Two base communication modes

not suitable for dynamically adding new capabilities to an existing agent: any modification implied to this interface requires the restart of all the agent servers on which *Guest* agent are running. Therefore, we need a more convenient way to solve this problem. Our solution is to embed in the kernel of the *Guest* agent a framework called *plug-in*. A plug-in is a component (compiled code of one or some objects) which can be associated with an agent and immediately offer some new services. This framework has several clear advantages:

- Enable Guest agent to modify dynamically its capabilities and therefore adapt to the evolving environment. For example, one agent can ”learn” to compress/decompress data by loading a compression plug-in. When it no longer needs these functions, it can simply ”forget” it by unloading the plug-in. The agent can even ”upgrade” its compression technique by changing the current compression plug-in for a more sophisticated plug-in;
- Enable reuse and sharing of services: a compression plug-in can be developped and deployed by a third party. It can then be used as an *Off-The-Shelf-Component*.

A plug-in can perform any of the following three types of actions:

- Observe the change of the associated agent state and possibly prevent this change in some cases (migration or deactivation of the agent);
- Observe the communication of the associated agent (sending and receiving a message) and possibly intercept an incoming/outgoing message;
- Offer a library of new services to the agent. These services can be used by indicating the name of the service and all the necessary parameters, or by first obtaining a plug-in reference and then accessing to the services using normal function call on this reference.

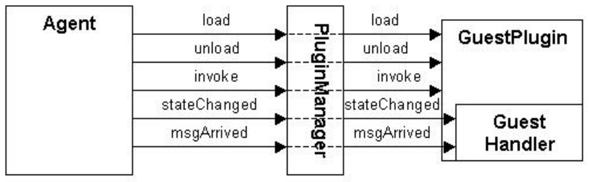

Figure 7: Plug-in framework

The relationship between a plug-in and an *Guest* agent is clearly defined in the base class GuestPlugin. This class contains necessary methods for the load, unload and discovery of a plug-in. Every plug-in must extend this class in order to cooperate correctly with the agent and offer its services to the agent user. All the new services (high-level functionality) of a plug-in are exposed to the agent user through its reference and can be easily accessed by simple function calls. Moreover, these services can be normally used outside the agent via the associated agent’s proxy, without having to modify this one. All the actions of observing/intercepting agent state/communication events are low-level functionality of a plug-in and separately handled by a ”secret part” of the plug-in, called *handler*. A handler makes part of a plug-in and is totally hidden from outside the plug-in in order to prevent unattended and unauthorized access. Every handler must extend the base class GuestHandler and implements some of the predefined interfaces to declare what kind of events it is interested in. For example, a handler which implements the interface ISynGuestMessageHandler will be automatically called by the associated agent each time a synchronous communication event is fired.

Moreover, the mecanism of event handling is not a mere chaining, but is in fact more sophisticated. Upon loading, a plug-in exports to the agent its *activation function*. This function is a filter, applied on incoming messages and events, deciding on which ones the plug-in offers its services. For example, a dedicated decompression plug-in would ask for activation only when compressed messages are received. It would then ask for activation for each potentially encapsulated compressed message. Any given plug-in can be used any number of times during the processing of a message or an event. This allows us to provide new capabilities (for instance compression and cryptography), even if we don't know in advance the proper layering of the message by the other agents (do they first sign, and then compress? Or do they compress, and then sign?). This is a very important capability in the kind of open environment we developed *Guest* for.

Our design ensures the maximum dynamicity for the modularization by allowing a plug-in to be associated with an agent at any time during the agent's life and removed whenever the user no longer needs it. The deployment of these plug-ins is also flexible: one plug-in doesn't have to be bundled in the packages but can be downloaded from an Internet site.

Using the plug-in framework, we already developed a graphical interface that allows a special user to visually observe the Guest agents on the network (using different native platforms), see figure 8.

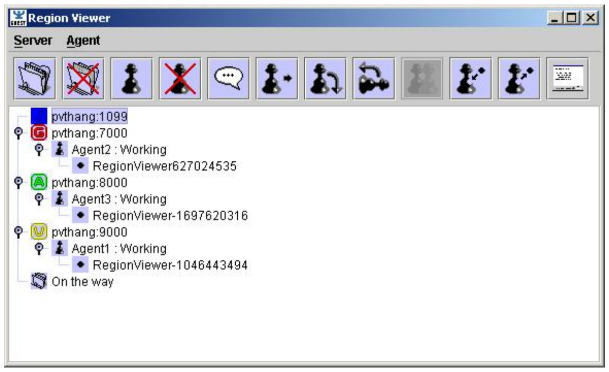

Figure 8: Guest Graphical interface

This was done by creating plug-in which intercepts the changes of agent's state during the migrations and informs the GUI to visually reflect these changes on the screen. Consequently, an agent can be observed only by associating with it this plug-in even if this agent was not initially supposed to offer this service. By the same way, it will be possible to allow a *Guest* agent to be compatible with standards of agent interactions, such as KQML(19), Fipa (3) or any new standard that can be defined in the future.

In the next section, we will show how to use the plug-in framework to intergrate the distributed hierarchical agent model in *Guest*.

5 Adaptive hierarchical multiagent systems

Not only agents need to be adaptive, but also organizations of agents need to be. Working on different kind of such organizations, we will focus on this paper on a specific one: agents' hierarchies[5], which can be dynamically modified.

We developed two different ways of building such hierarchies that are described in the following sections. The first one uses our middleware approach by implementing at the agent level the platform interface: each agent can now be seen as an agent server, which is able to receive and run other agents. The second one uses a specific plug-in we developed that can dynamically modify the routing of messages between agents to offer the functional appearance of a hierarchy.

5.1 Agents as servers

5.1.1 Principles

In the context of the *Guest* platform, an agent server is a container which is able not only to provide resources (such as CPU time) and communication service to other agents, but also to monitor and control the life cycle of these agents. We have designed *Guest* agents to be able to act as servers (parent agent), i.e. they can accept and execute other agents (children agents). These internal agents in turn can also be servers for other agents and so on recursively. Finally we have a group of agents organized into a hierarchy. In this hierarchy, only the root agent needs to be connected to the native platform, through the *Guest* interface. The other agents are just connected to their corresponding parent agents through the parent/child interface, which is mostly the same as the *Guest* interface.

This hierarchical model simplifies the design of a multiagent system, particularly for those whose processing can be divided into hierarchical tasks: the native platform sees a hierarchy as a unique agent, and the outside world may be given a unique entry-point (the root agent) for the hierarchy. The parent agent has total control on its children: migration, destruction, etc. and the root agent is thus able to control all the agents in the hierarchy. For example, when an application has to migrate from one server to another (because of harware failure, for example) it only needs to ask the root agent to move, then the

[5]A hierarchy in our platform is a direct acyclic graph of binary parent/child relations.

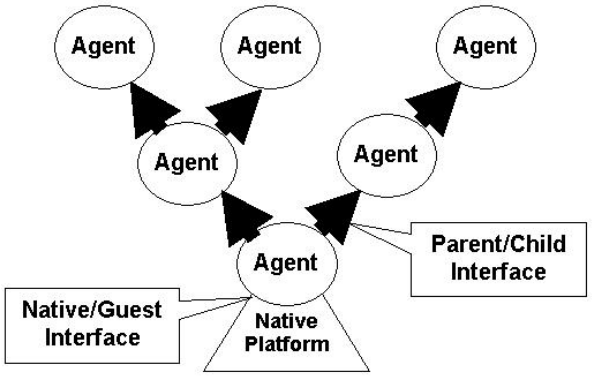

Figure 9: Agents as servers

whole hierarchy will move together. The organization is maintained and the application resume its work normally right after it arrives at the new destination. The communication between parent and child agents is local and very efficient. The consumed resources are considerably reduced because only one native agent is required for the whole hierarchy. The hierarchical model is thus one of the most natural ways to "agentify" a whole multiagent systems, except that it requires that all the agents in the hierarchy reside on the same server.

5.1.2 Implementation

This hierarchical model is implemented in the kernel of the *Guest* platform. To represent the structure of the hierarchy, the concept of GuestURL is extended to encapsulate not only the address of the server of the agent but also the path from the root of the hierarchy to this agent. That way, integration of an agent into a hierarchy is done by a migration where the URL of the target is another agent instead of a real server. The life cycle of the children is monitored and intercepted by a mechanism similar to the one used by the plug-in. All the messages sent to a child will be firstly received by its parent and then be forwarded (according to the parent policy) to it. We added a new interface into the GuestAgent to allow it to manage children' communication and their execution threads, this interface is similar to the native agent interface but has some new specific functions for the hierarchy : add, remove, etc. The child can be dynamically connected to its parent through this interface and has the impression that it is fostered by a server. The communication between the parent and its children is implemented as direct function calls, thus greatly increasing the speed of exchanging messages (compared to the usage of the "send a message" primitive).

To overcome the limit that all the agents must be on the same server, we propose in the next section another hierarchical model, using rerouting technique.

5.2 Dynamic rerouting

5.2.1 Principles

Rerouting messages between agents is the other method we use to model the creation of dynamic hierarchies between multiple cooperating agents. This mechanism allows a group of agents to organize themselves into a more efficient multi-agent structure by dynamically rerouting messages received by the structure. The main idea behind this concept of rerouting is: in order to function as a hierarchy, a group of agents needs to have a leader (root of the direct acyclic graph describing the hierarchy) that receive and dispatch all the messages for the group (this process is recursive, as is the case with the previous hierarchy model). If an agent in a hierarchy, that is not the root of this hierarchy, receives a message then it reroutes it to the root of its hierarchy. We are in fact replicating the way the previous model of hierarchy (see 5.1) works, but in this new model, agents don't need to be physically on the same server. The rerouting of messages will duplicate the way the previous hierarchy works, without its constraints of physical locality. This model is not without its flaw: the number of messages exchanged will increase, compared to the previous model of hierarchy (see 5.1).

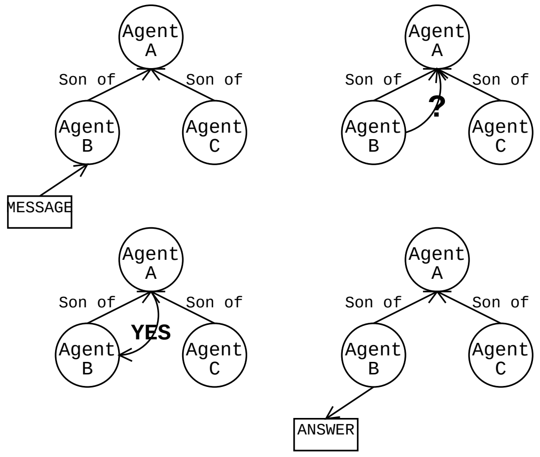

Figure 10: Building a hierarchy using dynamic rerouting

5.2.2 Implementation

This dynamic rerouting is implemented by a specific plug-in, the Hierarchical plug-in. This plug-in offers the following services:

- ask a potential father to register the agent (create a new Father-Child link)
- ask the father to unregister oneself (Child ask to destroy the link its Father)
- obtains the list of all the childs of the agent
- find whether or not a given agent is a children

- find whether or not a given agent is a grand-children
- allow each agent to express its acceptance policy for accepting a new child
- allow each agent to express its acceptance policy for letting a child leaving the hierarchy
- allow the Guest graphical interface to display the hierarchy as a tree

Group creation is implemented by a two-way handshake between the agent willing to join the group, and his desired leader in this group. The agent sends a request to the desired leader, and upon acceptance reroutes all of his incoming messages to this leader for approval. This mechanism, being used in a recursive way, leads to the creation of a hierarchy between agents.
Group destruction follows the same principle: each agent willing to leave the group generates a request to its direct father. This request can be forwarded through child to father to grandfather etc. up to the root of the hierarchy. The acceptance policy is based on the acceptance of each father, from the agent up to the root (that is, each father can express a veto on the decision). Upon entering a hierarchy, an agent looses its ability to process directly any incoming message. It will from now on have to ask its direct father for the authorization to process the incoming message (see figure 10). This ensures a uniform processing of the request by a unique agent (the leader), even if the outside agents are not yet aware of the new organization, and erroneously send their request to an agent inside the group. We now introduce a possible use of this model, considering the following context: how to add load-balancing capability to an application ?

5.2.3 Load-balancing example

A capacity that is usually not straightforward to implement for a system, but is easily implemented using this kind of hierarchy, is the capacity of load-balancing for an application. Load-balancing is the process by which a computation requiring lots of ressources (memory or CPU time usually) is distributed among a network of computers, according to the load of each computer, and the computation itself. Using Guest agents and hierarchies, a simple way of implementing load-balacing is feasible: an agent is created to realize a piece of the computation (for instance, an agent that is able to render a pixel, in a raytracing application). This agent loads the hierarchy plugin we describe in 5.2.1. This agent then connects to a GuestRegion[6] to discover the other servers. As this agent receives pixel-rendering requests from an outside (potentially non-agent oriented) application, it creates clones of itself, and these clones register themselves as children to the first agent. Each clone is able to move to a server with a lighter load than its own. The first created agent will be the only agent known to the outside world, and will process requests and dispatch them among pixel-rendering agents. These pixel-rendering agents won't respond to any other agent, and would automatically forward to their father any incoming message. In this context *G*uest provides a way to keep track of all pixel-rendering agents, to allow them to choose the server with the mininum load while keeping their link with the dispatcher agent and to enforce the rule that rendering agents will respond only via their dispatcher father.

[6]A GuestRegion is a specific service of Guest that allows agents to track existing agents and servers.

5.3 Designing a hierarchy using both approaches

Hierarchy is a useful way of modeling certain execution strategies in multi-agent systems. We have described in the previous section the use of hierarchies to add load-balancing capability to an application. This use of the concept of hierarchy (and its current implementation in Guest) can be refined even further. The distributed hierarchy we described in figure 10 is inefficient in terms of number of messages exchanged: for each message received by an agent inside the hierarchy, another one is generated to forward the incoming message to the root of the hierarchy. We describe now the inefficiency problem associated with distributed hierarchy when the number of agents is inferior to the number of servers, and a way to optimize the efficiency of a distributed hierarchy by combining it with a centralized hierarchy each time two or more agents are on the same server.

5.3.1 The problem: inefficiency of the distributed hierarchy

Suppose we have three servers, and ten agents in a hierarchy, distributed among these servers. We cannot use the highly efficient (in terms of messages exchanged) centralized hierarchy,because we have to use several servers, and the distributed hierarchy that we have to use is less efficient in terms of number of messages exchanged (due to the re-routing process). The solution we now describe is simply to combine both.

5.3.2 Combining both hierarchies

We introduce a new execution strategy, using both hierarchies in order to automatically minimize the number of messages exchanged inside a hierarchy. The strategy is simply to adapt the kind of link between two agents according to the servers they are on. The hierarchical link between two agents on the same server will be of the first kind (see 5.1), whereas the link between two agents on different servers will be of the second kind (see 5.2). When an agent in a hierarchy migrates, the type of link with its own parent is changed. Using both hierarchies, we are now able to minimize the number of messages exchanged, because we no longer have to reroute messages between agents on the same server. We discuss in the next section the use of a metamodel for automatic adaptation of the hierachy.

6 Self-adaptive agents by metamodeling

Until now, we presented ”base bricks”, which allow to modify, even in the course of execution, the control mechanisms and the functionalities associated with a *Guest* agent. However, the question of the layout of these bricks still remains unanswered. How to prevent, for example, two plug-ins, which are incompatible, to be used at the same time within the same agent? How to ensure that the use of one particular plug-in automatically leads to the installation of another plug-in, essential to the first one?

The first approach to solve these issues (that we could called ascendance) would consist in describing all these constraints in the form of rules. This approach could prohibit a certain number of configuration or add/remove some plug-ins associated with the agent, when certain particular conditions or actions are met.

However this solution seems too limited to be able to deal with complex and especially nonforeseeable configurations, such as the case with the Internet. This is why we prefer the descendant approach, based on the description of the agent in the form of the abstract models which can be instantiated *in fine*. Furthermore, such models allow a greater generalization to take place.

In order to more easily be able to manipulate these models, we should represent them in a uniform fashion. In other words, we should model them in a form which would become the *de facto* metamodel (18) of the agents. Our work is to research the mechanisms allowing to model the transformation from one model to another, and these changes, of course, can be defined by the same (static) metamodel. This metamodel should allow not only to describe all the possible transformations, but also to enable the correct transformation of one agent from a state corresponding to a given model to another state described by another model.

The principal of our model is the horizontal separation between the ”control” part of the agent and its ”function” part, based on the research in (21) (20) and validated in the development process of the platform Guest. The ”control” part is application-independant and presents in all or most agents. For example, the communication or the capacity of migration of an agent, or even the agent model itself, can be classified as ”control” part. Meanwhile, the capacity of learning or reasoning of an agent belongs to the ”function” part, because it is application-dependant or at least domain-dependant. If our model can support automatic adaptation to the ”control” part of agent, which is possible because this part is application-independant, the task of making an adaptive agent is much more easier and can be semi-automatically done. For example, we already have two models of hierarchical agent, one is most appropriate in the centralized scenario, while the other is preferable in the distributed context. The rest of agents (perception, deliberation, action) is unchanged. If the ”control” part of agents is adaptive, agents can therefore transparently ”switch” from one hierarchical model to the other model when the scenario changes from centralized to distributed one or vice versa.

The figure 11 presents our model of agent, which distingushes between ”control” and ”function” part, along with the concept of plug-in.

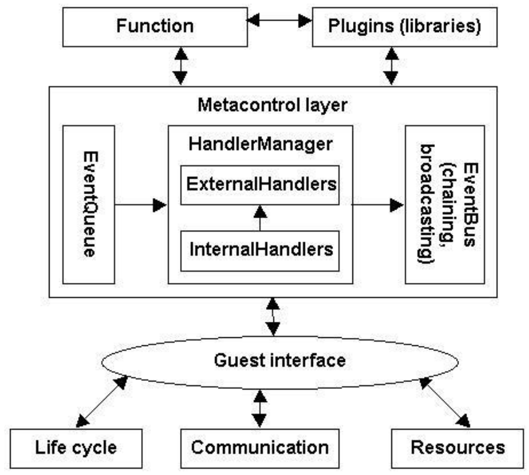

Figure 11: Metamodel of agent

In this model, the Metacontrol layer plays a crucial role, that is to connect different pieces of agent control, agent function and agent services (plug-ins) in the correct manner. Its architecture is inspired by the Java InfoBus technology (22), the semantic of the connection between different parts is captured in the types of events and in the handler’s type. We will demonstrate how this model works in the case of two types of hierarchical agent model.

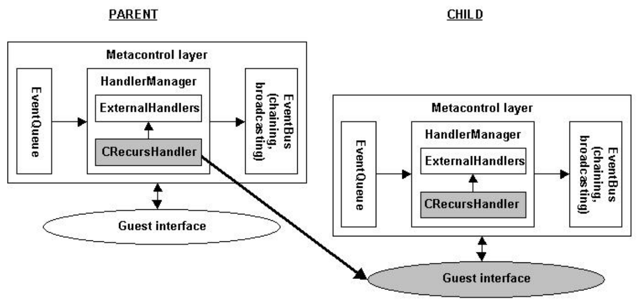

Figure 12: Centralized hierarchical agent

The model of centralized hierarchical agent is represented in the figure 12. In this model, the CRecursHandler is an internal handler, the bold line indicates that this handler is viewed by the child agent as an agent server.

The parent and child agent are in the same JVM and all the communications between parent and child are direct function call, thus very efficient. The CRecursHandler will control all the events related to the creation, destruction, migration and communication of the child agent.

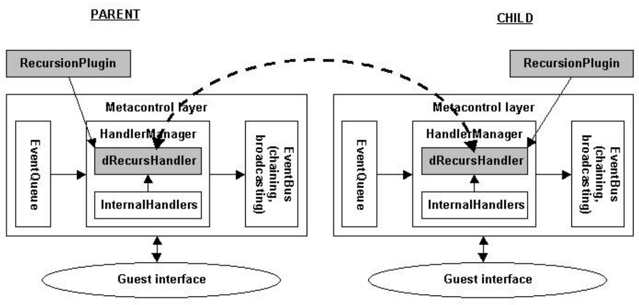

Figure 13: Distributed hierarchical agent

In the model of the distributed hierarchical agent, agent needs to use an external handler, DRecursHandler, which can be downloaded from a Web site and activated immediately after it is initialized. All the communications from/to the child agent are intercepted by this handler and rerouted to the parent agent to be authorized. The dotted link indicates that the rerouting is done through network communication.

As a consequence, the dynamic transformation from one model to another model is just simple as the activation/desactivation of the corresponding handler. model, works correctly to to their We are now working on the next step, that is to offer agents the faculty to automatically adapt themselves to their environment by using these transformations.

For example, when an agent, or even a branch, of the centralized hierarchy recognizes that the server on which it resides is overloaded, it wants to move to another less busy server, while assuring the integrity of the hierarchical organization by changing to the distributed hierarchy.

In order to do that, agent needs to use a third handler, which is external and handles events of type ”CPU overloaded”. Whenever an agent receives a signal from the server that it is overloaded, this handler will be activated and realize the transformation of the agent’s hierarchical model.

7 Conclusion

Today’s multiagent applications must deal with new constraints of open and complex environments, of which the Internet is the most typical example. The development of these applications requires that agents become generic and adaptive to their evolving environment, i.e. neither linked to specific execution platforms or to a unique standard, nor fixed to a limited set of models and services.

This is why, in this paper, we have presented a new model of uniform agents, called *Guest*, using the middleware approach. Based on a generic interface, *Guest* agent applications are able to execute the same way without worrying about the incompability between different agent platforms on which agents are running. To be adaptive, *Guest* agents can dynamically update their capabilities at runtime by adding or removing *on demand* plug-ins dedicated to specific tasks like the authorization of migration or the handle of secured communication. Our *Guest* agents also supports two organizational models: centralized and distributed hierarchy. Finally, we provide a metamodel which allows agents to automatically change their control mechanisms, such as their organizational model, to adapt to their environment.

Guest is already a full-fledged prototype with support for interoperability between Aglets, Corba, Jade, Grasshopper and Voyager platforms. The plug-ins framework is fully functional, both of the hierarchical models are implemented and currently tested. We are working on the implementation and validation of the presented metamodel. Applications of the *Guest* platform in the real world currently include a completed work in the industrial building management context.

8 Acknowledgements

This work is done at the CRIM laboratory and mainly founded by it. We have also to acknowledge the efforts of Nekrouf Ziani for HostCorba implementation and of Ivan Guentchev for his experiments on Jade, which were, as other students who worked on this project, supported by Laurent Magnin’s personal Research Grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Aglets Workbench, http://aglets.sourceforge.net/

[2] Control of Agent Based Systems, http://coabs.globalinfotek.com/

[3] Fipa, Foundation for Intelligent Physical Agents, http://www.fipa.org/

[4] Orbix by IONA, http://www.iona.com/ Concurrent Objects: Briot-Gasser Interview” http://www.lis.uiuc.edu/ gasser/AgentsAndObjects-07.html

[5] Grasshopper, Grasshopper by IKV++, http://www.grasshopper.de/

[6] The Java Agent DEvelopment Framework, http://jade.cselt.it/

[7] Joshi, A. and M. P. Singh (1999). Multiagent Systems on the Net. Communications of the ACM 42

(March 1999) pp. 39-49. Communications of the ACM 42 (March 1999) pp. 79-80. R. H. Guttman, et al., Agents That Buy and Sell, Communications of the ACM 42 (March 1999) pp. 81-91. First International Workshop on Mobile Agents for Telecommunication Applications (MATA 99), Ottawa, Canada, October 6-8, 1999

[8] Magnin L., ”Internet, environnement complexe pour agents situés, Proceedings of the Intelligence artificielle situe conference, Paris, October 25-26, 1999, pp.213-221.

[9] Magnin L. and Alikacem E.H., Guest: Multiplatform Generic Agents, Proceedings of the First International Workshop on Mobile Agents for Telecommunication Applications (MATA 99), Ottawa, Canada, October 6-8, 1999, pp. 507-514.

[10] Magnin L., et al., ”Our Guest agents are welcome to your agent platforms”, The 17th ACM Symposium on Applied Computing, Special Track on Agents, Interactions, Mobility, and Systems (AIMS), Madrid, Spain, March 10 - 14, 2002. l’organisation dans les SMA. Application la conception des Syst?mes Coopratifs Distribus et Ouverts, Ph.D. thesis, University Paris VI, Paris, 1998. Marketplace, Proceedings of Autonomous Agents, Seattle, ACM, May 1999.

[11] OMG, MASIF: Mobile Agent System Interoperability Facility, http://www.fokus.gmd.de/-research/cc/ima/masif/index.html

[12] Voyager, Voyager by ObjectSpace, http://www.objectspace.com/voyager/

[13] Wong, D., N. Paciorek, et al., Java-based Mobile Agents, Communications of the ACM 42 (March 1999) pp. 92-102.

[14] Wooldridge, M., Intelligent Agents, Multiagent Systems (ed. by G. Weiss, Cambridge, London, The MIT Press, 1999) pp. 27-78.

[15] Tjung D., Tsukamoto M. and Nishio S., A converter Approach for Mobile Agent System Integration: A Case of Aglet to Voyager, Proceedings of the First International Workshop on Mobile Agents for Telecommunication Applications (MATA 99), Ottawa, Canada, October 6-8, 1999, pp. 179-195.

[16] William Harrison and Harold Ossher, Subject-oriented programming: a critic of pure objects, in Proceedings of OOPSLA 93, Washington D.C., Sept. 26-Oct 1, 1993, pp. 411-428.

[17] Gregory Kiczales, John Lamping, Cristina Lopez, Aspect-Oriented Programming, in Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[18] Revault N., Sahraoui H.A., Blain G., Perrot, J.-F. ”A Metamodeling Technique: The MétaGen System”. Tools Europe proceedings, (1995).

[19] KQML, http://www.cs.umbc.edu/kqml

[20] Min-Young Yoo, Une approche componentielle pour la modélisation d’agent coopératifs et leur validation. Thèse de doctorat de l’université Paris VI, 2000.

[21] Jacques Ferber, Les systèmes multi-agents, InterEditions, 1996.

[22] http://java.sun.com/products/javabeans/infobus.