Supporting intelligent agents in a distributed environment: a COOL-based approach [1]

Bruno Dillenseger (+33 31 75 91 39, dillenseger@sept.fr),
François Bourdon (+33 31 75 91 19, bourdon@sept.fr),
SEPT SCE/ARC, 42 rue des Coutures, BP 6243, F-14066 Caen cedex, France.

1. in collaboration with LAIAC, Université de Caen.

KEYWORDS

object-oriented distributed system, agent programming, Computer Supported Cooperative Work, service negotiation, C++, COOL, Prolog.

ABSTRACT

The background of this article consists in a distributed object oriented approach for the development of logically and physically distributed applications in the field of Computer Supported Cooperative Work. The use of an object oriented distributed system (COOL) is a first step towards the implementation of solutions to these naturally distributed problems. But this support is insufficient to directly implement high-level interactions and behaviour: the entities have to adjust themselves to a dynamic system where services and servers appear, evolve, and disappear.

This article introduces both a model and a tool which are designed to enhance the entities cooperation ability and autonomous adaptability. The model springs from investigations in the field of Multi-Agent Systems and consists in a uniform representation of every resource of the system (hardware, software, user) by agents.

The implementation tool and associated developments are described in some detail. They are based on an enhanced Prolog interpreter integrating the main COOL features: various communication modes, active objects, migrating objects... Our Prolog-based approach is discussed and illustrated by the implementation of a distributed application and some generic multi-agent system structures and protocols for cooperation.

1. Introduction

By combining the distributed systems technology with an object-oriented approach, applications can be built as sets of independent and interacting entities. As these sets may be logically and physically distributed, this approach is specially relevant for many naturally distributed problem families. The SEPT[1] is typically involved in such research in the Computer Supported Cooperative Work field.

Applications about intelligent circulation of electronic documents [Deshayes 89] or cooperative editing revealed the necessity of an object-oriented distributed system. So, we adopted a Chorus micro-kernel based distributed system and specified COOL [Lea 93], an Object Oriented Layer integrated to C++.

Now, the key point consists in supporting advanced interactions and behaviour in order to increase the entities autonomy. They need to adapt to a dynamic distributed system where servers appear, disappear, evolve. The necessity of a high level cooperation implies a uniform view and interaction mode to be shared by every resource (software, hardware, users) that is available through the communication network.

Definitely, research in the Multi-Agent Systems field appear to be relevant to our needs. First, this leads to a multi-agent model proposal for the Human-Computer Cooperative Work. Then, we also need a Distributed Artificial Intelligence tool to implement this model. Although COOL/C++ is a powerful support for distribution, we lack a higher level language for matters such as knowledge representation, communication, reasoning and evolution capabilities. As a result, we developed our own tool by mixing Prolog and COOL in C++ objects.

1. joint research lab for La Poste and France Télécom.

2. The needs

2.1 Why an object-oriented distributed technology?

Autonomy increasingly becomes an essential element for the adaptation of the information systems to the unforeseeable evolution [Thiétart 93] of the office, whose environment is geographically distributed. This unforeseeable evolution forces the information systems to adjust themselves to the resulting organisational and operational changes.

We are currently in a transition period [Desreumaux 93]: depending on the visibility degree that the applications give about their functional features, users do not see systematically monolithic applications any longer. They tend to be increasingly aware of a system of interacting objects instead. The role of each object consists in the available autonomous services that are likely to help the user to manage his/her tasks.

This visibility becomes essential as elementary notions of costs control, implied by the autonomy principle, have to be taken into account by the users, specially in open distributed environments. Then, users see the information system as a natural and coherent interface for the tasks they have to achieve. It has to be possible for them to use it in order to:

- know/decide what they have to do;
- seek/archive information;
- negotiate, regroup;
- control/execute a task...

2.2 Why advanced behaviour and interactions?

Users need a guide to find the most relevant service among the profusion of autonomous and available services which appear, disappear and evolve within the office. This help requires a great deal of interoperability and implies that the computer entities rely on a standard interface language which is supposed to create a minimal semantic universe. All the agents of the system that are likely to communicate with each other have to share this common semantic universe. This *integrator principle* was introduced by Jean Erceau and Michel Barat in [Barat 93].

Beyond the functional representation of the offered and/or requested services, this search has to feature

- negotiation possibilities to make the requests suitable for the “market” tender,
- learning (memory) capabilities to avoid systematic and useless quests, and
- sophisticated communications allowing manual (human) actions in case of hard negotiations.

3. Towards a multi-agent approach

3.1 From the object-oriented distributed systems to the multi-agent systems

The agent concept comes from the Distributed Artificial Intelligence and Multi-Agent Systems field. Among the abundance of theoretical issues, such as cooperation matters [Smith 80], distributed plan generation and execution [Lesser 87], knowledge and beliefs representation [Lefèvre 93], one particular problem appears to be fundamental: when an agent has to complete a task it can not solve on its own, how is it going to find the most accurate cooperating agent?

This issue is also essential in any object-oriented distributed system, which is a particular multi-agent system. When a client object intends to run a remote object’s method, it has to look up in the network to find the most accurate server. To help the client, some special structures are dedicated to guiding or even making the link between the client and the server. For instance, the trader concept [ODP 93] is used in the Open Distributed Processing architecture, such as the ANSA platform [Deschrevel 93], to meet this need. It consists in a services directory which may be built following the X.500 distributed directory recommendations. The trader is used by the server objects, to declare the exported services, and by the client objects, to seek the relevant servers.

But the accuracy of the agent concept is not bound to the server quest in a distributed environment. It can also be justified by its recent software engineering aspect through the search of an agent-oriented method [Ferber 94]. The *agent-oriented programming* concept is presented in [ODP 93] as a specialization of object-oriented programming: the *mental state* of the agent takes into account the notions of belief, decision, obligation; the basic types of communication come from the speech-act theory (inform, request...).

3.2 A multi-agent model

3.2.1 Making a uniform system

The first step of our “integrator principle” relies on a homogeneous vision of the office information system. Roughly speaking, the network links up a set of machines (*sites*) which hold a few *resources*. These resources are likely to appear, disappear or evolve: application software (e.g. database, editors), migrating application entities (e.g. circulating electronic documents), peripheral hardware (e.g. printers), connected users. Each resource holds a particular set of available *services* which are associated with some specific *attributes*:

- The *structural attributes* are static, or weakly dynamic when the service is upgraded. They specify a particular implementation of a service. For instance, they may consist in the printer brand, the printing process and speed.
- The *conjunctural attributes* are typically dynamic because they depend on the current context. For instance, the context of a printing service could consist in the size of the printing queue, the unavailability for maintenance matters, the number of remaining paper sheets.

When these resources need some other resources' services, they are confronted with the basic problem: they have to find the most suitable service and therefore the accurate server. This implies coping with two issues:

(1) being able to contact the resources although the system is dynamic;
(2) being able to negotiate as precisely as possible to get the most accurate service.

First, as both points require a great deal of interoperability, the resources have to be integrated into a uniform representation of the overall system. Thus, each resource is associated with an object whose role consists in representing it for the other resources. The projection of the system entities results in the creation of a uniform layer (cf. figure 1) where the servers quests and the services negotiations occur.

Secondly, point (1) introduces the need for special structures and associated protocols whose aim is to provide servers with clients and clients with servers. The first idea is the services directory and the implementation of trader objects but other structures could be imagined (e.g. analogy with classified advertisements, yellow pages). Any agent, at creation time, has to know at least one

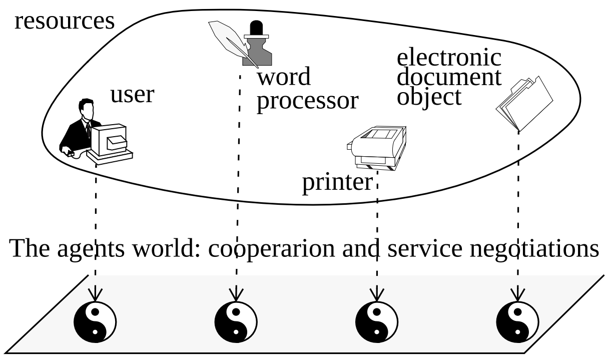

resources
user
word processor
electronic document object
printer
The agents world: cooperarion and service negotiations

figure 1: making a uniform system representation

structure of this kind, as well as its associated usage protocols.

3.2.2 From the object to the agent

Our approach will be in vain unless the objects we are introducing hold some information about their resources and the provided services, as well as the overall system and some particular resources. Moreover, these objects have to know some basic protocols and share a common negotiation language. Such an object, which behaves like a unique controller and representative of a resource and its associated services, is called an *agent*. It can be considered as an active interface for a resource:

- it acts like a *client* when it looks for servers and negotiates services for its resource;
- it behaves like a *server* when it tries to cooperate with the other clients.

Thus, the behaviour of the agent consists in permanently listening to the internal (i.e. from the resource it represents) and external (i.e. from the other resources) requests. Moreover, it may follow a private activity such as supervising some tasks it is responsible for, or consulting the system (i.e. other agents) and organising its knowledge for learning matters.

If the object concept is currently rather clear and widely shared, it is not the case as far as the agent notion is concerned. In fact, this word is often used in various situations, as if there existed an immanent definition, but without any real consensus. Moreover, a confusion is often introduced with the actor notion.

The agent concept is not mature yet and there is no universal agent model. This context allows a great variety in the multi-agent research field which will benefit from this diversity. But it also results in some groping and communication difficulties between research workers. It is the reason why we will now define more precisely our point of view.

The actor term may have two meanings: either it is a generic notion which deals with any active entity playing a role in a system, or it is a reference to the actor model [Agha 88] for parallel computing and its resulting actor languages and systems. The former case is near our conceptual view of agents. In the latter case, the actor model can be considered only as a special way of implementing agents.

Compared to the object, the agent gets more autonomy and encapsulation: it follows its own independent activity, guided by its own will. It is also characterised either by high level or emergent (cf. artificial life) communication protocols (). Finally, it is associated with the skill notion. In our model, an agent skill is represented by the set of services it offers, refined by some attributes.

4. Implementing the model

4.1 An object-oriented distributed system: COOL

The Chorus Object Oriented Layer was specified by SEPT in order to allow the actual distribution of applications such as the Intelligent Circulation of Distributed Folders. This application had already been conceived an object oriented way and had mostly been implemented in C++. COOL v1 [Lea 91] mainly embeds the communication features of the Chorus distributed system micro-kernel into an object oriented layer, available via C++ as a COOL class (figure 2).

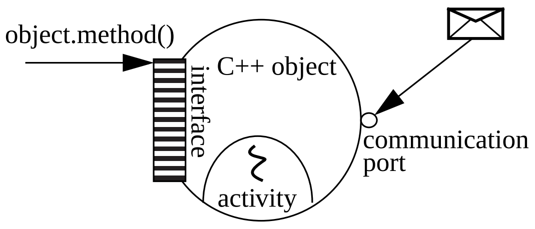


figure 2: COOL as an extension of an object model

Any object from this class possesses blocking (call/reply) and non-blocking (send/receive) communication methods, and communication groups management methods. Such an object is able to migrate from site to site through the network, to synchronously call another COOL object's method (in the same address space with COOL v1, or in a transparent distributed way with COOL v2 which is being released). It may own a mail-box and may be either passive or execute its own activity.

Various facilities are also available such as a semaphore object class and a persistence management. Finally, the communication system is completed with a distributed symbolic naming service which allows the objects to register their mail-box address using chosen names.

4.2 Introduction of a high level language: Prolog

The use of C++ is justified by its object-orientedness and its efficiency inherited from C, the UNIX systems traditional language. But it appears to be limited and not to meet some of our needs: knowledge representation, high level negotiation language, “intelligent” and adaptive behaviours... The steps we performed towards Distributed Artificial Intelligence with our multi-agent approach, called our attention to the traditional languages of this field. As a result, as we could not do without the COOL facilities, we decided to integrate one of these languages in a C++ object.

Prolog was chosen for several reasons. From a general point of view, Prolog is a high level interpreted language, featuring a uniform data and program representation. Hence, by adding communication predicates, Prolog programs are able to exchange knowledge as well as know-how. Within our model, this facility allows the propagation of new procedures or the update of old ones. The agents can dynamically upgrade themselves(2), without interrupting their functioning. Besides, this technique is also suitable in a heterogeneous environment.

Moreover, Prolog gives a common communication language base which is shared by every agent. Originally conceived as a natural language analysis tool, Prolog is a very efficient support for the definition and interpretation of new languages that may be designed for our specific needs (e.g. constraints expression language for service negotiations). At last, our agents benefit from the powerful unification engine in order to implement some more or less clever reasoning.

A multi-agent model in the Human-Computer Cooperative Work field has been carried out already in the IMAGINE [Haugeneder] Esprit project. It has resulted in the development of a parallel Prolog based multi-agent environment and tool-box. The approach is very near ours: mixing

2. a Prolog program can consult and edit itself

both the Computer Supported Cooperative Work research, which aims at assisting the cooperation between human agents by the use of computer resources, and the DAI anthropomorphic cooperation models, so as to make it easier for the users and the applications to interact and cooperate. But this work does not benefit from the support of an object-oriented distributed system such as COOL. Nevertheless, further investigations about this project will be necessary.

4.3 Prolog Upgrade for Multi-Agent system

4.3.1 Overview

PUMA results from the integration of a Prolog interpreter into a COOL object. This integration is double:

- Prolog is integrated into COOL, which means that a C++/PUMA object owns a Prolog kernel as well as the associated control interface.
- COOL is integrated into Prolog, as the COOL features are embedded into Prolog communication predicates

This double faced integration results in the ability for the developer to conceive compound objects whose activity is a C++ program making Prolog calls from time to time, or a Prolog program calling some C++ procedures, or the combination of both. The methods and predicates the C++/Prolog interactions rely on, are presented in figure 3. From

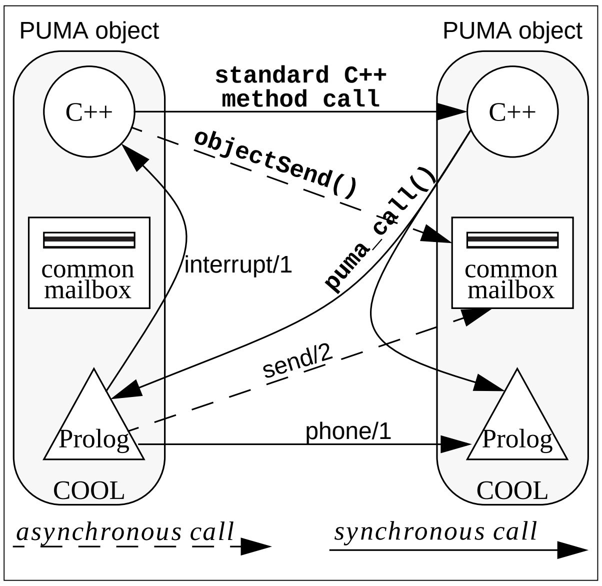


figure 3: **C++ (COOL)** - Prolog (PUMA) communications

a general point of view, these features are interesting simply because of the fact that these languages are suited to different tasks (this remark partly explains the agents' symbols chosen in figure 1).

4.3.2 The PUMA class

The **PUMA** class holds the complete access and control interface to the Prolog kernel. The Prolog dialect features new predicates embedding many COOL functions. Communication predicates include asynchronous messages delivery with group facilities. The synchronous call predicate makes it possible for a PUMA program to access to another PUMA object's prolog kernel. PUMA objects may also create other COOL deriving objects.

Migration is also an interesting PUMA feature. As a matter of fact, COOL doesn't solve the complex activity migration issue: the COOL object activity is restarted from zero after each migration. But, with a PUMA object, migration is transparent for any Prolog program that calls the migrate/1 predicate, thanks to the Prolog state save/restore feature.

The **PUMA** constructor method takes two arguments: the symbolic name of the object for the naming service and the name of the initial Prolog file to be loaded. As this class does not define any activity, it is typically bound to be used by operational classes through inheritance, but it is also likely to be directly used in two ways:

- actions can be performed by defining the PUMA creation reflex in the initial Prolog file;
- otherwise, the **PUMA** object can be used as a passive Prolog server by other **COOL** deriving objects.

4.3.3 The PUMA derived classes (figure 4)

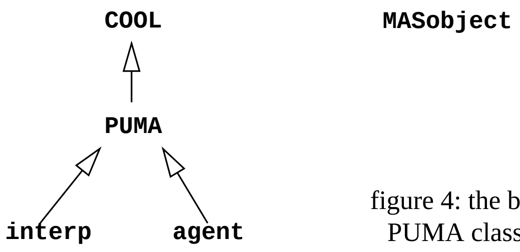


figure 4: the basic PUMA classes

The **interp** class defines an interactive interpreter activity and its instances exactly behave like Prolog interpreters with an extended dialect. This class is quite useful for debugging, observing, and perfecting **PUMA** objects. To a certain extent, it can be also useful for any **COOL** deriving object as the added predicates embed many COOL functions.

The **agent** class defines a generic agent activity (see figure 5) which consists in repeated and

THE C++ LEVEL ACTIVITY:

```
void agent::main()
{
    puma_init();
    do {
        puma_P();
        if (pstate != P_END)
            puma_call(“activity.”);
        puma_V();
    } while (pstate != P_END);
    puma_exit();
}
```

THE PROLOG LEVEL ACTIVITY STEP:

```
activity :-
    read_one_message,
    my_own_activity_step.
read_one_message :-
    receive(Msg),
    process_msg(Msg).
read_one_message.
process_msg(call(Goal)) :-  % extendable message
    Goal.                   % processing procedure
```

figure 5: PUMA **agent** activity scheme

continuous calls to a particular predicate representing the elementary activity step. This predicate has to be defined in the initial Prolog file. The step typically consists in reading and processing one message -if any- from the mail-box. There may be also an individual task to run but each step should be as fast as possible:

- to avoid a mail-box overflow,
- to keep the Prolog kernel available enough for external synchronous calls (the Prolog kernel has to be invoked through mutual exclusion).

If an agent has to perform a long lasting task, it may create another agent whose activity is dedicated to this task.

At last, any **COOL** derived class can be automatically and almost transparently linked to a private **PUMA** agent by inheriting from the **MASobject** class. The instance of an **MASobject** deriving class holds attributes recording the agent mail-box address for asynchronous messages and the agent’s object address for synchronous calls (i.e. standard C++ method calls). Moreover, the agent automatically follows the object it is associated with during its migrations and it may execute some programmable creation, migration and destruction Prolog reflexes. The use of this class is related to the projection principle of the system resources in a homogenous agents layer.

5. Implementation example

5.1 A multi-agent meeting room reservation system

We will now describe the implementation of our model in an actual and simple case, using the tools we have just presented. This multi-agent reservation system uses two main agent types: the room agents and the user agents.

Each meeting room is exclusively represented by a private agent which maintains some information about its structural attributes (equipment, size, place...) and conjunctural attributes (reservations planning, scheduled repairs...). The agent is the complete interface to the typical meeting room services: reservation and cancellation.

When a user wants to reserve a meeting room, the corresponding service can be invoked through its user agent which in turn contacts the right servers (i.e. the room agents). Hence, this agent is not only in charge of representing the user in the system, but it is also the entry point towards the whole set of available services.

The user and room agents are *permanent* because they represent some resources of the system and their appearing and disappearing is a rare occurrence. The *temporary* agents are dynamically created in order to execute one particular task. This kind of agent is not associated with a resource and disappears as soon as its mission is completed.

5.2 The system components

5.2.1 The permanent agents

As a permanent agent represents a resource, it has to hold a minimal administration interface: the user who created such an agent has to be able to move it onto another site, or to kill it. The agent + control interface compound entity is implemented with the **MASobject** deriving **interface** class. This results in the definition of two parallel activities: the agent activity and the interface activity, typically waiting for an event from the user.

The minimal control interface also offers the user the possibility of invoking the agent in order to access the local or system wide available services. Now, each resource type is associated to a specific set of services. Each service may be either *internal* or *external*, according to the ability of the agent to execute it by itself or by invoking another agent.

As some services are likely to evolve or appear dynamically in the system, the agents may have to upgrade their services without disturbing their functioning. Consequently, the use of Prolog to define the various services seems to be quite a convenient approach for declarative programming and dynamic concerns.

As a result, in our system, the type of an entity is defined in relation to two structures:

- C++ inheritance trees;
- sequence of loaded Prolog files.

By loading a particular Prolog file, each permanent agent learns two internal services: the agent attributes viewing service and editing service. Room agents, by adding a specific Prolog file, gather new internal services such as planning consulting, room reservation and cancellation. As far as user agents are concerned, they load another specific file which defines an internal memento service, but also two external services: room reservation and reservation cancellation. Thanks to this Prolog coded representation, agents are able to modify their services or learn some new services dynamically and then propose them to the interface.

5.2.2 Temporary agents

A temporary agent does not need any control interface: it is created by another agent to perform a particular mission and it disappears when this mission is completed or when its creator tells it to do so. It is made of an **`agent`** class object and a specific sequence of Prolog files. There are many types of temporary agents because they are specialized in the field of the mission they must carry out.

In our system, we use some temporary agents when it takes a long time for a service to be executed, as it is the case in the user agent memento service example. Although it is an internal service, since the agent manages autonomously with it, an auxiliary agent is created to run the service each time it is called. This way, we avoid disturbing the agent functioning and its permanent listening to the system with a user-interactive service.

The temporary agent runs the service and opens a dialogue interface with the user. But, instead of copying the memento data into its own database, it directly gets each datum in the user agent database each time it needs it. Thus, the information that the user consults is permanently up-to-date, even if he/she keeps the service running for a long time. He/she may also run several times the same service in parallel if he/she wants. Moreover, this way of accessing information is quite efficient and not costly because it uses a method call between two objects in the same address space. The memento service temporary agent disappears if the user or his/her agent tells it to do so.

This technique allows the parallel execution of several local internal services, without duplicating information. It prevents the system from redundancy and the resulting waste of memory and coherence maintenance problem. But temporary agents are also used to run migrating services, as illustrated by the protocol in 5.3.

5.2.3 The negotiation language

Neither the uniform representation of the system resources with client/server agents, nor the availability of a common language (Prolog) for the agents are enough to create a complete *integrator principle*. Some semantics must be shared by every agent in the system; otherwise, any communication and then any cooperation attempt is illusory.

Consequently, as the service abstraction is the key element of cooperation in our system, the agents have to be able to precisely describe the service they are expecting. To face this issue, our agents are given:

- a set of symbols, representing particular services (e.g. print, reserve room) and particular server types (e.g. printer, room);
- a set of unique attribute symbols (e.g. resolution, surface) for a given service field;
- a language to put some constraints on the service attributes into words.

Our constraints language makes it possible to specify whether or not an attribute belongs to a set or an interval. The generic constraints solving engine we implemented can be enriched by each agent to solve special cases by introducing default constraints and new domain dependant rules.

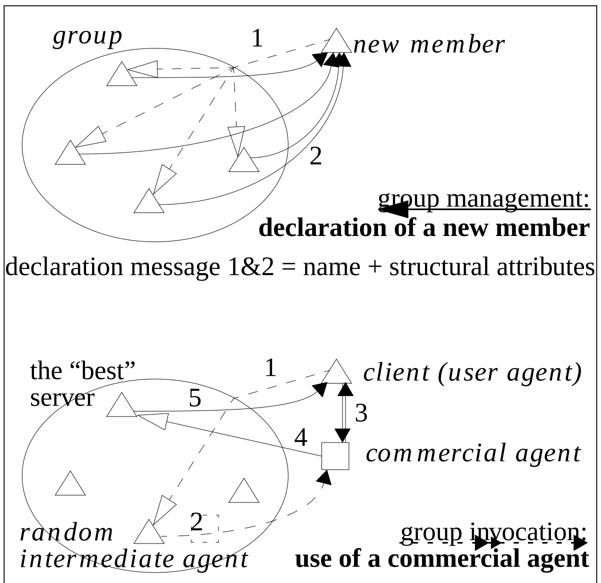
Moreover, the language takes into account a satisfaction notion which enriches the “or” notion with ordered preferences. The satisfaction value is computed from a Prolog procedure and the values of some attributes. Thus, this language is quite expressive and allows the formulation of requests as complex as “reserve a room for 10 persons, with an overhead projector, from 8:30 AM to 11:0 AM, preferably on the first monday of January 95, otherwise on another monday of January 95”.

5.3 A cooperation protocol example

5.3.1 Detailed description

The cooperation protocols our system is based on makes use of the Chorus group communication features. Our cooperation structure is associated with two protocols: the management of the structure and its use by the clients when they are looking for a server (cf. figure 6). We tried to avoid the drawbacks of some costly protocols such as the contract net protocol [Smith 80]:

- communication network overload;
- agents mail-boxes overflow;
- agents message processing overhead.


group
1
new member
2
group management:
declaration of a new member
declaration message 1&2 = name + structural attributes
the "best" server
1
5
client (user agent)
3
4
commercial agent
2
random intermediate agent
group invocation:
use of a commercial agent

1: uncomplete service request;
2: creation and migration of the commercial agent;
3: interactive set-up of a complete service request;
4: complete service request;
5: service contract.

figure 6: special group protocols

The structure we implemented consists in making the agents from a given field maintain a kind of an artisan group. This group aims at satisfying the clients instead of competing. As soon as a room agent is created in the system, it broadcasts a declaration message to the room group. Then, each member of the group returns a message to declare their belonging to the group[(3)]. Since each declaration message contains the structural attributes (surface, equipment, place...) of the sender agent, each room agent permanently knows the other room agents and their associated structural attributes. Conjunctural attributes (e.g. reservations planning) are not transmitted to avoid misusing the group broadcast facility.

When a user wants to reserve a meeting room, he/she invokes his/her agent and chooses the room reservation service. This is an external service as the user agent can not complete it by itself. Moreover, the agent is not able to put the constraints for this service into words since it is not an expert in the room reservation field. Therefore, it has to contact a room agent and tell it the kind of service it is expecting but without any precision.

Now, our structure shows its attractiveness: the user agent does not broadcast its request to the room group but uses the Chorus so-called *functional* mode. It consists in sending a unique message to a random member of the destination group. Then, the room agent replies by creating a temporary agent, a kind of a *commercial agent*, whose mission is to manage with the client's request. Thus, the physical and logical distribution of the problem is respected: the complete request for a given service is put into words by an expert of the concerned field, created by a specialised artisan from a remote site.

As soon as the commercial agent is created, it copies the room agents list with their associated structural attributes by directly accessing to its creator's Prolog database. Then, it migrates to the client site and opens a window so as to interactively establish a complete service request with the user. It asks him/her for the main constraints but also directly gets information from the user agent's Prolog database. Since the commercial agent knows the structural attributes of every room, it is able to make a list with the relevant rooms. Afterwards, the list is ordered by examining the satisfaction criterion[(4)] for each selected server.

Finally, the commercial agent successively contacts the potential servers as long as it receives refusal answers for unavailability matters. When a server accepts the request, the commercial agent gives the contract to the user agent. The reservation information is transmitted to the user and recorded for a future use (e.g. memento service). Once its mission is completed, the commercial agent dies.

3. A similar protocol ("hello protocol") is used by the Chorus micro-kernel when a site is appearing (i.e. booting) on the network in order to guess its incarnation number, which is necessary for unique identifiers generation.

4. The satisfaction criterion defaults to a best fit between the needs of the client and the equipment and size of the room.

5.3.2 Justifications and comments

The accuracy of this structure and associated protocols relies on a few assumptions which are met in our example. On the one hand, it is supposed that the service quest is a usual phenomenon whereas a server appearing, disappearing or modification is an unusual phenomenon. As a matter of fact, the server seek protocol is clearly optimized (the communications are spread over time with few messages and no broadcast) whereas the internal group management is rather a heavy duty. On the other hand, there should not be too many group members to avoid a penalizing server declaration overhead.

Practically, we can try to evaluate the performance of this structure. As far as the management protocol is concerned, the declaration of the (n+1)th member in a n-members group causes 2n messages to be sent. n messages are sent in the same broadcast and n other ones are individually sent. Moreover, n of these messages are mailed in the same mailbox. From a general point of view, as we allow data replication, our assumptions aim at making sure that consistency maintenance is not too costly.

The evaluation of the server seek protocol is harder to figure out because it is context dependant, but we can try to compare it to the typical contract net protocol. In a formal way, table 1 tries to figure out the communication costs for both protocols[5] (n is the number of group members). We first notice that the commercial agent migration, which is costly but used only once, prevents the negotiation phase from many messages that would have been caused by a remote interaction between the user and the commercial agent. Moreover, the fact that the commercial and the client agents are in the same address space allows fast and communication system load independent interactions. It also allows the commercial agent to get information from the user agent without bothering the user.

5. Of course, the accuracy of this comparison depends on the assumptions we described which make the punctual group management communications insignificant.

contract net	group
the best	
- n tender requests - 1 tender - 1 contract - 1 execution report	- 1 tender request - 1 migration - 1 service request - 1 execution report
n+3 messages	**3 messages, 1 migration**
the worst	
- n tender requests - n tenders - n contracts - n rejections[a]	- 1 tender request - 1 migration - n service requests - n rejections
4n messages	**1+2n messages, 1 migration**
“average”	
- n tender requests - n/2 tenders - n/4 contracts - n/4 messages (rejections + 1 report)	- 1 tender request - 1 migration - n/4 service requests - n/4 messages (rejections + 1 report)
2n messages	**1+n/2 messages, 1 migration**

table 1 : evaluation of the basic CNet protocol and our group protocol

a. The server may accept contracts during the delay between its sending the (non guaranteed) tender and its receiving the contract, which causes it to reject the contract.

We insist on the differences between the migration and the message sending, which may not be quite obvious as an interpreted program can move within a message. We distinguish:

- passive messages, which contain simple data;
- active messages, whose content is an interpretable program that can be executed by the receiver, even in a heterogeneous system;
- possibly active binary object migration, only in homogeneous systems.

We choose the migration technique because our objects are two-headed: a binary object and a Prolog program. It could be possible to migrate the Prolog part through messages but, since there is a message size limit, it would not be convenient. Moreover, the binary part of the object has to migrate anyway, as it contains its own data and also because its class text segment is not necessarily present in the destination site.

5.4 Practical concerns

The system runs on three Chorus micro-kernel based PC[6] and shows the great efficiency of the group protocol and of the communication system, integrated in the micro-kernel. Nevertheless, our system needs a lot of resources, specially because of the many parallel activities. It was also tested on a stronger hardware configuration (SPARCstation 10) which offers more power but introduces a serious communication bottleneck, as it still runs a Chorus simulator instead of a native Chorus micro-kernel based system.

The size issue for the PUMA objects is rather an interesting point and we will now explain it in some detail. The text and data segments of the **`agent`** and **`interp`** classes (cf. 4.3.3) do not exceed 17K, because the binary part of the Prolog interpreter is integrated into the address spaces on each site. Each address space also includes the COOL library and its text and data size is about 120K. Dynamic memory allocation essentially springs from the Prolog interpreter whose code is partly written in Prolog. Once the Prolog files are loaded, the Prolog database fills about 100K.

So, it is important to find the balance and make the components of our system fit one another: COOL, the hardware configuration and the integrated language. For instance, we note that each PUMA object carries about 60K for the Prolog part of the interpreter whereas this part is identical in every object. It could be shared by all the objects in a given address space if it was written in a compiled language.

6. Conclusion

6.1 The current results

This article proposed an agent-oriented integration and cooperation rough model for the resources (software, hardware, human) which are reachable through an office information system. In the Computer Supported Cooperative Work and Human Computer Cooperative Work context, our purpose is to make it easier for servers and clients to meet each other as well as to enrich the negotiation phase:

- in order to help the client to find quickly the most accurate service for its needs,
- although overall service tender of the system is typically dynamic,
- without saturating the communication system.

Once having dealt with the need of an object-oriented distributed technology, we have presented the various layers and tools (COOL, PUMA and deriving classes, Prolog files, constraints solving) that we have chosen or built to implement our model. A pragmatic system example (multi-agent meeting room reservations) illustrates our approach. We also introduced a special group structure and its associated use and management protocols which aim at making it quick and easy for a client to find the most relevant server. This structure is designed for frequently invoked services which are executed by few and weakly dynamic servers (e.g. meeting rooms, printers, facsimile...)

But there still remains some work to be done on the model. First of all, we did not deal with the entities and system granularity issue. As far as the agents are concerned, their granularity is rather high, but it could be imagined to conceive each agent as a multi-agent system in order to more rigorously shape our system development. The granularity of the system is also a very important point because our structures efficiency depends on the size of the physical searching space. Our model typically fits a Local Area Network with local services, semantics and rules. Then, we may think about a special structure type which allows enhanced interactions between several LAN through a Metropolitan Area Network. Similarly, but with tough translation and semantics issues (see an approach in [Lee 89]), there should be wide area network cooperation agents in each LAN or MAN.

Our model would also benefit from a higher level of shared semantics (meta representation) for agent adaptability and domains interconnection matters. At last, since many protocols, auxiliary structures, message types, agent types are likely to be implemented in an overall multi-agent integrated information system, there lacks an agent oriented computer aided software engineering tool which takes into account the Prolog part.

6.2 Future prospects

6.2.1 The multi-agent approach

The multi-agent approach seems to be full of promise and we envisage the development of other applications, structures and protocols.

6. These computers are 66 MHz 486 PC running CHORUS/Fusion for SCO UNIX.

The first application is an “intelligent” mailing service allowing the recipients to be specified according to some criteria. Within our model, it consists in finding the agent(s) whose attributes match a constraints set. We would like to use a kind of a directory agents structure that would be distributed. The invocation of this structure will be a simple service request: “find an agent that matches these constraints...”. There also should be some local optimization with site agents, designed according to the fact that most of the agents from one site generally deals with a limited number of remote agents.

Another application consists in a multi-agent implementation of CIDRE, our intelligent circulation of distributed folders application, which is the historical origin of our multi-agent approach. This application is physically and logically distributed by nature and is likely to benefit from our model, tools, structures and protocols:

- A declarative circulation schema representation makes it possible for an administrator to intervene during a circulation and modify it.
- Prolog/PUMA is a good support for an on-board “intelligent” schema execution engine (time, cost or quality optimization), and an on-board jamming management expert system (so-called “exception handling”).
- The user agents may know enough information to be able to automatically propose the answers when the user has to fill some particular fields in an electronic form (e.g. address, name, birthday, phone number, department and role in the organization).
- The agents may hold the unavailability and delegation information...

6.2.2 The scripting language approach

We mostly presented PUMA as a multi-agent system implementation tool, but its attractiveness is not limited to this concern. A PUMA object also gives an access to an object-oriented distributed system through an interpreted language (cf. `interp` class in 4.3.3). Thus, PUMA is a step towards the *scripting language* principle, which is more and more popular among:

- the object-oriented distributed systems developers (shell, debugger...);
- the programmers who could quickly build new applications;
- the users who would like to easily tell autonomous entities (“agents”) to perform some more or less complex tasks.

PUMA is only a first step which was not originally designed for this approach. SEPT, in collaboration with Chorus systèmes, are currently amplifying their work in this field: COOL is being thoroughly upgraded (version 2 is to be completely released soon) and some studies try to specify the most accurate scripting language.

Acknowledgments

This work was led in collaboration with Patrice Enjalbert from LAIAC - Université de Caen.

Special dedication to Christophe “tof” Trompette, the originator of the first PUMA interpreter.

References

[Agha 88] Gul Agha. *Actors, A Model of Concurrent Computation in Distributed Systems.* MIT Press 1988

[Barat 93] Michel Barat, Jean Erceau. *Utilité et utilisation d’un principe intégrateur dans un outil de conception de systèmes complexes multi-experts.* 2ème congrès européen de systémique, vol. III pp 860-869. Prague, october 1993.

[Deschrevel 93] Jean-Pierre Deschrevel. *The ANSA Model for Trading and Federation.* Architecture Report, ANSA phase III, Architecture Projects Management Limited 1993

[Deshayes 89] Jean-Marc Deshayes, Vadim Abrossimov, Rodger Lea. *The CIDRE distributed object system based on Chorus.* Proc. of TOOLS’89.

[Desreumaux 93] Marc Desreumaux. *Pourquoi les entreprises ont-elles besoin de systèmes d’information flexibles ?* Tome 7, 1er congrès biennal AFCET, Versailles, june 1993.

[Ferber 94] Jacques Ferber. *BRIC : essai de mise en perspective d’une méthodologie multi-agent.* Journée d’étude AFCET “méthodes orientées agents”, Paris, septembre 14th, 1994

[Haugeneder] Hans Haugeneder. *IMAGINE Final Project Report.* ESPRIT project 5362, IMAGINE Consortium

[Lea 93] Rodger Lea, Christian Jacquemot, Eric Pillevesse. *COOL: system support for distributed programming.* Communications of the ACM, Vol.36, No.9, 1993.

[Lea 91] Rodger Lea, James Weightman. *Supporting object oriented languages in a distributed environment: The COOL approach.* Proc. of TOOLS USA ‘91, Santa Barbara, CA, july 1991

[Lee 89] Jintae Lee, Thomas W. Malone. *How can groups communicate when they use different languages? Translating between partially shared type hierarchies.* technical report CCSTR#103, SSM WP #3076-89-MS, MIT, September 1989

[Lefèvre 93] Claire Lefèvre, Claire Beyssade. *Système multi-agents et modalités épistémiques.* Premières journées francophones IAD & SMA, Toulouse 1993

[Lesser 87] V. Lesser, D. Corkhill. *Distributed problem solving.* Encyclopedia of Artificial Intelligence, Vol. 2 (1987), 245-251

[ODP 93] *Basic Reference Model of Open Distributed Processing: non-normative (descriptive) specification of trader.* ISO/IEC JTC 1/SC21 WG7 N, Standards Australia 1993

[Shoham 93] Yoav Shoham. *Agent-oriented programming.* Artificial Intelligence 60 (1993), Elsevier, 51-92

[Smith 80] R.G. Smith. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.* IEEE transactions on computers Vol. C-29, No. 12 (december 1980), 1104-1113

[Thiétart 93] R.A. Thiétart *Ordre et Chaos dans les Organisations.* Journée d’étude AFCET “Les Systèmes d’Information, Autonomie et Chaos”, Paris, november 24th, 1993.