Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications

PDPTA'98

Volume I

Editor:
H. R. Arabnia

Associate Editors:
Rajkumar Buyya
R. A. Olsson
R. Pandey
X-H Sun

Las Vegas, Nevada, USA
July 13 - 16, 1998
CSREA Press

ARéVi : A Distributed Virtual Reality Toolkit Based on an Oriented Multiagent Language

V. Rodin, S. Morvan, A. Nédélec
Ecole Nationale d'Ingénieurs de Brest
Laboratoire d'Informatique Industrielle
Technopôle Brest-Iroise, CP 15
29608 Brest Cedex, France
e-mails:{rodin,morvan,nedelec}@enib.fr

Abstract *ARéVi is a Distributed Virtual Reality Toolkit. ARéVi is built on our dynamic multiagent language, oRis. This dynamic language is particularly well adapted to the creation of co-operative applications. In fact, during an ARéVi session, this language allows the addition of new entities, and the modification of the behavior of an entity or a whole entity family. The modifications may either be introduced by the user or received by the network. In this article, we mainly present the co-operative characteristics of ARéVi which allow several applications (ARéVi sessions) to share the same 3D universe.*

Keywords: Distributed Virtual Reality, Dynamic Multiagent Languages, Java Network Communication, Mobile Agents.

1 Introduction

ARéVi (Atelier de Réalité Virtuelle) is a platform for the development of applications in Distributed Virtual Reality. It is developed in C++ and its kernel is independent from 3D libraries. It may be used for rapid application development in distribution of "real time" 3D visualization, with or without the immersion of human operators. It may also be used for co-operative work applications or distributed applications integrating universes inhabited by agents whose behaviors are more or less complex. The behaviors of those agents are very precisely and simply described in *oRis* [1, 2], an agent-oriented object language.

The main two aspects of *ARéVi* are the agent distribution and the co-operation between agents. For example, *ARéVi* allows the creation of distributed simulations where the different agents are spread out on different nodes of a network. This particularly allows to:

- distribute the agent behavior calculations on several computers,
- make several human agents co-operate, each of them driving a particular application.

Besides, co-operation between software agents is possible, whether the agents are distributed or not. On each site participating to a distributed co-operative session of *ARéVi*, one can make "real agents" evolve among agent proxies or "ghosts" whose behavior is simpler than their "real agents" and are located on distants sites participating to the session.

At any time, an *ARéVi* application may either join or leave a collective simulation, bringing along or taking away its own agents. Eventually, when leaving a simulation, it may move its agents towards other applications which participate to the simulation. In fact, to move an agent, it is only necessary to transmit some *oRis* code through the network.

Finally, some applications may simply listen to events on the network, thus only moving agent's ghosts whose "real agents" participate to the application.

Our study is part of a large amount of research works in virtual reality and distributed simulation toolkit such as VR-DECK [3], DIVE [4], NPSNET/DIS [5, 6], and on 3D animation and interaction as VB-II [7], Virtual Studio [8].

These last years, concurrently to those studies, a great effort has been made by designers and publishers so as to improve the rendering algorithm and elaborate efficient graphic engines. Various commercial products can be found: World Toolkit (Sense8, 1991), dVISE (Division) or Clovis (MEDIALAB, 1995).

As far as standardization is concerned, we witness the emergence of a description standard for virtual reality applications: V.R.M.L. 2.0 [9], even though the behavioral or distributed aspects are not really taken into account yet. On the communication part, the next version of the IP protocol (IP v6) introduces the notion of flow and includes the multipoint in native, essential in Distributed Virtual Reality.

Therefore, a proposal for a new toolkit architecture for virtual reality has to be widely opened and as independent as possible from the rendering libraries, so as to be able to use the best solutions [10, 11]. It should be able to describe larger and larger environments, made up of agents with more and more complex behaviors.

Firstly, in the following part of this article, we will present our *ARéVi* platform and *oRis* language. Then, we will describe the distributed aspects of our platform in details.

2 ARéVi and oRis

2.1 ARéVi Platform

ARéVi is a Distributed Virtual Reality platform. *ARéVi* lies in the Entity, Scene and Viewer concepts (see figure 1).

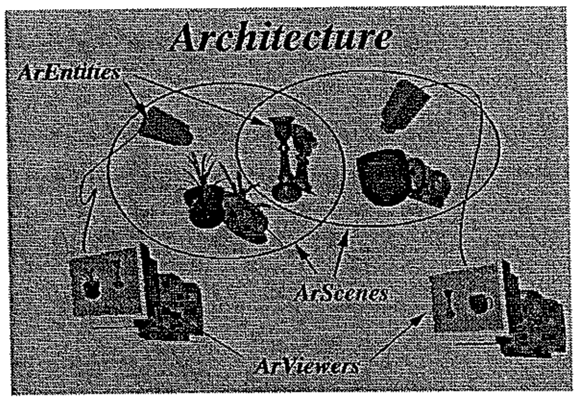

Figure 1: Entity, Scene and Viewer concepts.

The entities are identified agents in space, with a 3D representation. This representation includes animations (succession of consecutives representations) as well as several levels of details (reduction of the facets' number according to the distance from the camera). Then, the entities may be specialized (in an object programming sense) as cameras, lights or every other kind needed by the user [12].

In an *ARéVi* session, each entity is an agent. Therefore, we have built our *ARéVi* platform around our dynamic multiagent language, *oRis*.

2.2 oRis Language

Multiagent formalism as well as the dynamic characteristics and generics of a language are specifically well adapted in a virtual reality system context.

Let's consider two kinds of existing languages: multiagents and "general" object oriented. Most multiagent systems, as described by Starlogo [13], are designed for a specific application and cannot be used in another context. On the opposite, general languages, for example Java, are by definition not specialized, they have some dynamic properties but not all those we require: in particular, the instance granularity is not present. In our *ARéVi* platform we want to be able to re-define a method not only on the class level, but also on the instance level at the same time, which means that the instance may have a very different behavior from those defined in its original class.

To totally fulfill our needs, we have developed our own language: the *oRis* language which is a dynamic multiagent language [1, 2].

2.2.1 oRis concepts

Firstly, *oRis* is an object language: use of classes with attribute and methods. The syntax is close to C++. It embeds most of the language possibilities including multiple inheritance. It is also an agent language: every object with a `main()` method becomes an agent, or active object. This method is cyclically executed by the system scheduler and thus contains the entity behavior.

It has to be noticed that, during a simulation cycle the activation order of the different `main()` methods is randomly accessed.

2.2.2 oRis's dynamic properties

oRis is also a dynamic language. It is able when executed to accept new code, define new classes and re-define methods on the class level as well as on the instance level. Then, it is possible to add on the instance level the definition of a new method which does not exist in the original class. All those possibilities make this dynamic language particularly well adapted to the creation of co-operative applications. In fact, during a session, this language allows the addition of new entities (planned or not when starting the session), the modification of behavior on an entity or an entire entity family. Those modifications may either be introduced by the user (through an HCI or a peripheral) or by the network.

2.2.3 oRis agents' naming

The *oRis* language proposes the type "name of instance" which makes it possible for the differentiation of instances. This type is very similar to the notion of pointer to instances in C++. Actually, a value of the type "name of instance" is not an instance but something allowing to refer to it.

A value of the type "name of instance" is written as a class name followed by a dot and an integer representing an instance number which allows to make distinction between instances of the same class. For a given class, this integer is automatically incremented by the system when creating each instance with the `new` instruction. If you need to define yourself the instance name, you can use the *oRis* `create()` function. The creation doesn't occur in the case of an instance already named.

2.2.4 oRis code examples

The following *oRis* code states the definition of a class A as well as the instantiation of three agents from this class:

```
 // class A
class A{
  void main(void);
}
void A::main(void){
  println("I am ",this);
}
 // instances creation
execute{
   new A;       // --> A.1
   create(A.4); // --> A.4
   new A;       // --> A.5
}
```

The result of two cyclical executions of the agent's method `main()` are represented beneath:

```
I am A.4
I am A.1
I am A.5
I am A.1
I am A.5
I am A.4
...
```

During this execution, the following *oRis* code can also be introduced if defined by a user or received from the network. In this code we define a new class B, which inherits from class A, we instantiate an agent from this class and we also redefine dynamically the method `main()` of the pre-instantiate agent A.4.

```
 // class B inherits class A
class B : A {
  void setB(int value);
  int b;
}
void B::setB(int value){
   b = value;
}
 // instance creation from class B
execute{
   new B;  // --> B.1
}
 // A.4 : main's method re-definition
void A.4::main(void){
  println("I am an agent, my name is A.4");
}
```

Therefore, after the insertion of this code, the results of two cyclical executions of the agent's method main() are represented beneath:

```
I am A.1
I am B.1
I am an agent, my name is A.4
I am A.5
I am A.1
I am A.5
I am B.1
I am an agent, my name is A.4
...
```

2.2.5 oRis/C++ coupling

oRis allows a deep coupling with C++ language. The programmer can specify a connection between an *oRis* class and a C++ class. The call of native reported *oRis* methods initiates the associated C++ methods.

Thanks to the *oRis*/C++ coupling, *ARéVi* offers the user predefined *oRis* classes (native methods) which correspond to 3D graphical classes libraries like OpenGL, OpenInventor on Silicon Graphics.

Moreover, thanks to the *oRis*/C++ coupling, *oRis* may be connected to other languages. For example, we are able to call Java for network communication.

3 Distribution in ARéVi

In its basis version, *oRis* authorizes different types of communications between agents located on the same machine: synchronous (direct method's call), peer-to-peer (message box) and diffusion (events emission towards agents).

Thus, we have developed a communication layer allowing communication among distant agents located on different machines.

Besides, on each collaborating site in a distributed *ARéVi* session, we have decided to make "real' agents (full behavior agents) evolve between proxies ("ghosts" of "real" agents having a simplified behavior) located on the other sites. The "real" agent management is made in *oRis* in order to take advantage of the possibilities given by the communication layer between agents on other sites. Agent migration is also made by simply transmitting some *oRis* code on the network.

3.1 Distribution of agents communication

In *ARéVi*, low level communications are performed thanks to Java programming language. The choice of this language may be penalizing when talking about performances (-20%), but it offers some very interesting development facilities in terms of simplicity (multi-threading), safety (pre-existing group of safe communication classes), HCI (JavaBeans), integration of multimedia processing (Java Media Players), high level network extension (servlets, RMI, aglets), interoperability (CORBA's IIOP) and database connectivity (JDBC) (see figure 2) .

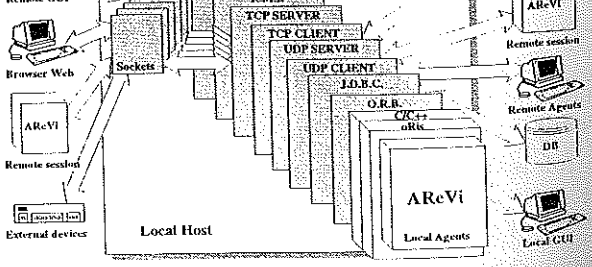

Figure 2: Low level communications in *ARéVi*.

First of all, to launch an *ARéVi* session, it's necessary to start on the Java Virtual Machine. Then, the *ARéVi* session has to be launch in the main Java thread. This architecture allows to very easily add new functionalities by setting them in new threads.

Moreover, because of *oRis*/C++ coupling, it is possible to write, in *oRis*, C++ native methods. With J.N.I (Java Native Invocation) we have another closing connection between Java and C++, and thus we are able to reach Java objects from *oRis* agents, through a pointer on the Java Virtual Machine. The whole communication is written in Java, the programmer only uses the *oRis* layer, where everything is agent. The C++ is acting as an interface from *oRis* to Java.

We have developed different communication classes (Unicast, Broadcast, Multicast), with each time, the notion of clients and servers. Each client and each server is a particular *oRis* agent. Therefore, by inheritance, any kind of agent can transmit or receive a message. Each message passing through the network is identified thanks to the transmitter agent's name and IP address.

Those different functionalities allowed us to extend the communication basis classes between *oRis* agents. Thus, *oRis* agents on different machines may use both kinds of communications, asynchronous (peer-to-peer) or diffusion (emission of events towards the agents).

Today, we are working on the synchronous communication between distant agents (direct method's call). For this, we are thinking about the use of R.M.I Java package (Remote Method Invocation) or a compliant CORBA 2.0 broker allowing us to use the Common Objects Services Specification (COSS) they are defining for distant method invocation.

3.2 "Real" and "ghosts" agents management

With the approach (agent/ghost) we use for the distribution of *ARéVi* sessions, coordination between "real" agents and his ghosts is made on a very simple way by writing *oRis* classes. Therefore, to distribute an agent, it only has to inherit the co-ordination classes.

3.2.1 "Real" and "ghosts" creation

When creating a shared real agent, an *ARéVi* application diffuses this creation to other applications involved in the co-operative session. It asks them to create a ghost (sending some *oRis* code). The created ghost will evolve thanks to its simplified behavioral model.

The *oRis* code sent for the creation of ghosts has to take into account the naming of *oRis* agents described in section 2.2.3.

In fact, in the case of simultaneous instantiation of two agents from class A on different hosts (IP 1 and IP 2), those agents can have the same name. Therefore, it is necessary to distinguish the ghosts from the agents. The solution is to create a ghost for each real agent.

This can be done by simple inheritance and diffusion through the network so that the names of created ghosts enclose the IP address of the host on which the agent has been created (IP1A.1 on host IP 2 or IP2A.1 on host IP 1).

Figure 3 presents the ghosts' creation on hosts IP 2 and IP 3 of the real agent A.1 located on host IP 1.

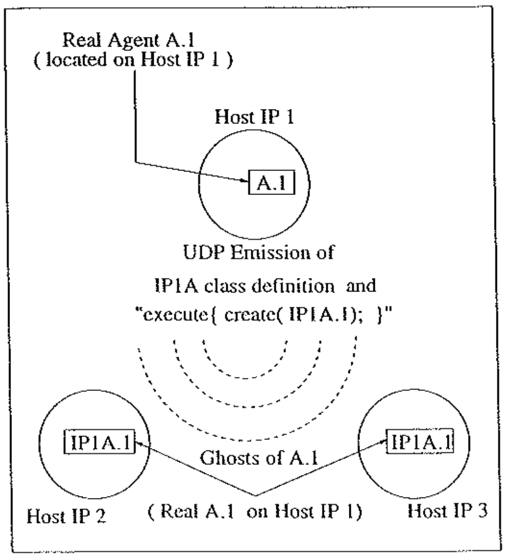

Figure 3: Creation of "real" and "ghost" agents.

This creation implies the emission of *oRis* code on the network. This code described hereafter contains, the new class (IP1A) statement deriving from class Ghost, the simplified model of real agent, and the order of creation of the ghost IP1A.1.

```
class IP1A : Ghost {
 // ghost's naming of an agent from
 // class A located on host IP 1
}
 // instance creation of ghost IP1A.1
execute{
   create(IP1A.1) // --> IP1A.1
}
```

To make it simple and fast, the code emission is made in multicast. But using a datagram transmission can induce a loss of information on the network. The fact that an order of ghost's creation does not reach its destination may occur. We have set up a simple and reliable mechanism to solve the problem. Actually, when an application is ordered to update an unknown ghost, it asks the broadcasting application to send further information in peer-to-peer TCP (reliable mechanism).

3.2.2 "Real" and "ghost" behavior

"Real" agents have a developed behavior described in *oRis*. Ghosts have a simplified behavioral model which requires few calculation. Today, this model is a kinematical model (coordinates, position and orientation, speed and acceleration).

"Real" agents are aware of the model used by ghosts, which allows to practice some "dead-reckoning": they only diffuse their state to their ghosts if this one is significantly different from the state predicted by ghosts. This action aims to reduce the traffic on the network.

When the difference is important between the agent's real state and the state of the ghost predictive model, the agent transmits towards its ghosts an updating order (still by sending some *oRis* code).

Figure 4 presents the updating of ghosts from the session holding the real agent.

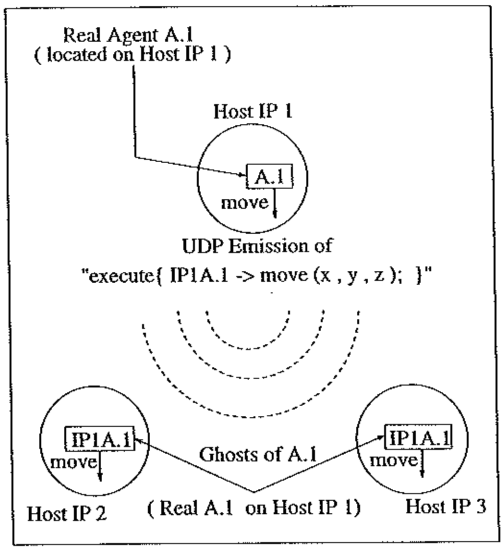

Figure 4: Updating "ghosts" from "real" agent.

The *oRis* code sent to update a ghost's position is very simple:

```
// updating order of a ghost's position
execute{
   IP1A.1 -> move (x,y,z);
}
```

The method called move() corresponds to a method common to every agent in an *ARéVi* session. It allows the modification of the agents' position in 3D space.

3.2.3 "Real" and "ghost" interactions

During the interaction of the user or an agent with another agent, things will happen in a different way whether this latest is a real agent or a ghost:

- the real agent will treat the interaction,
- the ghost will send a request (in *oRis*) to its leading agent or, if it is able to do it, treat by himself the interaction.

In the ghost request mode to the leading agent, the communication uses the TCP mode. Therefore, the agent is able to send to its "master" an order to move. Then the "master" will transmit this order to the ghosts located on other hosts (see figure 5).

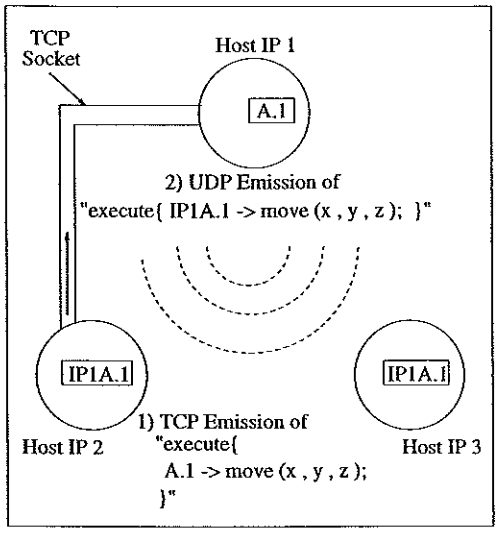
TCP Socket
Host IP 1
A.1
2) UDP Emission of "execute{ IP1A.1 -> move (x , y , z); }"
IP1A.1
IP1A.1
Host IP 2
1) TCP Emission of "execute{ A.1 -> move (x , y , z); }"
Host IP 3

Figure 5: Updating "real" agent from "ghosts".

3.2.4 Agents' migration

In our *ARéVi* platform we need to have the ability to move agents from an *ARéVi* session to another. These migrations can only be obtained if it's possible to move our agent's behavior on the network.

This may be easily done thanks to *oRis* because, to move an agent, it is only necessary to transmit, in TCP mode, some *oRis* code through the network described in a simple way as follows:

```
 // class A
class A{
  // methods runnable by an A agent
}
 // instance creation order
execute{
   new A; // --> A.#
 }
```

For ghosts' renaming reasons we need to transmit in UDP mode the following *oRis* code:

```
 // class A
class IP2A : Ghost{
 // ghost's naming of an agent from
 // class A located on host IP 2
}
// ghost's renaming order
execute{
    IP1A.1->rename(IP2A.#);
 }
```

The migration of agent A.1 from host IP 1 toward Host IP 2 is represented in figure 6.

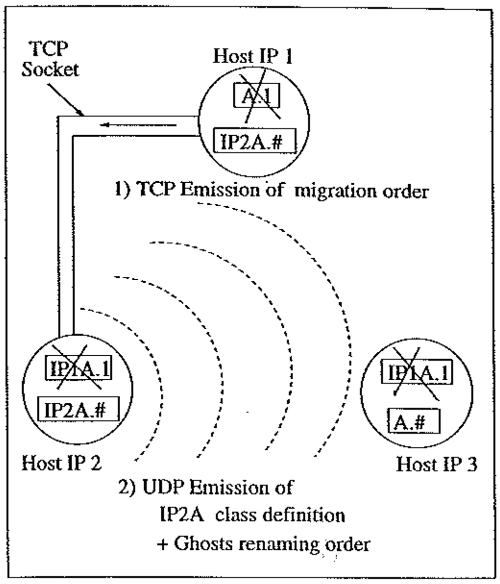
TCP Socket
Host IP 1
A.1
IP2A.#
1) TCP Emission of migration order
IP1A.1
IP2A.#
Host IP 2
IP1A.1
A.#
Host IP 3
2) UDP Emission of IP2A class definition + Ghosts renaming order

Figure 6: Agent's migration.

4 Conclusion

The *ARéVi* platform, built around the *oRis* language, has been made for developing Distributed Virtual Reality applications. The existence of a dynamic multiagent language into the system allows us to create modular 3D universes; modularity is made possible by the lack of general controler and thus by the use of elementary "bricks" with their own goals (agents).

Thanks to *oRis*, we are also able to modify these 3D universes when a session is in progress; this may be done by introducing some *oRis* code by a user action (through an HCI or a peripheral) or by the network.

This essential characteristic of the *oRis* language allowed us to develop a group of communication classes (asynchronous, broadcast) and to move agents on the network. The development of a Distributed Virtual Reality application according to the "real" and "ghost" agent model adopted under *ARéVi* may be done by simple inheritance of existing classes. The classes we have developed allow proper "ghosts" naming on the network and suitable interactions and updates between "real" entities and their "ghosts".

5 Future works

Today, we work on the synchronous communication (call of methods) between distant agents. We are also studying the great possibilities given by the *oRis* language which makes feasible cooperative work prototyping according to specific configurations. For example, the distribution of an agent's methods on different sites to manage load balancing between workstations and the number of connected users on the same distributed *ARéVi* session.

References

[1] F. Harrouet, R. Cozien, P. Reignier, and J. Tisseau. oRis, un langage pour simulations multi-agents. In *Journées Francophones de l'Intelligence Artificielle Distribuée et des Systèmes Multi-Agents*, La Colle-sur-Loup, April 1997.

[2] Harrouet F. Virtual reality for interactive prototyping. Technical report, ENIB/LI2, 1998.

[3] Codella C.F, Jalili R., Koved L. , and Lewis J.B. A toolkit for developing multiuser, distributed virtual environnements. In *IEEE Virtual Reality Annual International Symposium*, 1993.

[4] Carlsson C. and Hagsang O. Dive - a platform for multi-user virtual environnement. *Computer and Graphics*, pages 663-669, 1993.

[5] Brutzman D.P. *A virtual world for an autonomous underwater vehicle.* PhD thesis, Naval Postgraduate School, Monterey, California, 1994.

[6] Macedonia M.R. *A Network software Architecture For Large Scale Virtual Environnements.* PhD thesis, Naval Postgraduate School, Monterey, California, 1995.

[7] Gobbetti E. *Virtuality Builder II, vers une architecture pour l'interaction avec des mondes synthétiques.* PhD thesis, EPFL DI-LIG, 1993.

[8] Balaguer J.F. *Virtual Studio : Un système d'animation en environnement virtuel.* PhD thesis, EPFL DI-LIG, 1993.

[9] Hartman J. and Wernecke J. *The VRML 2.0 Handbook : Building Moving Worlds on the Web.* Addison-Wesley Publishing Company, 1996.

[10] Roohlf J. and Helman J. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *ACM SIGGRAPH*, pages 381-393, 1994.

[11] Strauss P.S. and Carey R. An object-oriented 3d graphics toolkit. In *ACM SIGGRAPH*, pages 341-347, 1992.

[12] Reignier P., Harrouet F., Morvan S., Tisseau J., and Duval T. ARéVi 4.0: A virtual reality multiagent platform. In *Virtual Worlds, July 1-3 1998, Paris (France)*, 1998.

[13] M. Resnick. Starlogo : An environment for decentralized modeling and decentralized thinking. In *CHI*, pages 11-12, April 96.