In the Proceedings of the 6th Eurographics Workshop on Virtual Environments (EGVE'2000), J.D. Mulder and R. van Liere (eds.), Computer Science / Eurographics Series, Springer-Verlag WienNewYork, pages 105-113, Amsterdam, The Netherlands, June 2000.

An Asynchronous Architecture to Manage Communication, Display, and User Interaction in Distributed Virtual Environments

Y. Fabre, G. Pitel, L. Soubrevilla, E. Marchand, T. Géraud, and A. Demaille

EPITA Research and Development Laboratory
14-16, rue Voltaire, 94276 Le Kremlin-Bicêtre cedex, France
thierry.geraud@epita.fr

Abstract. In Distributed Virtual Environments, each machine equally handles the communications over the network, provides the user with a view of the world, and processes her requests. A major issue is to ensure that the network communication does not hinder the interactivity between the machine and the user. In this paper, we present a program designed to achieve this goal, based on tools rarely used in this area.

1 Introduction

Our project, URBI ET ORBI, is a Distributed Virtual Environment, DVE for short: a virtual world in which users may wander and interact with in real time, which is distributed on a set of computers connected via a network. As in [Das97], we focus on virtual worlds for a large audience therefore, there are no other material requirements than a standard personal computer, with a connection to a local network or the Internet. Windows and UNIX or LINUX platforms are supported.

Our first prototype was developed in *Java*; it handled VRML objects and the distribution layer was managed by an *Object Request Broker* as in [Diehl98,Derig99]. Unfortunately, this prototype proved to be disappointing: firstly, the combination of *Java*, VRML, and an ORB turned out to be a major performance penalty, and secondly, many difficulties arose when we wanted to process communication, display and user interaction simultaneously. This is why we have attempted to use non-traditional approaches.

We aim at worlds which are rich intensively (complex scenes), extensively (size of the world), and semantically (a scene description is not limited to 3D data). This excludes VRML, and NPSOFF [Zyda93].

Users may participate in the construction of the world by publishing their own objects / scenes (denoted as *kingdom*), as they can do on the world wide web on HTML pages. But in our context, the task is more complex since we intend to produce an illusion of spatial continuum. We don't not want “hyperspace links” from one kingdom to another.

In order to provide the users with the possibility of publishing their kingdoms, *and* to support large scale environments, we exclude the client/server paradigm, and aim at symmetric (peer-to-peer) distribution. In this framework, no host is dubbed *server*, each machine runs the same software, and participates equally in the distribution, as in DIVE [Fréco98] and MASSIVE [Green95]. This software is therefore in charge of publishing and fetching pieces of the world, while simultaneously displaying and receiving information from the user.

A world is, by essence, dynamic: the avatars move, the scene itself may include animated elements, newcomers may add new descriptions and owners may change their kingdom etc. This dynamic behavior requires a heavy load of data exchange on the network to keep information up to date. One of the main issues in implementing a DVE is the design of a scalable architecture [Singh99].

To scale up properly, we rely on group communication based on multicast. Applications may connect to or disconnect from a group dynamically. Several techniques are used to minimize both the volume of data exchange and the number of participants in these exchanges. Nevertheless, each host is still involved in a large volume of data exchange, which must not interfere with the display or with the interaction with the user.

Since DVE users are humans, the requirements concerning interactivity are strong. In particular, no matter what the state of the network, the system should always be responsive to the user and never freeze.

Because of this requirement, our application is based on several different modules run in different threads which communicate asynchronously. The communications between computers are also asynchronous. The architecture is based on a event-driven model and the pressure upon the network is kept as low as possible.

In this paper, we focus on the software architecture. We explain in section 2 the variety of DVE we aim for. We present our requirements and introduce the language used to implement worlds, *Goal*. Section 3 presents the architecture of our application, its components and its functioning. An example of the execution of a *Goal* instruction is presented in section 4. We conclude in section 5.

2 General description

Before presenting the architecture of the application, we shall give information about our DVE design.

As demonstrated in the introduction, communications are the key issue [Brutz97]. While most projects have centered their approach upon the 3D data, we have oriented ours towards the structure of the information, 3D data being like any other kind of data, and towards the communication.

2.1 Replication or Distribution?

A specific feature of URBI ET ORBI, with regard to most other frameworks, is that each host which participates in the world is both a “server’ and a “client”

[Lea97,Das97,Sugan97]. Therefore, all the computers run the same application which combines both server (propagating the information) and client tasks (rendering and interaction). Urbi et Orbi can be seen as a different web: people publish and visit kingdoms.

As is the case for the world wide web, no single host knows the current state of the whole world, or even merely an outdated approximation: the knowledge is distributed across the machines, which only know the kingdom of their user, and the part of the world where their user are. This is so called "shared distributed data bases with peer-to-peer updates" according to [Maced97].

Now, consider a scene with a windmill whose arms are rotating. One can imagine at least two ways of publishing this information:

distribution the "owner" of the windmill sends frequent updates of the position of the arms to its observers.

replication several windmills were actually "created", one per host. The implementation of the windmill is run on each machine.

The *distributed* approach is very adapted for avatars: because its behavior is unpredictable and there can be only one command center: the person who commands the avatar. Conversely, imagine a vulture flying in circles over a herd of sheep: its movement is fully predictable, and frequent updates of its position would waste the bandwidth. Here, the *replication* is a natural solution.

Because both approaches are needed to develop a distributed world, the language must help the programmer and support both types of publication.

Because the vulture may suddenly change its attitude, we must also be able to switch the mode of publication dynamically. Such a task is inherently complex, but the complexity should be hidden from the programmer, and handled by the system itself.

2.2 Group communication

We have chosen to use group communication as our base communication layer. We excluded Unicast because each time a machine needs to publish an update, it would have to send one message per connected host: destinations accumulate. We excluded Broadcast, since each time a machine publishes an update, all the machines would have to process the message, and maybe discard it, what results in useless additional work: sources accumulate.

Group communication has several benefits [Tramb99]. Firstly, it allows the deployment of several groups with the same members but with different protocols, hence various qualities of service. The possibility to choose the protocol is a means to control the load imposed on the network: urgent messages which must be delivered safely obtain most of the bandwith, while messages of little importance may be delayed or even lost.

The notion of group also helps in the design of the virtual world, since they constrain the programmer to structure its description and to partition it properly.

Because a single “player” may belong to several different groups, the “navigator” of the player has to deal with several communication channels. This results in quite delicate network programming. Therefore, group communication should be a primitive of the environment provided to the programmer. In our project, the programmer is provided with a high level language, *Goal*, which offers a group communication layer (see section 2.4).

The distribution layer we use is based on *Ensemble*, a parameterizable stack of group communication protocols [Hayde98] written in the functional language *Objective Caml* [Rémy98,Leroy99].

2.3 Code migration

In the example of the vulture, presented in section 2.1, while in the distributed approach it is sufficient that only the creator of the vulture has its code, in the replicated case the “vulture program” needs to be published. Therefore, because we want to be able to distribute *behaviors*, in addition to “inert” data, the project must support code migration.

There are several options available for code migration, most prominently *Java* byte code.

However, because no language offered all the facilities we wanted to implement a DVE, we decided to build our own language, *Goal* (see section 2.4). Since one of the main purposes of *Goal* is to offer an abstraction of the network to the programmer, its runtime naturally resembles a miniature OS kernel, which is named MMK, for Matrix Micro Kernel (see [Schma96] for etymology).

2.4 *Goal*

Goal is the vehicular language used throughout the project. “High” *Goal* is used between the human and the system to describe the world and its evolutions, and “low” *Goal* is exchanged between some of the modules and between the machines.

Goal is a frame language with classes, inheritance, and reflexivity. Each slot, or attribute, of the objects can be equipped with a daemon which is activated when the data is altered. We have found the daemon approach to be extremely natural and to have a very good effect on the modularity of the description of the world.

Goal is a scripting language, interpreted within MMK (see figure 1). Although it is interpreted, thanks to the many optimizations (e.g., early binding as in PostScript) its interpretation is rapid. On the other hand, because the programs are scripts, code can easily migrate, which is necessary to properly model animated objects.

Goal is a language designed to support distribution and replication (services from MMK, see Section 2.1). Because we want also the knowledge of the world to be distributed, the information is implemented in *Goal* as a conceptual graph, of which each host has a partial view.

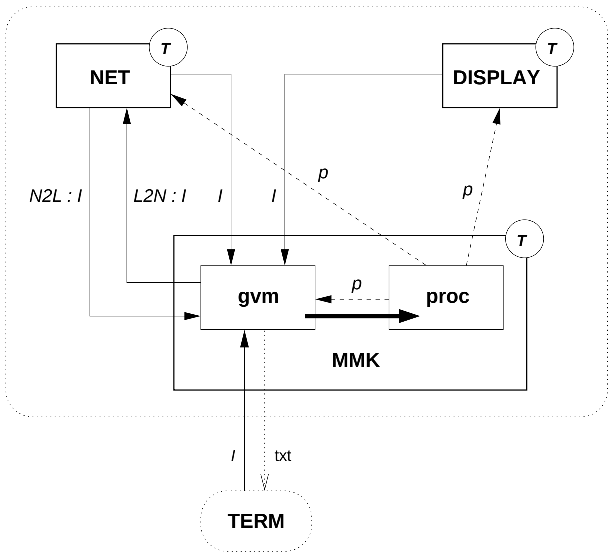

Fig. 1. Software Architecture. Each rectangle denotes a module. Each separate thread is symbolized with *T*. The arrows represent *channels*, i.e., queues of messages/requests, and not function calls.

3 Software architecture

The software is composed of three active modules as depicted in figure 1. Each module is responsible for an independent task and offers particular services.

MMk (or kernel) is the core of our software; it receives a flow of instructions, schedules their execution and manages the resources (files, memory, display and communication).

NET is in charge of the communications through the network; it handles the connection / disconnection and information transfer.

DISPLAY (or renderer, or navigator) provides the user with read access to the state of the world as known by MMk. It also allows the user to "write": when a user moves or activates an element of the world, feedback is given to MMk. For simplicity's sake the data circuit is not represented in figure 1 and it will not be discussed.

In figure 1, the TERMinal is also represented; it is a shell (command interpreter) which allows direct textual communication with MMk, via *Goal* instructions. This proved to be a great help in the design and implementation of virtual worlds.

Several other projects have based their approach upon a kernel, such as Maverik [Hubbo96]. One of the most striking difference between the two projects lies in the fact that their approach is based upon modules, while we stressed the importance of the high level language, *Goal*.

A central module of Maverik is its SMS, Spatial Management System, which is in charge of processing any thing related to 3D. In Urbi et Orbi, there is no such module. Because we aim a generic approach of the information, we have tried to avoid any dedicated low level code for 3D in the kernel: the processing of 3D information is spread across the components (more specifically the renderer).

This results in a wide set of primitive operations that are bound to *Goal* instructions. The programmer then merely needs to attach such *Goal* instructions to her objects to access these primitives. For instance *Goal* objects, typically 3D objects, maybe be bound to a cell, which means the object is "in" the cell. Then, the programmer merely has to declare that her object `implements gridlistener[pos]`, in order to have automatic binding of the object to the proper cell. This makes heavy use of the daemons.

3.1 Multi-Threaded and Asynchronous

Monolithic software is not adapted for interactive worlds: in order to provide the user with proper interactivity, tasks should be scheduled to favor human-visible operations. Fairness is also a strong requirement: neither rendering nor communication can be interrupted without degrading the interactivity of the whole application. Even if the traffic is high, the renderer must not freeze, and conversely, a complex rendering must not prevent the host from communicating.

Therefore, in Urbi et Orbi, each task runs in a separate *thread* (symbolized by the letter *T* in figure 1). The implementation, based on time slots, guarantees that we cannot enter a deadlock. *Goal* is sufficiently expressive so that programmers may introduce bugs in their *Goal* programs, such as deadlocks. However, thanks to the time slots, the system is still fair, and each task will be provided with the ability to make a step.

In addition to the real OS of the host, MMk handles concurrency between the modules, such as the renderer and the communication modules. Each module may send requests to another via MMk; contrary to procedure calls, the modules cannot be stuck while making a request. MMk delegates their execution to the proper modules, which run concurrently. Again, thanks to the time slots, it is also impossible for a module to be stuck.

3.2 Message Passing between Modules

Two types of data are exchanged between the three modules:

- *Goal* instructions (denoted *I* in figure 1),
- procedures to run (denoted *p* in figure 1).

MMk includes the Goal Virtual Machine, *gvm* (see figure 1), to run *Goal* scripts. When a *Goal* instruction requires services (network, display etc.), the interpreter delegates the procedure call to the sub-module proc. proc then sends

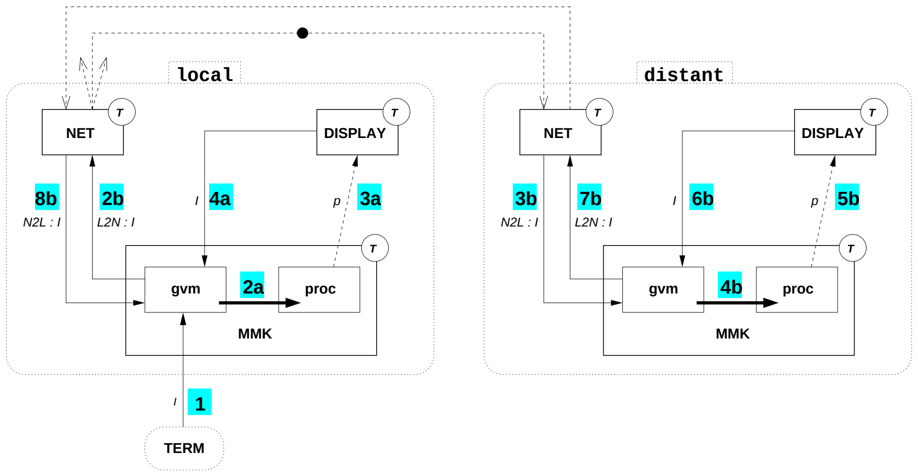

Fig. 2. Moving an `anvil` in Urbi et Orbi

the *procedure* ***p*** to the proper module. Technically, ***p*** is a closure: it contains both the procedure and the environment it needs. Then the module applies the closure: once the whole environment is filled, the procedure is executed. The receiving module is not blocked while waiting further data: it still executes other routines. This mechanism could be considered as an *asynchronous execution transfer*.

Most of the low level service routines are compiled *Objective Caml* code, executed in the proper module (hence a thread). Therefore, a single *Goal* instruction, when received by the *gvm*, can lead to executions in several different threads.

4 Example of execution in MMk

We shall use a simplified example to demonstrate the functioning of our architecture. Below, we enumerate the different phases induced by a simple drag and drop of an `anvil` by the user, hosted on the machine `local`.

In order to simplify figure 2, we do not represent the navigator component in charge of listening to the user. Nevertheless, its reaction to the drag and drop is rigorously the same as if the user had typed the following code on her terminal:

```
. anvil <- set-position([1.0, 1.0, 1.0])
```

The first character, denotes the class of the communication: here `.` stands for broadcast to the members of the group, and `@` for local communication only. In our case, since an object is moving, a broadcast is the appropriate communication mode.

The next token, `anvil`, designates the object which will receive the message. The previous sentence can also be interpreted with a object-oriented reading.

The message consists of the name of the routine (or *method*), together with its arguments, here a simple vector.

When the interpreter processes this message, it first notes the communication class, and forwards the message to its peers. The message is slightly altered, the class is now *local*:

```
@ anvil <- set-position([1.0, 1.0, 1.0])}
```

Then, on `local`, the interpreter processes the request, and delegates to the module `proc` the handling of commands that are bound with compiled code. In our case, the native code associated with `set-position` sends a request for an update to the `renderer`. The vocabulary was chosen on purpose: while the previous actions where executed with traditional procedure calls, at this point the request is *posted* into the `renderer`'s mailbox.

The renderer, a separate thread, then proceeds and updates the display (calls to the C bindings to OpenGL) after having verified that there are no collisions. Since it has ended with success, an acknowledgement is posted back to the interpreter.

Let us concentrate now on the machine `distant` which is one of the members of the group. It has received the message, and starts the processing as did `local`. In our scenario another action occurred during the transmission of the message which now blocks the move: for instance, almost simultaneously, someone moved his `feather` to the same place. This is possible because of network latencies.

The `anvil` then collides with the `feather`. This will be detected by the OpenGL engine, which will resolve the collision itself, i.e., another vector v' will be taken into account. This crucial information will be posted back to the interpreter, which will forward this information only to `local`.

There is no reason to forward this update to all the other members of the group, which will also detect the collision, therefore a broadcast would be useless and costly here. Nonetheless, although `local` will probably receive the updates from all the members of the group, this feedback is crucial in order to keep the states synchronized by `local` and the `distant` machines.

It should be noted that we made the decision to use asynchronous and unsafe communication because it is much cheaper than “better” communication modes. Because of this choice, there is no guarantee of the results.

5 Conclusion

Urbi et Orbi is a dve based on non standard choices. It uses on symmetric distribution provided by its core, MMk, which is very much like a tiny OS kernel. The worlds are implemented in a *language*, *Goal*, as opposed to *formats* such as vrml and our implementation language is functional, *Objective Caml*. Our prototype runs on standard PCs. Our experiments were limited to a fast LAN (Ethernet 100MB), with PC equipped with 3D video cards: we typically reach

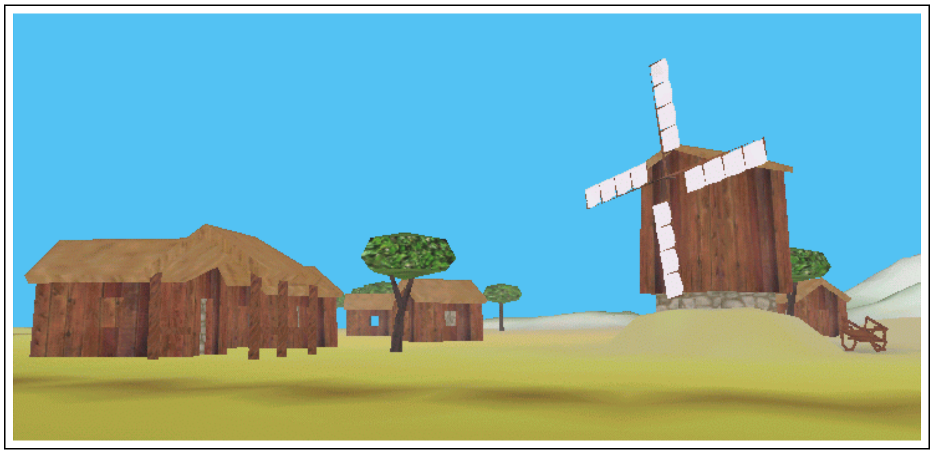

Fig. 3. A scene from Urbi et Orbi

40 fps with high quality images (see figure 3), and excellent interactivity. We consider these figures to be satisfactory, they validate the architecture of Urbi et Orbi, presented in this paper.

References

Brutz97. Don Brutzman. *Graphics Internetworking: Bottlenecks and Breakthroughs*, volume [Dodsw97], chapter 4, pages 61–97. Addison-Wesley, 1997.

Das97. Tapas K. Das, Gurminder Singh, Alex Mitchell, P. Senthil Kumar, and Kevin McGee. Developing social virtual environments using NetEffect. In *Proceedings of the 6th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE'97)*, IEEE Computer Society Press, pages 148–154, 1997.

Derig99. M. Deriggi, M. M. Kubo, A. C. Sementille, S. G. Santos, C. Kirner, and J. R. F. Brega. CORBA platform as support for distributed virtual environments. In *Proceedings of the IEEE Virtual Reality International Conference (VR'99)*, Houston, USA, March 1999.

Diehl98. Stephan Diehl. Towards lean and open multi-user technologies. In *Proceedings of the International Symposium on Internet Technology (ISIT'98)*, Taipei, Taiwan, April 1998.

Dodsw97. Clark Dodsworth, editor. *Digital Illusions*. Addison-Wesley, 1997.

Fréco98. Emmanuel Frécon and Møarten Stenius. DIVE: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments)*, 5(3):91–100, September 1998.

Green95. Chris Greenhalgh and Steve Benford. MASSIVE: a distributed virtual reality system incorporating spatial trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems (DCS'95)*, IEEE Computer Society Press, pages 27–34, Vancouver, Canada, May-June 1995.

Hayde98. Mark Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, January 1998.

Hubbo96. Roger Hubbold, Xiao Dongbo, and Simon Gibson. Maverik – the Manchester virtual environment interface kernel. In *Third Eurographics Workshop on Virtual Environments*, 1996.

Lea97. Rodger Lea, Yasuaki Honda, and Kouichi Matsuda. Virtual Society: Collaboration in 3d spaces on the internet. *Journal of Collaborative Computer Supported Cooperative Work (CSCW)*, 6(2/3):227–250, 1997.

Leroy99. Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml system.* INRIA, 1999. http://caml.inria.fr/index-eng.html.

Maced97. Michael R. Macedonia and Michael J. Zyda. A taxonomy for networked virtual environments. *IEEE MultiMedia*, 4(1):48–56, January-March 1997.

Rémy98. Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Objects Systems*, 4(1):27–50, 1998.

Schma96. D. Schmalstieg and M. Gervautz. Implementing gibsonian virtual environments. In *Proceedings of the 13th European Meeting on Cybernetics and Systems Research*, pages 928–933, Vienna, Austria, April 1996.

Singh99. Sandeep Singhal and Michael Zyda. *Networked Virtual Environments – Desgin and Implementation.* ACM Press, SIGGRAPH Series. Addison–Wesley, 1999.

Sugan97. Hiroyasu Sugano, Koji Otani, Haruayasu Ueda, Shinichi Hiraiwa, Susumu Endo, and Youji Kohda. SpaceFusion: A multi-server architecture for shared virtual environments. In *VRML'97*, 1997.

Tramb99. Henrik Tramberend. Avocado: A distributed virtual reality framework. In *Proceedings of the IEEE Virtual Reality International Conference (VR'99)*, Houston, USA, March 1999.

Zyda93. Michael J. Zyda, Kalin P. Wilson, David R. Pratt, James G. Monahan, and John S. Falby. NPSOFF: An object description language for supporting virutal worlds construction. *Computer and Graphics*, 17(4):457–464, 1993.