A Distributed Kernel for Virtual Time Driven Applications

Philippe Ingels, Carlos Maziero*, Michel Raynal
Team ADP - IRISA[†]
Campus de Beaulieu - 35042 Rennes cedex - France
<*Name*>@irisa.fr

Abstract

The advent of distributed memory parallel machines turns feasible the design and implementation of dedicated environments for distributed applications. This paper addresses such a realization by presenting a distributed kernel, called FLORIA, whose aim is to provide a virtual time environment for application programs (distributed discrete event simulation is the most known of the applications based on such a virtual time - the so called simulation time - and consequently constitutes the paradigm of this class). This paper presents first basic features of the virtual time concept and then describes the time-related aspects of the implementation of a distributed kernel dedicated to virtual time driven applications.

1 Introduction

Uses of computers can be roughly divided into two categories according to the way they take the time into account, namely the applications whose semantics is based on some notion of time and those that do not. Real-time applications are the most famous representative of the first category; numeric applications are an example of the second one. Specificities of each category have to be taken into account by the dedicated underlying operating system; actually, in the first category the physical time is a programming object that drives the application and that can be used by it, whereas in the second category the time appears only as an implementation support.

Thus for real-time applications, physical time, defined by the environment, constitutes the point of reference common to all the processes that allow them to synchronize the ones with the others and with the environment in order to carry out the desired control. This way of using time, applied to its logical aspects, has given rise to the *virtual time* concept [3]. Such a context is so characterized by a global logical clock, common to all the processes; its progress is defined by application relevant rules (and not by external physical phenomenon as in real-time applications). This clock allows to manage the computation, to control its progress, to timestamp events, to measure virtual time durations, etc.

The FLORIA project is devoted to the definition, and to the associated implementation on a distributed memory parallel machine, of a distributed kernel for virtual time driven applications. This paper is composed of two main parts; the first one (§2) states the properties offered to user programs by the virtual time concept; the second one (§3) presents the corresponding distributed system kernel we have implemented on an Intel iPSC/2 hypercube.

*This author is supported by a fellowship from the CAPES Agency of the Government of Brazil.

†This work has been partly funded by the french national project C^3 on concurrency and distribution.

2 The virtual time provided by the kernel

2.1 Virtual time properties

A virtual time driven application is structured as a network of processes communicating though FIFO channels. Each process is endowed with input channels from which it can receive messages, and output channels on which it can send messages.

The two main features of the virtual time offered by the kernel can be expressed as temporal properties that can be used by processes of an application program:

P1: All the application processes have the same perception of the virtual time (logical synchronism of the processes), that is to say, the virtual time is the same and progresses logically at the same speed for all of them.

P2: Within the virtual time frame, any message sent by a process at the date t is received by its destination process at the same date t.

The processes interact only by messages. Consequently, virtual time constitutes the only mechanism to synchronize processes.

0-8186-2812-X/92 $03.00 © 1992 IEEE

2.2 The kernel interface

The kernel can be seen by a process P_i as a machine working in virtual time; in particular it provides a global virtual clock *gvc* which gives the current virtual date; for this, the kernel supplies each process P_i on the one hand with a local representation of the global virtual clock *gvc*, and on the other hand with all messages sent to it till the current virtual date (which is by definition the value of *gvc*).

So as all the messages sent to a process P_i at a time less than *gvc* are known by P_i, it can consumes them or not (the effective message consuming depends on its own behaviour): these messages are put in a queue called $known_i$, ordered by increasing sending dates. The kernel offers some primitives to access this queue (**Empty**, **First**, **Last**, **Next** (*msg*), **Previous** (*msg*)) and to take messages from it (**Take** (*msg*)). These operations have a zero duration in virtual time. So at virtual time t the queue $known_i$ contains all the messages sent to P_i before or at the date t, not yet consumed by P_i with the help of the primitive **Take** (*msg*).

In addition to the message sending primitive, the kernel provides the processes with the following primitives, which have non-zero durations with respect to virtual time:

- **Wait** ($gvc = t$) : the invoking process progresses without interruption till the value of the global virtual clock is t.
- **Wait msg** : the process is blocked till a new message is received (then the global virtual clock value is equal to the message sending date - property **P2** - and the message is put in $known_i$; it can be consumed by **Take** (*msg*)).
- **Wait** ($gvc = t$) **or msg** : the progress of the invoking process is blocked till the global virtual clock value is t, unless a message m is received by the date t (in other words, that is a waiting for a message with a deadline t).

All the other operations a process can execute have zero duration with respect to the virtual time.

Designing and implementing a kernel consists in building a basic software layer providing a set of primitives general enough to solve a class of problems. These kernel primitives are not to be used directly by the user program. For example, if one is interested in designing queueing network simulation programs, he/she will use a well-suited simulation language, and a compiler will translate the statements in which virtual time occurs into calls to the kernel primitives. More generally, operations offered to the user are implemented with calls to the previous basic primitives. So for example waiting, by a process P_i, for a message according to the FIFO discipline would be realized with calls to **Empty**, **Wait msg**, **Take** (*msg*) and **First**.

3 Kernel implementation

3.1 Objectives

Building a virtual time kernel on an asynchronous distributed memory parallel machine demands to solve the two following important problems: the assignment of the processes to the processors and the consistent implementation of a virtual-time run-time. In the current version we concentrated our efforts only on the run-time needed by the virtual time; this one has to be consistent, i.e. ensure properties **P1** and **P2**, and it has to be efficient in order to have the best possible computing time (measured in real time).

The kernel has been implemented using the language C, on a 64 processors Intel hypercube iPSC/2. It is made of two parts: the communication layer that ensures the property **P2** and, in conjunction with this communication layer, process managers that drive the progress of each process as independently as possible.

3.2 The Communication layer

Here the property to ensure is **P2**, namely every message sent at virtual time t has to be delivered to its destination process at the same virtual time t. In order to implement it first each message piggybacks a timestamp (its sending virtual time) and second the communication layer delivers to each process the messages received according to the increasing timestamp order. This rule ensures the consistent delivery of messages to a process according to the virtual time progress (in other words, a process cannot backtracks in virtual time by receiving first a message sent at virtual date t and after another message sent at date t' with $t' < t$) [2, 3, 5].

This simple rule ensures safety of deliveries but is not sufficient to ensure their liveness (i.e. each message that can be delivered will be delivered); deadlock situations are actually possible [2, 3, 5]. Let us consider the figure 1, where process P_i, endowed with two input channels, has received three messages m_1, m_2 et m_3 with respective timestamps such that $t_1 < t_2 < t_3$.

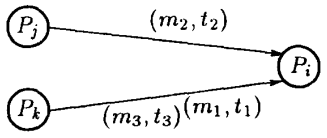

Figure 1: Deadlock in message delivering

The communication layer can safely deliver m_1 at virtual time t_1, then m_2 at time t_2, but cannot deliver m_3 at time t_3 : because process P_j can send to P_i a message m'_2 at virtual time t'_2, with $t_2 < t'_2 < t_3$, and in that case m'_2 has to be delivered to P_i before m_3; in other words, while a message with a timestamp greater than t_3 has not been received from P_j, the kernel cannot safely delivers m_3 to P_i. Consequently a deadlock is possible. To solve this problem, two class of techniques have been proposed: the optimistic ones [3] and the conservative ones [1].

The FLORIA kernel implements a conservative approach to prevent deadlock [5] : additional control messages, called $NULL$ messages, are used. When a process P_j knows it will not send an application message on some channel by virtual time t, the kernel sends the control message $(NULL, t)$; this anticipation on the future of output channels allows a look-ahead for receiving processes that allow them to prevent deadlocks; the higher the value of t is, the more efficient the implementation is [2]. The kernel offers to a process a primitive to define its look-ahead value.
Let the following variables, the process P_i being endowed with k_i input channels : $in_i[1], \ldots, in_i[k_i]$:

- $channel\_time(in_i[x])$: the timestamp of the last (application or $NULL$) message received by P_i on its input channel $in_i[x]$.
- $ec_i = min_{1 \leq x \leq k_i}(channel\_time(in_i[x]))$; that is actually a local clock associated to the set of input channels of P_i; this entry clock is managed by the communication layer.

Then the communication layer can safely delivers to P_i (in their timestamp order) all the messages (m, t) received that have a timestamp t less than or equal to ec_i; in other words this value represents the date upperbound till which the content of input channels of a process P_i are safely known and consequently can be delivered to it (put in $known_i$) according to the progress of the virtual time.

3.3 Process managers

Each application process P_i is endowed with a local representation lc_i of the global virtual clock gvc; in other words, $lc_i = t$ means that P_i has progressed till the virtual date t. In fact it is possible to have, at a given execution time, $lc_i \neq lc_j$ without violating property **P1**; to ensure this property it is sufficient that every process P_i at virtual date t ($lc_i = t$) does not perceive another application process at a different virtual date.
If for P_i the current virtual time is t ($lc_i = t$), $ec_i \geq t$ is always verified, thanks to the virtual time communication layer (§3.2) which put in $known_i$ all the messages (m, t') such as $t' \leq lc_i = t$. We remind that P_i is free to take or not the messages from $known_i$, that depends on its own program text.
In fact these two virtual clocks lc_i (process clock) and ec_i (entry clock) of a process P_i need not be tightly synchronized; a loose synchronization improves execution efficiency, the kernel do only the synchronization needed to ensure properties **P1** and **P2** in virtual time. The following describes this loose synchronization for two primitives whose semantics is based on virtual time (the third one is only a combination of those two ones).

3.3.1 Implementation of a waiting for a deadline

When a process executes **Wait** ($gvc = t$), it continues with $lc_i = t$. We remind that by definition the $known_i$ queue contains, at time t, all the messages sent to P_i before t and not yet consumed by the **Take** (msg) primitive. With this implementation, which consists simply to do the assignment $lc_i := t$, this property can be violated. But the analysis of the two following situations shows that it can always be restored in such a way that this temporary inconsistency can never be perceived by the process P_i.

- $lc_i < ec_i$: all the messages (m, t') sent in the past of P_i (i.e. such that $t' \leq lc_i$) are available in the queue $known_i$ queue (figure 3.3.1); P_i can access and take them, according to its program text.

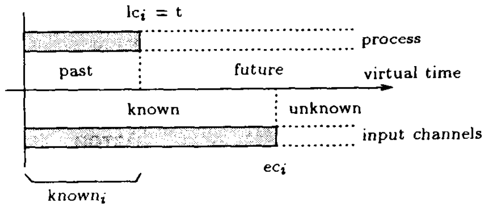

Figure 2: situation $lc_i < ec_i$

- $lc_i \geq ec_i$: in that case, a part of the messages sent in the past of P_i (the messages (m, t') such as $ec_i < t' \leq lc_i$) is not really available, hence the queue $known_i$ is not completely known (white part in figure 3.3.1).

  Only the messages (m, t') such as $t' \leq ec_i$ are really available in $known_i$. In this situation we could block P_i's execution till $lc_i < ec_i$, but that is not always necessary. (for example, when P_i don't use messages sent between ec_i and lc_i, as it is the case with a simulation program of a FIFO server that has still to serve client requests arrived before ec_i). We have chosen to allow P_i to continue its execution in this situation; but, in order to be consistent, if P_i calls an access primitive to $known_i$ (**First**, **Last**, etc), the call will be blocked if it tries to access the unknown part of $known_i$ (the white part in figure 3.3.1) and the execution of P_i will continue when the virtual time communication layer delivers to it the needed messages; so the property associated to the definition of $known_i$ will always be verified from P_i's point of view.

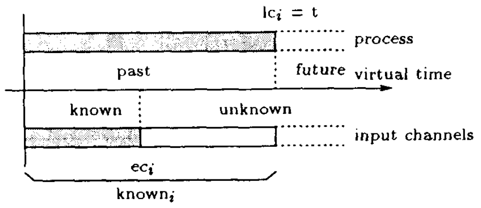

Figure 3: situation $lc_i \geq ec_i$

3.3.2 Implementation of an unconditional waiting for a message

The execution by P_i of a **Wait msg** primitive always provokes the progress of P_i's virtual clock. P_i can be blocked or not, according to the existence of a message in its input channels:

- $\exists(m,t) \mid lc_i < t \leq ec_i$: in this case the older message (m,t) is put into $known_i$ and lc_i is moved forward to t (P_i can then consume it with the kernel primitive **Take** (msg), §2.2).
- $\nexists(m,t) \mid lc_i < t \leq ec_i$: the process is blocked and its execution will continue only when a new message is delivered to it by the virtual time communication layer; then the same treatment as in the previous case can be applied.

4 Conclusion

A first version of the FLORIA kernel is now implemented; the debugging and assessment process is on. Various queueing networks simulation programs have been executed on top of this kernel. These experiments show that the primitives offered by the FLORIA kernel provides an efficient and simple run-time for this kind of virtual-time based applications.

At a more general level, the design of a distributed kernel suited to virtual time driven applications is an interesting research area, thanks to the synchronization constraints defined on communications (property **P2** : a message send at time t is received at the same time t); we call this property VTO (virtual time ordering). The potential causality relation between events, defined by Lamport [4] and noted "$\rightarrow$", allows to model a distributed computation as a partial order on the events produced that shows only causalities between them. Some system kernels offer the causal communication [6] (noted CO, for "causal ordering") as the basic communication scheme; CO is defined in the following way : if $send(m_1)$ and $send(m_2)$ are two sendings of messages to the same destination process P_i and if $send(m_1) \rightarrow send(m_2)$, then m_1 must be delivered to P_i before m_2 (in the case m_1 and m_2 are sent by the same process, the CO property is reduced to the FIFO property). It is easy to see that for communications we have VTO property $\Rightarrow$ CO property $\Rightarrow$ FIFO property. That suggests a thorough study of the properties of basic communication schemes. The FLORIA kernel is a concrete step in this direction.

References

[1] K.M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Trans. on Soft. Eng., Vol. 5, No 5*, 440–452, September 1979.

[2] R. M. Fujimoto. Parallel discrete event simulation. *Communications of ACM*, 33:31–53, October 1990.

[3] D. Jefferson. Virtual time. *ACM Toplas, Vol. 7, No 3*, 404–425, July 1985.

[4] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] J. Misra. Distributed discrete-event simulation. *Computing Surveys, Vol. 18, No 1*, 39–65, March 1986.

[6] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Proc. Letters*, 30:343–350, 1991.