

LIFL 2002-n°04

Mai 2002



Publication

LIFL 2002-n°04

**Towards Distributed Process  
Networks with CORBA**

Abdelkader Amar, Pierre Boulet and Jean-Luc Dekeyser

*15 mai 2002*



# Towards Distributed Process Networks with CORBA

Abdelkader Amar, Pierre Boulet and Jean-Luc Dekeyser  
Laboratoire d'Informatique Fondamentale de Lille  
Université des Sciences et Technologies de Lille  
Cité Scientifique, Bat. M3, 59655 Villeneuve d'Ascq cedex France  
{Abdelkader.Amar,Pierre.Boulet,Jean-Luc.Dekeyser}@lifl.fr

May 15, 2002

## Abstract

Process networks is a widely used model to describe highly concurrent applications. We present here a distributed implementation of a slightly restricted process network model realized using the CORBA middleware. This implementation allows the non computer science specialist to easily program heterogeneous meta-applications based on an assembly of components communicating through FIFO queues.

## 1 Introduction

Many parallel applications, specially in the signal processing domain can be modeled with Kahn process networks [5, 6]. In this model, processes communicate via unbounded first-in first-out (FIFO) queues exclusively. This model has a dataflow flavor and can express a high degree of concurrency which makes it particularly well suited to model intensive signal processing applications or complex scientific applications. This model makes no assumption on the computation load of the different processes and thus is heterogeneous by nature.

Distributed architectures provide an attractive alternative to supercomputers in terms of computation power and cost to execute such complex and computation intensive applications. The two main weak points of these architectures are their communication capabilities (relatively high latency) and often the heterogeneity of their hardware.

We present in this paper a distributed implementation of a subcase of the process network model on heterogeneous distributed hardware. The different processing power of the connected computers is a good support for the different computation needs of the networked processes. We have chosen to use the Common Object Request Broker Architecture (CORBA) [10]

middleware to handle the communications for its interoperability properties. Indeed, each process of the process network can be written in a different language and run on a different hardware, provided that these are supported by the chosen Object Request Broker (ORB).

The concept of dynamicity (migration and replacement of components) seems essential to us because of our application domain, namely scientific computing and more specifically intensive digital signal processing. Indeed, we target long running distributed applications. Actually these applications may run indefinitely, taking an infinite data stream as input. Allowing to improve the code or the hardware while the application is running is an important goal to us.

The rest of this paper is organized as follows. In section 2, we present the model we use and motivate our approach. Section 3 gives a description of the basic building block of our distributed process networks, namely distributed FIFO queues. The components (implementing the distributed processes) are described in section 4 and section 5 explains how to build a dynamic component graph. We then detail an application in section 6 and we outline our conclusions and plans for the future work in section 7.

## 2 Model and Implementation Choices

### 2.1 Process Networks

The process network model has been proposed by Kahn and MacQueen [5, 6] to easily express concurrent applications. Processes communicate only through unidirectional FIFO queues. A process is blocked when it attempts to read from an empty queue. A process can be seen as a mapping from its input streams to its output streams. The number of tokens produced and their values are completely determined by the definition of the network and do not depend on the scheduling of the processes. Thus the process network model is called determinate.

The choice of a scheduling of a process network only determines if the computation terminates and the sizes of the FIFO queues. Some networks do not allow a bounded execution. Parks [11] studies these scheduling problems in depth. He compares three classes of dynamic scheduling: data-driven, demand-driven or a combination of both with respect to two requirements:

1. Complete execution (the application should execute completely, in particular if the program is non-terminating, it should execute forever).
2. Bounded execution (only a bounded number of tokens should accumulate on any of the queues).

These two properties are shown undecidable by Buck [3] on boolean dataflow graph which are a special case of process networks. Thus they are also

undecidable for the general case of process networks. Data-driven schedules respect the first requirement, but not always the second one. Demand-driven schedules may cause artificial deadlocks. A combination of the two is proposed by Parks [11] to allow a complete, unbounded execution of process networks when possible.

Several implementations of process networks are used for different purposes: for heterogeneous modeling with PtolemyII [8], for signal processing application modeling with YAPI [4] and for metacomputing in the domain of Geographical Information Systems with Jade/PAGIS [12, 13]. Only the Jade/PAGIS implementation is distributed. The PtolemyII and YAPI implementations use threads to represent the different processes. Furthermore, none of these implementations allow the coupling of processes written in different languages.

## 2.2 Restrictions on the Process Network Model

One of our goals is to hide the complexity of building distributed applications to the programmer, typically a non computer science specialist. The user should just have to write his domain specific processing functions and their prototypes and a code generator should take these descriptions to produce distributed code, effectively hiding all the details of communication and synchronization, thus achieving a high level of transparency.

To reach this goal, we make some restrictions on the processes in our implementation. These restrictions simplify the scheduling of the process network while retaining the expressing power we need for our applications. In our implementation processes are functional, meaning that they work on the following schema:

1. optional initialization phase where the process can write to its output queues (to allow cyclic process networks),
2. infinite loop:
  - (a) read the inputs from the input queues,
  - (b) compute the outputs,
  - (c) store the results in the output queues.

When a process reads from an input FIFO queue, it can read (*get*) several tokens at a time and remove (*take*) another number of tokens. This is a common extension to the process network model that can be expressed easily by the original model.

These restrictions allow us to easily represent complex applications based on an assembly of components. This coarse grain view of the application is better suited to a performant execution on a network of computers than a finer grain view which generates too much communications. This does not

forbid the component to be parallel and to execute on a parallel computer. Basically, this model is well suited to model computation intensive meta-applications.

### 2.3 Distribution

The distribution is done in a different way than in Jade [12]. To avoid a central point of failure and unnecessary copies, the communications are handled by the components themselves through half FIFO queues. These queues implement the blocking read needed by the process network model but, as they are bounded, the write may block also if the queue is full. This can create deadlocks<sup>1</sup>, but as the execution is fully distributed, deadlock detection is difficult. We propose that the user selects the maximum size of the FIFO queues at creation time as he has an idea of the expected behavior of his application and of the resource usage he is ready to pay to run this application. These queues run asynchronously with the process computation so as to mask the network overhead and allow the vectorization of the communications. See section 3 for more details on the implementation of the distributed FIFO queues and the hybrid data-driven, demand-driven communication protocol.

The fact that the FIFO queues are bounded also allows for an incremental development where computation can start even if the application is not complete. When all the output queues are full, the computation is suspended and can restart as soon as a consuming component is attached to the not-yet-connected output queues. More details about the dynamicity of the component graph is given in section 5.

### 2.4 Distributed Heterogeneous Components with CORBA

To deal with the heterogeneity of the connected components of the application, we have chosen to use CORBA [10]. The interoperability features of CORBA allow us to build distributed components which communicate through distributed FIFO queues without concern for their implementation language. All the user has to do is to provide an IDL<sup>2</sup> interface for its computing functions. The rest of the code generation is done by an automatic code generator we have developed [1] and the CORBA tools (IDL compiler, libraries, etc).

Other component based architectures are currently being defined, such as the CORBA Component Model (CCM) [9], the Common Component Architecture (CCA) [2] and the Parallel Application Workspace (PAWS) [7].

---

<sup>1</sup>Artificial deadlocks can only appear in cyclic graphs which are uncommon at the coarse grain programming level we propose.

<sup>2</sup>IDL is the Interface Definition Language used to define the exported interfaces of the CORBA objects.

But none of them hides the network transparency by using FIFO queues, the components communicate directly with each other without transparent communication latency hiding.

### 3 Distributed FIFO Queue

We describe in this section our implementation of a distributed FIFO queue. We focus here on efficient communications and transparency of the distribution. The manipulation of these FIFO queues is done by the usual `get`, `put` and `take` methods to respectively remove, insert and read tokens (or data elements) from the queue. To achieve this transparency, we have split the queue in two parts which communicate over the network. These two half-queues manage both the storage of the data and their communication.

#### 3.1 Interface

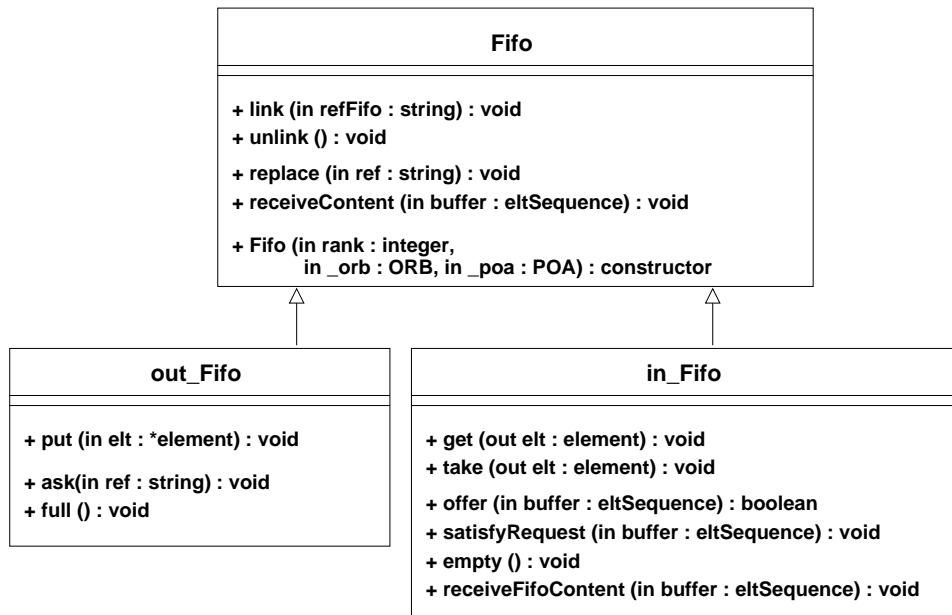


Figure 1: Class Diagram for the distributed FIFO queue

The FIFO queue class diagram is shown in figure 1. This class has the `get`, `put` and `take` methods used to interact with the queue, hiding the distribution.

The other methods are used to manage the interactions between the two half-queues or to initialize the queue. The `link` and `unlink` methods permit to create or remove (in the replacement case) a link between two half-queues.

The `replace` and `receiveContent` methods are used to replace a half-queue by another: the half-queue to be replaced receives a `replace` call with the reference of the new one, it then sends its content by a `receiveContent` call on this last one.

From the initial FIFO class, we derive two classes which are the output FIFO queue (first half or producer) and the input FIFO queue (second half or consumer). The output half-queue contains results of data processing and implements a method to receive token requests (`ask`) from the linked input half-queue. On the other hand, the linked input half-queue contains data to be processed and implements methods to receive (`satisfyRequest` or `offer`) data from the corresponding output half-queue.

### 3.2 Communication Protocol

The communication protocol is based on the exchange of data between the half-queues. A communication can be triggered by any of the two half-queues in a hybrid data-driven, demand-driven protocol. This protocol is completely distributed, no central authority directs the communication. Each FIFO queue handles its data transmission independently of the rest of the process network.

To manage the communications, two thresholds on the number of elements of the FIFO queue have been defined:

- a maximal threshold (for the producer half-queues) which indicates that offering a part of its tokens is necessary to avoid overloading,
- a minimal threshold (for the consumer half-queues) which indicates that it is necessary to ask the linked producer half-queue for some tokens.

When the length of the output half-queue exceeds the maximal threshold, it sends an `offer` request to the linked input half-queue. If this one reaches its maximum capacity, sending `offer` requests is momentarily suspended and the input half-queue responds by sending a `full` request to signal that it is full and can't receive data. The initiating output half-queue will then cease to `offer` data until it receives an `ask` request. Of course, if the capacity of the input half-queue was not reached, this one doesn't respond, and the output half-queue can continue to send `offers`.

In the other case, when a minimal threshold is detected, the input half-queue is alerted. It sends an `ask` request to the linked output half-queue. This one makes a read operation in its data (the size of read data is calculated according to some criteria taking into account the size of the output half-queue) and responds with a `satisfyRequest` method invocation. If the output half-queue does not have enough data, it responds with `empty`, and the input half-queue will not ask for anything until it receives an `offer` request. This avoids numerous requests without answers.

Obviously a good choice of these thresholds is important. As this choice depends on the application, we will not discuss it here and leave an in-depth study of these parameters for future work.

## 4 Distributed Processes as Components

We describe here the internal structure of a component, embedding the user provided computation function and the half-queues connecting this process to others in the process network.

### 4.1 Component Structure

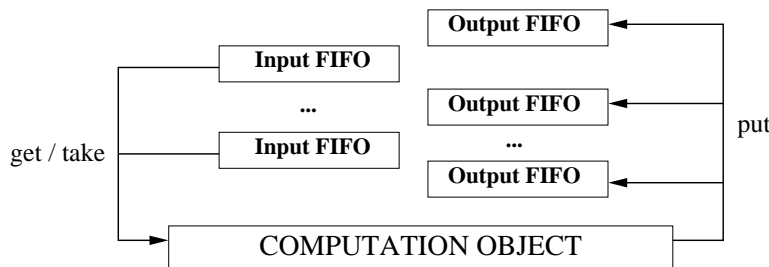


Figure 2: Component Structure

In our model, each process is a component which encapsulates different objects. In each component, there are three object types as shown in figure 2: a computation object to carry out the processing of the input FIFO queue elements and one or more input and output half-FIFO-queue objects. Each of these objects is a thread inside the component process and thus they run concurrently. The class diagram corresponding to this structure is given in figure 4.

The computation object handles the FIFO queues by calls to the `get`, `take` and `put` methods which respectively indicate the consumption, read and production operations.

The Computation class diagram is shown in figure 3. This class implements the *computation\_Interface* that defines the methods used to suspend or resume the computation, and a method to receive the internal state of another component. The CORBA type `any` that can hold the value of any IDL type is used to transfer the internal state. The actual representation of this state is the responsibility of the programmer.

### 4.2 Asynchronism

The computation in the calculation object and the communication in the management object are executed concurrently, each object being a thread.

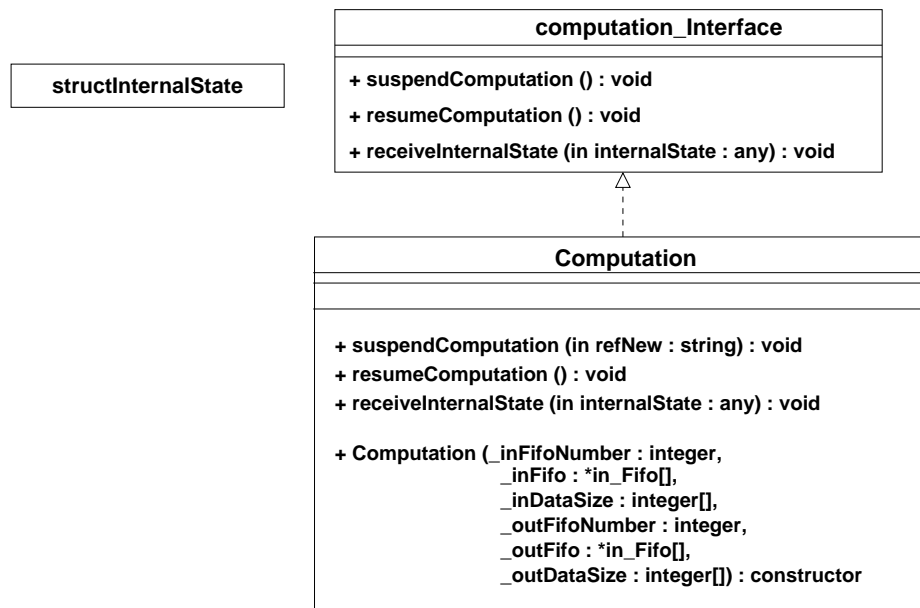


Figure 3: Class Diagram for the Computation object

However, to preserve the consistency of the data, we use critical sections to lock the concurrent accesses to the FIFO queue elements. To make the accesses to the FIFO queues more flexible, the critical section concerns the accesses to one element. We have also taken into account the performance aspect by eliminating the data copying between the different objects.

The computation inside the component and the communications with the other connected FIFO queues are done asynchronously. To do that, the requests are asynchronous and asking for data or sending data is not blocking for the computation.

## 5 Dynamic Component Graph

As said before, a process network is represented by a component graph. This graph is dynamic. This dynamicity is seen under three aspects:

- The first is the application's incremental development. It allows the construction of the application by adding available components interactively.
- The second is to be able to replace, during the execution, a component by another (for example to change the calculation function or to use a more performant or specialized component)<sup>3</sup>.

<sup>3</sup>we focus on long running applications, so this feature is important

- The third is to migrate interactively a component from a computer to another for various reasons such as hardware upgrading or load balancing.

## 5.1 Interactive Console

To control the components, a console has been developed. It consists of a program which controls the component connection and execution by the use of a simple language. The presence of a manager program is contrary to the peer-to-peer character of component systems. However, the console is minimal and serves only two roles, collaboration control and component replacement. All the communications between the components are done without involving the console through the distributed FIFO queues presented in section 3. One of our future objectives is to add automatic capabilities to this console so that it adapt the mapping of the components in function of the load of the participating computers for example.

The component links that form the process network are made interactively by this console to which all the components must be connected just after their launching. The use of a console allows more flexibility in the connection choice and a dynamic control of the components.

The Process Network class diagram is shown in figure 4. In this diagram, we see mainly methods and data structures that are visible by the CORBA ORB.

The FIFO queue implements the `basic_Management` interface. Additionally, `out_Fifo` and `in_Fifo` implement respectively the `out_Management` and `in_Management` interfaces. The Console implements the `console_Management` interface which contains the method used by the components to bind the console at the beginning of their execution. In this method, the console retrieves the component name, FIFO queue informations and computation object reference. FIFO queue informations are stored into a sequence of structures containing the FIFO queue references and types.

## 5.2 Console Command Language

This console allows the user to interactively build and control its graph with the following commands:

- *list*: list the active components on the system;
- *init id*: launch the fabric object on a component;
- *link id1 id2*: link two FIFO queue between them;
- *suspend id*: suspend the computation on a component;
- *replace id1 id2*: replace a component by another;

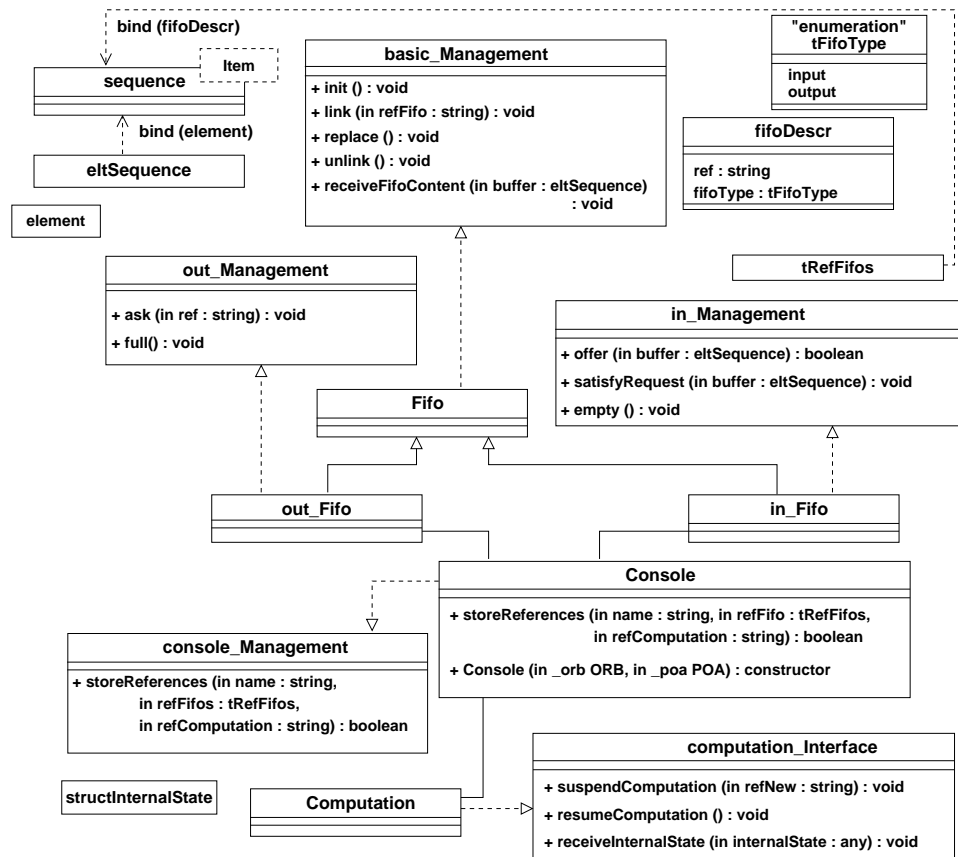


Figure 4: Class Diagram for the Components

- *show*: to see all the links between the FIFO queue;
- *length*: to get the number of tokens in all the FIFO queues;
- *fifolen id*: to get the number of tokens in given FIFO queue;
- *state*: to get informations about the system;
- *check*: to verify the integrity of all the components.

### 5.3 Replacement of a Component

The component replacement is done interactively from the console. The replacement scenario is the following:

1. The FIFO queues connected to the component that will be replaced are prevented to send or to request anything. This avoids the loss of data or the waiting of a response that will never come.
2. The console then asks the component that must be replaced to suspend the computation by invoking the `suspendComputation` method in the computation object. When this component receive this request and suspend really the computation, it sends its internal state to the new component computation object.
3. After that, FIFO queues of the component to be replaced receive the order from the console to transfer their contents to the new component FIFO queues.
4. The other components of the graph are connected to the new one, by relinking the different FIFO queues and finally computation is launched.

## 6 Signal Processing Application

We illustrate our model on a representative example of signal processing.

### 6.1 Application Description

The application (see figure 5) consists of providing frequencies and location correlations (so called *beams*) from a continuous stream of data. It is based on elementary signal transformations: FFT (Fast Fourier Transformation) and discrete integration.

- The *Hydrophones*, an ( $h = 512 \times T = \infty$ ) array, **HYDRO**, is the input of the application. It delivers a continuous stream of data from a set of 512 captors.

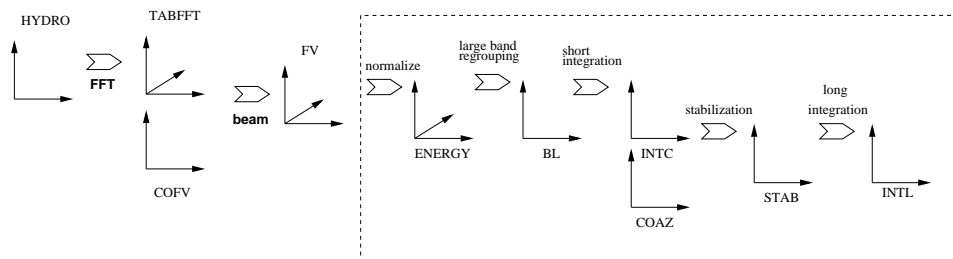


Figure 5: Application Description

- The first task computes FFT for each captor and period of 512 units of time. It produces TABFFT, a three dimensional array ( $512 \times 256 \times \infty$ ).
- The second task computes a beam for each period, frequency and set of captors. It outputs *Beams*, an ( $t = 128 \times T = \infty \times f = 200$ ) array, FV.
- The beams are then treated successively by different functions, to finally extract characteristic frequencies. The input and output array sizes of the different functions are:

Function	Input array	Output Array
Normalization	$(128 \times \infty \times 200)$	$(128 \times \infty \times 200)$
Large band regrouping	$(128 \times \infty \times 200)$	$(128 \times \infty)$
Short integration	$(128 \times \infty)$	$(128 \times \infty)$
Stabilization	$(128 \times \infty)$	$(128 \times \infty)$
Long integration	$(128 \times \infty)$	$(128 \times \infty)$

## 6.2 Process Network Specification

From the application description, it is easy to define the corresponding process network. Due to the high speed of the five last tasks compared to the two first ones, we group them in just one process. We obtain three processes in a linear pipe-line. The first task computes an ( $h = 512$ ) array, **HYDRO** and so on. The time dimension (with infinite size) becomes the successive tokens produced as input of the first process.

The sequential program used for performance analysis breaks the infinite dimension too, but only one process executes the complete application.

## 6.3 Performance Measures

For the performance measures, we have used three networked workstations. All the input tokens are almost available at the beginning of the run. The

acquisition of the hydrophone values is considered as instantaneous and the pipe-line works in a saturation state. We have done several measures (see table 1): the sequential program running on a 1 GHz Pentium III computer, the three component programs on two 1 GHz Pentium III computers and one 266 MHz Pentium II computer linked by a 10 Mb/s Ethernet bus. All the times measured here are wall clock times and it should be noted that the workstations were lightly used by other processes.

# of tokens	Sequential P3	CORBA P3	CORBA cluster	1st Task P3	2nd Task P3	3rd task P2
5000	51 s	52 s	32 s	22 s	26 s	10 s
10000	100 s	104 s	62 s	41 s	48 s	20 s
15000	154 s	168 s	97 s	56 s	68 s	29 s
20000	203 s	214 s	123 s	75 s	91 s	38 s
25000	246 s	271 s	148 s	94 s	115 s	48 s
30000	309 s	322 s	173 s	118 s	131 s	57 s
35000	362 s	382 s	198 s	141 s	156 s	65 s
40000	408 s	430 s	230 s	166 s	180 s	74 s
45000	459 s	496 s	263 s	184 s	205 s	83 s
50000	516 s	570 s	290 s	210 s	236 s	93 s
2 500 000	26120 s	N.A.	15980 s	11270 s	12590 s	4570 s
2 750 000	27580 s	N.A.	17730 s	12910 s	13140 s	5120 s
3 000 000	29050 s	N.A.	19096 s	13550 s	13800 s	5580 s

Table 1: Application performances

Working under saturation state, the ratios between the sequential and the CORBA solutions stay similar. It means that the queue management time is proportional to the size of the queue. The overhead cost due to the CORBA layers without communications is less than 10% (on the same monoprocessor P3).

The speed-up allows to increase the frequency of acquisition in the same proportion, see figure 6 that shows the relative times of the tasks with respect to the total execution time.

**Dynamic redistribution and FIFO queue length evolution.** Table 2 and figure 7 show the evolution of the FIFO queue length between the two first components. During the execution under saturation state, the second component which runs on a Pentium II 266 MHz PC, has been replaced by another one which runs on a faster machine (Pentium III 1 GHz PC). We note that the FIFO queue length has been decreased immediately after the replacement. A bigger acquisition frequency should be supported on this new cluster of PCs.

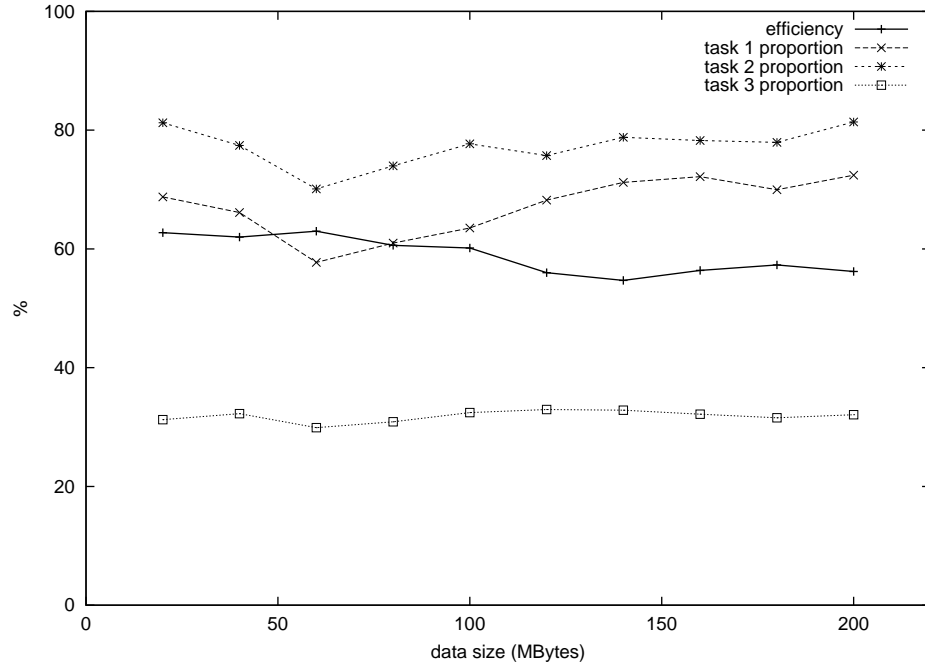


Figure 6: Cluster execution efficiency - load unbalancing of the 3 tasks

Data size (Mbytes)	P3-P2-P2	P3-P3-P2
20	80 s	47 s (replacement after 30 s)
40	166 s	90 s (replacement after 60 s)
60	258 s	158 s (replacement after 90 s)
72	302 s	188 s (replacement after 120 s)

Table 2: Dynamic redistribution: the second task migrates from a P2 to a P3 during the run

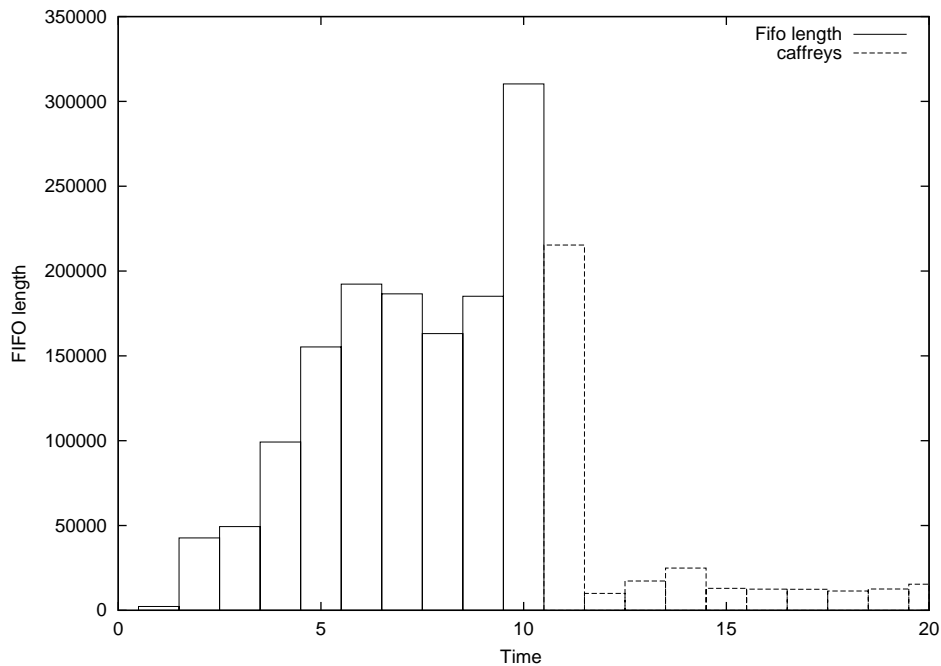


Figure 7: Fifo queue length evolution during the redistribution

## 7 Conclusion

The process networks are specially suited to the development of intensive signal processing applications or scientific computation. Those are built on the concept of FIFO queues which convey tokens between processes. This execution model becomes a preferential target of a specification model dedicated to applications of this type for network architectures such as PC clusters.

We have proposed a distributed execution of this model on the top of the CORBA middleware. It guarantees interoperability between the languages and the dynamicity not only of the application specification but also of its placement on a network. The results obtained are promising, they show that the distributed implementation of a signal processing application makes possible to satisfy higher frequencies of acquisition according to a higher speed-up.

The mapping and the control of the scheduling of the processes is carried out explicitly via a console interface. That makes feasible to start an application respecting the process network model before this one is not completely specified. It also allows the migration of processes between two machines, for example to ensure a better load balance. Finally it authorizes the substitution of one process by a more efficient one. This can be done

during the execution of the application, without untimely shutdown. Thereafter, it would be appropriate, by the means of informations received during the execution to set up an automatic or semi-automatic adaptive system which carries out these transformations during the execution. A decision tool would then be coupled with our interface.

For applications where the processing time of the various processes strongly varies, the improvement of the performances and thus of the acquisition frequencies in the case of the signal processing, could be obtained by association of several processes to the same input queue.

Finally experiments under a stabilized acquisition mechanism should make possible to build performance cost function of a placement by taking into account the times of acquisition, the sizes of the input data as well as the processing times of each process.

## References

- [1] Abdelkader Amar, Pierre Boulet, and Jean-Luc Dekeyser. Assembling dynamic components for metacomputing using CORBA. In *Parallel Computing 2001*, Naples, Italy, September 2001.
- [2] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, , and Brent Smolinski. Toward a Common Component Architecture for high-performance scientific computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, 1999.
- [3] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California at Berkeley, 1993.
- [4] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzter, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, June 2000.
- [5] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.
- [6] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977. Proc.IFIP Congress.

- [7] Kate Keahey, Pat Fasel, and Sue Mniszewski. PAWS: Collective interactions and data transfers. In *High Performance Distributed Computing (HPDC-10)*, August 2001.
- [8] Edward A. Lee. *Overview of the Ptolemy Project*. University of California, Berkeley, March 2001.
- [9] Raphaël Marvie, Philippe Merle, and Jean-Marc Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, Antwerp, Belgium, September 2000.
- [10] Object Management Group, Inc., editor. *Common Object Request Broker Architecture (CORBA), Version 2.6*. [http://www.omg.org/technology/documents/formal/corba\\_iiop.htm](http://www.omg.org/technology/documents/formal/corba_iiop.htm), December 2001.
- [11] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.
- [12] Darren Webb, Andrew Wendelborn, and Kevin Maciunas. Process networks as a high-level notation for metacomputing. In *Workshop on Java for Parallel and Distributed Computing (IPPS)*, Puerto Rico, April 1999.
- [13] Darren Webb, Andrew Wendelborn, and Julien Vayssière. A study of computational reconfiguration in a process network. In *IDEA7*, Victor Harbour, South Australia, February 2000.