

Distributed Process Networks Using Half FIFO Queues in CORBA

Abdelkader Amar, Pierre Boulet, Jean-Luc Dekeyser, Frans Theeuwen

▶ **To cite this version:**

Abdelkader Amar, Pierre Boulet, Jean-Luc Dekeyser, Frans Theeuwen. Distributed Process Networks Using Half FIFO Queues in CORBA. [Research Report] RR-4765, INRIA. 2003. inria-00071821

HAL Id: inria-00071821
https://inria.hal.science/inria-00071821

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Distributed Process Networks Using Half FIFO Queues in CORBA

Abdelkader Amar — Pierre Boulet — Jean-Luc Dekeyser — Frans Theeuwen

N° 4765

February 21, 2003

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--4765--FR+ENG



Distributed Process Networks Using Half FIFO Queues in CORBA

Abdelkader Amar*, Pierre Boulet*, Jean-Luc Dekeyser*, Frans Theeuwen†

Thème 1 — Réseaux et systèmes
Projet DaRT

Rapport de recherche n° 4765 — February 21, 2003 — 15 pages

Abstract: Process networks are networks of sequential processes connected by channels behaving like FIFO queues. These are used in signal and image processing applications that need to run in bounded memory for infinitely long periods of time dealing with possibly infinite streams of data. This paper is about a distributed implementation of this computation model. We present the implementation of a distributed process network by using distributed FIFOs to build the distributed application. The platform used to support this is the CORBA middleware.

Key-words: Process Network, Distributed Process Network, CORBA

This work has been supported by the ITEA 99038 project, Sophocles.

* Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Cité Scientifique, 59655 Villeneuve d'Ascq cedex, France
† Philips ED & T / Synthesis, WAY 3.13, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

Unité de recherche INRIA Futurs
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Réseaux de processus distribués avec des demies files d'attentes en CORBA

Résumé : Les réseaux de processus sont des processus séquentiels communicant uniquement par des canaux se comportant comme des files d'attentes. Ils sont utilisés pour modéliser des applications de traitement du signal ou de l'image devant fonctionner en mémoire bornée pendant des périodes de temps potentiellement infines et traitant des flux de donnés eux-aussi potentiellement infinis. Cet article s'intéresse à l'implémentation distribuée de ce modèle de calcul. Nous présentons l'implémentation distribuée de réseaux de processus grâce à l'utilisation de files d'attentes distribuées. La plate-forme logicielle utilisée est l'intergiciel CORBA.

Mots-clés : Réseaux de processus, réseaux de processus distribués, CORBA

1 Introduction

Kahn Process Networks [6, 7] are well adapted to model many parallel applications, specially dataflow applications (signal processing, image processing). In this model, processes communicate only via unbounded first-in first-out (FIFO) queues.

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 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) [9] 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).

In addition of the heterogeneity, our implementation presents the following characteristics:

- automation of data transfer between distributed processes
- dynamic and interactive linking of the processes to form the data flow
- hybrid data-driven, demand-driven data transfer protocol, with thresholds for load balancing
- the implementation was carried out such as to enable a distributed or local execution without any change to the program source.

This paper is organized as follows. In section 2, we motivate our approach and we present our implementation. Section 3 describes a process network deployment and distributed execution. The transfer strategies (demand and data driven) are detailed in section 4. And we finally outline our conclusions and plans for future work in section 5.

2 Design and Implementation

2.1 Related Works

The process network model has been proposed by Kahn and MacQueen [6, 7] to easily express concurrent applications. Processes communicate only through unidirectional FIFO

queues. Read operations are blocking. 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.

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 [10] 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 [10] to allow a complete, unbounded execution of process networks when possible.

In the context of a distributed execution of a process network, the process execution is inherently asynchronous. We have thus chosen a completely asynchronous scheduling: each process runs in its own thread that is scheduled by the underlying operating system. As explained by Parks and Roberts in [11], who use a similar scheduling technique, using bounded communication queues allow for a fair execution of the process network. This blocking write when the output queue is full can lead to deadlocks. Determining if the queue length is large enough to avoid such deadlocks is undecidable. We provide a way for the user to modify this length at runtime.

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 [13, 14]. To our knowledge, only the Jade/PAGIS implementation and the one by Parks and Roberts [11] are distributed. Parks and Roberts use the Java Object Serialization to automate the distribution of the network processes while we use a central console to deploy the processes. In Jade, all communications proceed through a central communication manager while in Parks and Roberts' and our implementation the processes communicate directly. This allows a greater scalability. Only our implementation allow the coupling of processes written in different languages.

2.2 Design Directions

The design of our distributed process network implementation was done so as to:

- enable users to simulate their network model quickly and effectively,

- keep the source compatibility with the Yapi library: this library developed by Philips [4] implements the process network model for a local execution, Yapi is a C++ library which focuses on signal processing applications,
- enable the distributed and the local execution without any change to the program source.

The idea of the Yapi syntax is to group processes into process networks. The processes communicate via ports. These ports are linked by point-to-point unidirectional FIFO queues. A process network can be seen as a process and used in the same way. This hierarchical construction allows an easy modeling of complex applications.

A shared memory multiprocessor implementation of Yapi is demonstrated in [5]. For this study, we have completely reengineered Yapi to be able to distribute any application over a CORBA bus without any change to the application code. In our implementation the communications are hidden to the programmer who can though configure the data transfer parameters.

2.3 Implementation Sketch

The class diagram is as shown in figure 1. The basic class from which all the other classes inherit is *Id*. This class is used to store the object name and a reference to its parent. The *Fifo* class is the communication channel. To be able to handle the FIFOs, we define ports. They are represented by two classes: *OutPort* and *InPort*. They are associated to one FIFO, and are used to respectively read and write in the *Fifo*. Each port is bound exactly to one FIFO to respect Kahn's rules. To allow conversion between FIFOs and ports, the three classes inherit from a common class. *Fifo* and *OutPort* inherit from *Out* class, and *Fifo* and *InPort* from *In* class. The *ProcessNetwork* class represents the process networks. It is the class that creates and initializes the processes and the FIFOs. Since the process networks can be hierarchical, a *ProcessNetwork* object can create other *ProcessNetwork*. Finally, the class *Process* is used to define process type. To exploit the parallelism of the model, each process is executed in a thread.

3 Process Network Distribution

3.1 Deployment

To control the distributed process networks, a console has been developed. It consists of a program which controls the processes connection and execution by the use of a simple language. It also provides a frontend used for monitoring. 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 as collaboration control. All the communications between the components are done without involving the console through the distributed FIFO queues

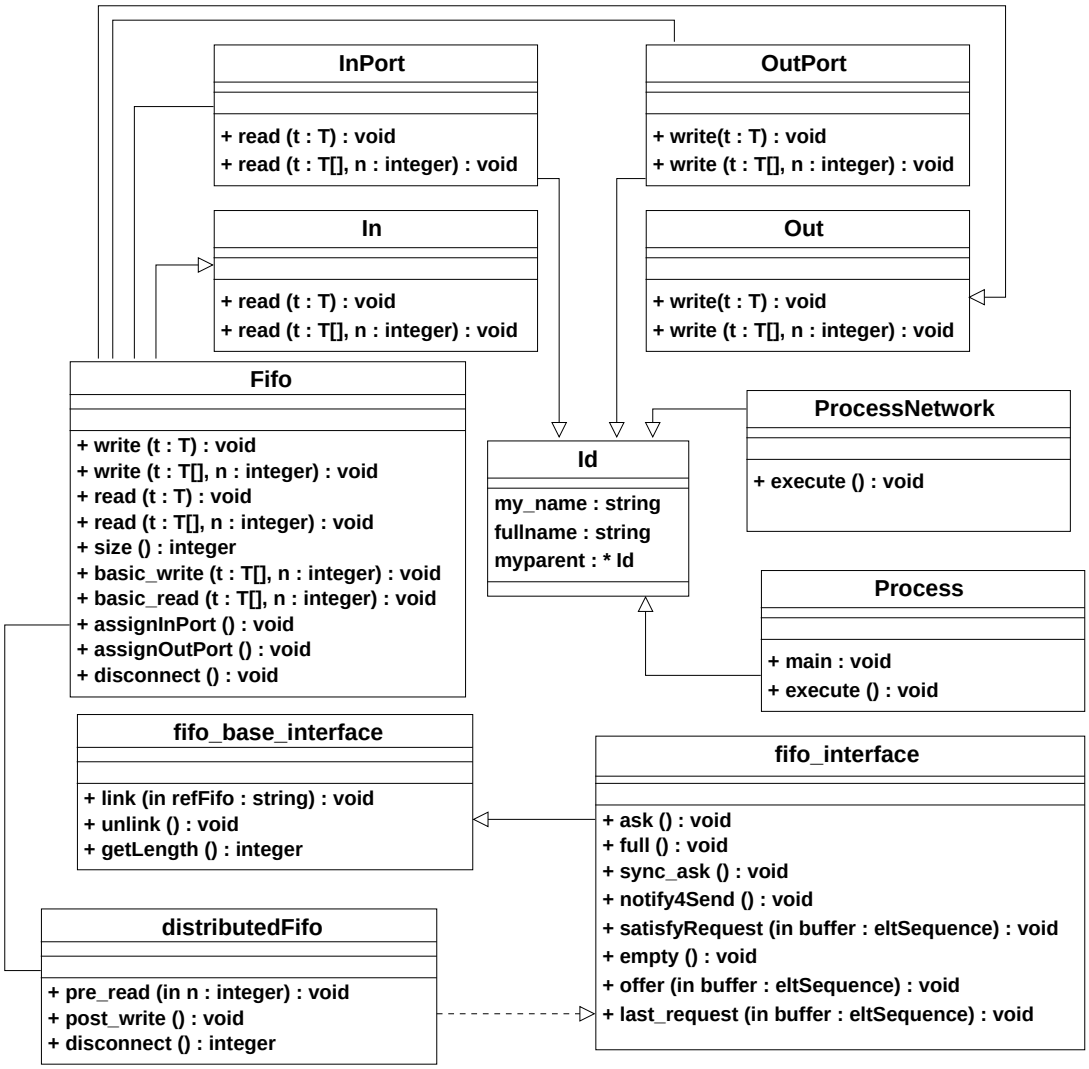


Figure 1: Class Diagram for the Distributed Process Network Framework

presented in section 3.2. The FIFO links that form the process network are made interactively (or via a script) by this console. The use of a console allows more flexibility in the connection choice and a dynamic control of the components and the communication parameters.

The use of an interactive console and the fact that the FIFOs are bounded also allow for an incremental development where computations 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.

3.2 Distributed FIFOs

The FIFO queues are completely distributed, and distributed process networks communicate directly, without a central point contrarily to what is done in Jade [13]. These queues implement the blocking read needed by the process network model but, as they are bounded, the write may block also if the FIFO queue is full. A deadlock can appear, but as the execution is fully distributed, deadlock detection is difficult and has not been implemented.

To guarantee the code reuse with our implementation, the implementation of the distributed FIFOs must be done without programmer intervention or code change. This was done by encapsulating the distributed FIFOs (the CORBA object) in the FIFOs. The figures 2 and 3 show the structure of the FIFO objects. For the programmer, no difference exists between the local and the distributed FIFO queue. To determine if the FIFO is a distributed FIFO or not, the runtime uses its ports. When a FIFO is local to the program, it has an input and output port. On the other hand, the distributed FIFO is represented by two half FIFOs, the output FIFO queue (producer side) and the input FIFO queue (consumer side). Each half FIFO has one port (an input port for the input FIFO, and an output port for the output FIFO) and should be linked to the other half FIFO. The runtime uses this property to activate the CORBA object only on the distributed FIFOs, and thus, the FIFO distribution is implicit for the programmer.

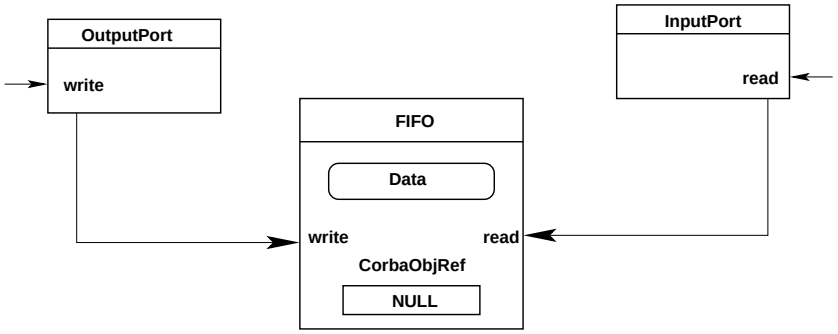


Figure 2: Local FIFOs implementation

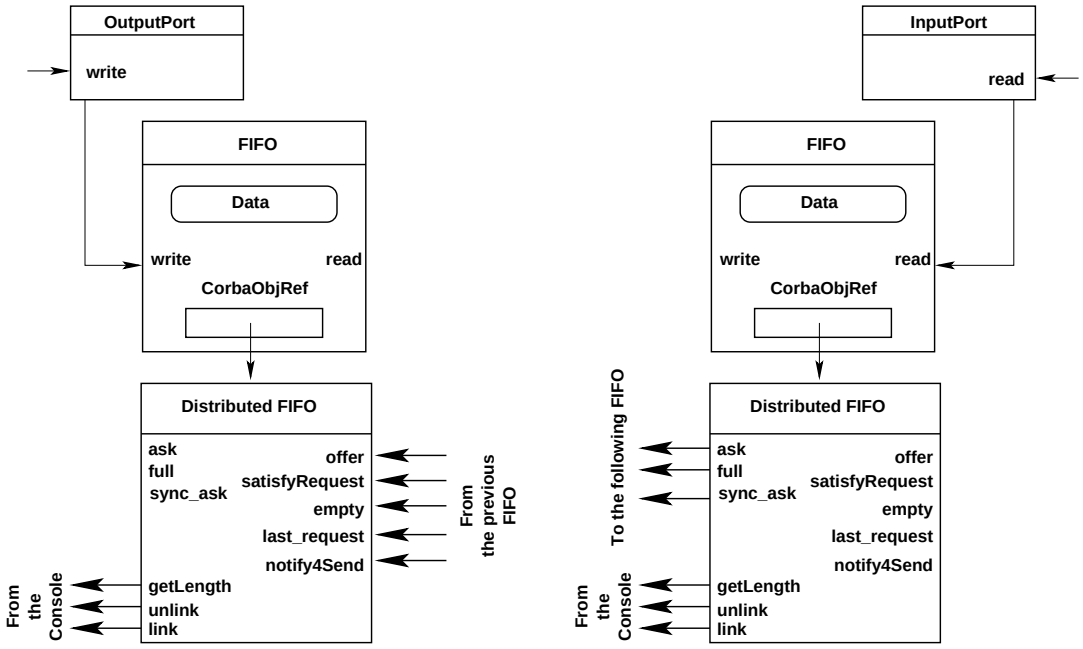


Figure 3: Distributed FIFOs implementation

To exchange data between distributed FIFO, we implement a communication protocol based on thresholds. This protocol which will be detailed in section 4 is a hybrid of data-driven and demand-driven transfers.

4 Transfer strategies

In this section, we are interested in the data transmission between FIFOs. The transfer strategies were implemented having in mind the minimization of communications between FIFOs, the overlapping of communications by computations, and the load balancing.

4.1 Threshold Based Protocol

The communication protocol governs the exchange of data between the FIFOs. A communication can be triggered by any of the two FIFOs 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 FIFO queues) which indicate that offering a part of its tokens is necessary to avoid overflowing,
- a minimal threshold (for the consumer FIFO queues) which indicates that it is necessary to ask the linked producer half-queue for some tokens to avoid idle time.

The distributed FIFO IDL interface described bellow defines the methods used for the initialization, the interaction with the other FIFOs, and the recovery of informations. This interface which is implemented by the *distributedFifo* class is not accessible to the user, but used implicitly by the distributed FIFO queue to communicate with the other distributed FIFO queue and the manager.

```
interface fifo_base_interface
{
  void link(in string refFifo);
  void unlink();
  unsigned long getLength();
};

interface fifo_int : fifo_base_interface
{
  oneway void ask();
  oneway void full();
```

```
  boolean offer(in eltSequence buffer);
  oneway void satisfyRequest
                (in eltSequence buffer);
  void empty();
  void sync_ask(inout eltSequence buffer);
  bool notify4Send();
};
```

The `link` and `unlink` permit to create or remove a link between two FIFOs. The `ask` and `satisfyRequest` methods are used respectively by the input FIFO to ask for data, and the output FIFO to response to an `ask`. `sync_ask` is a synchronous version of the `ask` method. The `full` and `empty` methods are used to indicate the FIFO state, while the `offer` and `notify4Send` are used by the output FIFO to send data to the following FIFO respectively with and without event notification. It should be noticed that communications are vectorized. Indeed we transfer token sequences together and not each token at a time.

4.2 Transfer Policies

The two next paragraphs present the two transfer policies. It can be demand driven with event notification, and data driven without event notification. The other cases are implemented but not used by the runtime (demand driven without event notification by `sync_ask` method and data driven with event notification by `notify4Send/sync_ask` methods). However, the programmer can modify the transfer strategy by calling the `changePolicy` method. Below, we suppose that the two FIFOs *F_out* and *F_in* are linked, the first one being the output FIFO and the second one, the input FIFO.

Demand driven When the transfer is *demand driven*, it is with event notification. When the token number becomes lower than the minimal threshold, The *F_in* FIFO notifies (`ask`) the *F_out* FIFO which responds by sending data (`satisfyRequest`). The demand driven without event notification protocol was not used because such requests are synchronous and the process would be blocked until the availability of the requested data. With the event notification, the process can continue computing if there are still some tokens in its input FIFO (*F_in*).

Data driven When the transfer is *data driven*, it is without event notification. When the token number in the FIFO *F_out* exceeds its maximal threshold, the FIFO sends data (`offer`) to the *F_in* FIFO. To avoid the overloading of the *F_in* FIFO, the request returns a result. This result indicates if the receiver FIFO can receive or not other requests from the FIFO *F_out*. If the *F_in* FIFO is full, the *F_out* FIFO will not send any data until it receives a request for data from *F_in*.The *offer* requests being asynchronous (return a result), a separated thread manages this communication mode. The data driven with event notification protocol involves two remote requests, and the use of this mode with the demand driven with event notification does not guarantee the order of the data reception.

Summary Figures 4[1] and 5 represent the state diagrams of the communication protocol as seen by the two half FIFO queues. In these figures, the events are methods called from the other FIFO. The `put` and `get` methods are blocking respectively when the output FIFO is full and when the input one is empty. As the number of tokens send is bounded (the size parameter), the overflow of the input FIFO is detected after each data reception. The input FIFO notices the output one (false return to an `offer` call or `full` call) when it can no longer receive a complete data transmission.

All the communications are hidden to the programmer, only the links between the distributed FIFOs determine the data exchange between processes. For the distributed FIFOs, the communications can be triggered in two ways:

1. Inside the process: according to whether the distributed FIFO is an input or an output FIFO, the communication triggering is done in two ways:
   - when a read operation is executed in an input FIFO (figure 6), a pre-read processing is started to ask eventually for data (if the number of tokens in the FIFO is below the minimal threshold).
   - on the other hand, when a write operation is executed in an output FIFO (figure 7), post-write processing is started to eventually send data to the following FIFO (if the number of tokens in the FIFO is above the maximal threshold).
2. Outside the process: this is done directly by the remote invocations from the linked FIFOs which ask for data or send data.

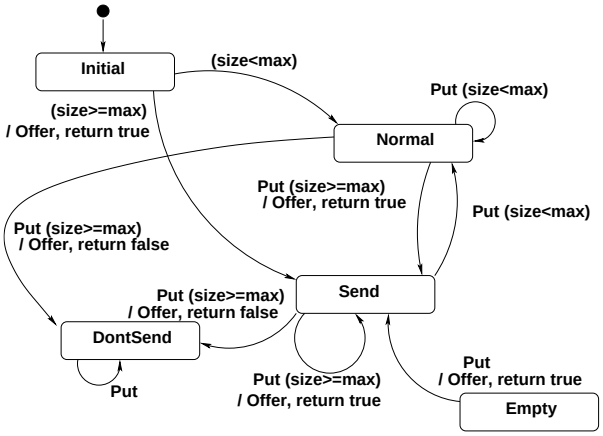
However, to adapt the communications to particular applications, some methods were implemented to give to the programmer the possibility to configure the communications, by setting some communication parameters: minimal and maximal threshold, maximal FIFO capacity, exchange buffer's maximal size.

5 Conclusion

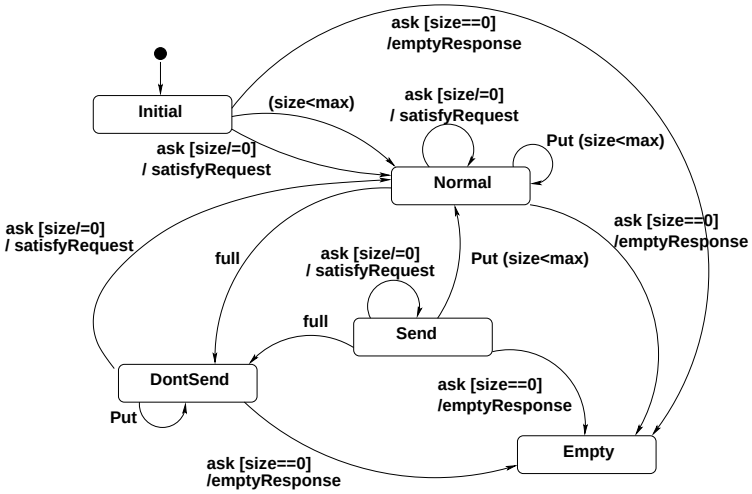
We have described here our distributed implementation of process networks using CORBA. This implementation is based on the assembly of software components to form a distributed application. Each software component represents a process network which can be hierarchical, and thus exploits the parallelism of the process network model. The CORBA architecture choice is motivated by the handling of the hardware and software heterogeneity. The mapping and the control of the scheduling of the processes is carried out explicitly via a console interface. That makes it feasible to start an application respecting the process network model before its complete implementation.

There are a lot of important issues we plan to investigate in the near future. The most important is the dynamicity aspect of the network. Our previous implementation of a

[1] The transition from Empty to Send allows the termination of the computation in case the output FIFO does not reach its threshold.

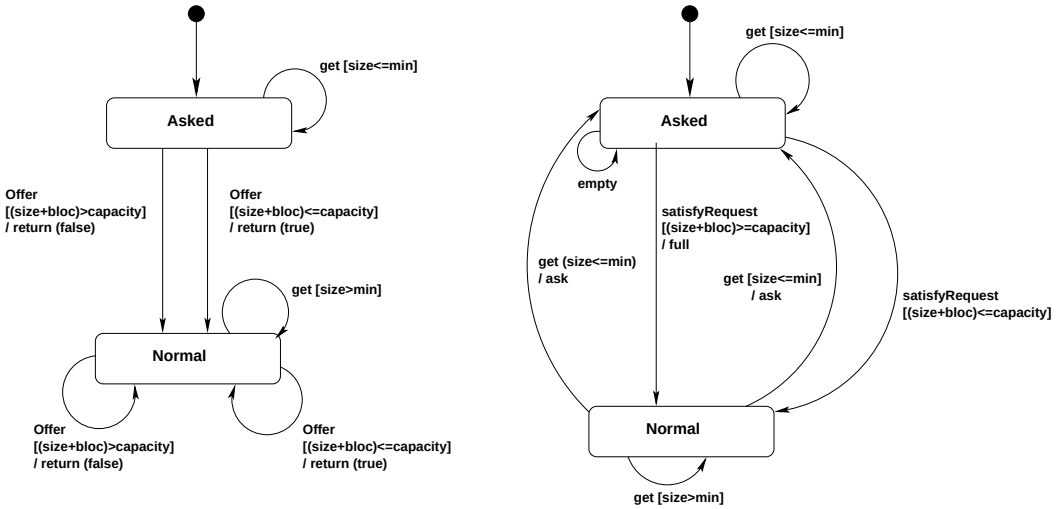


demand driven part



data driven part

Figure 4: Output FIFO state diagram



demand driven part data driven part

Figure 5: Input FIFO state diagram

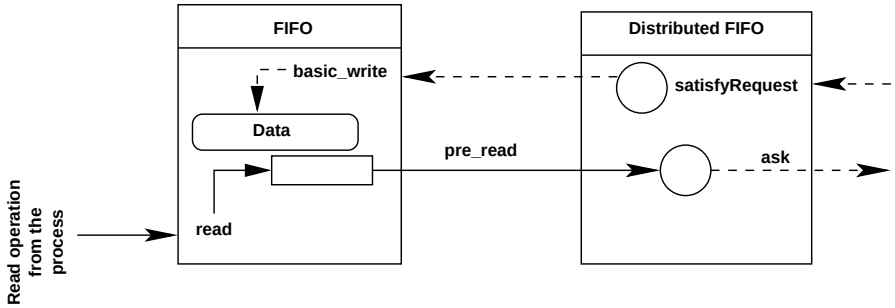


Figure 6: Read and pre-read operations in the distributed FIFO

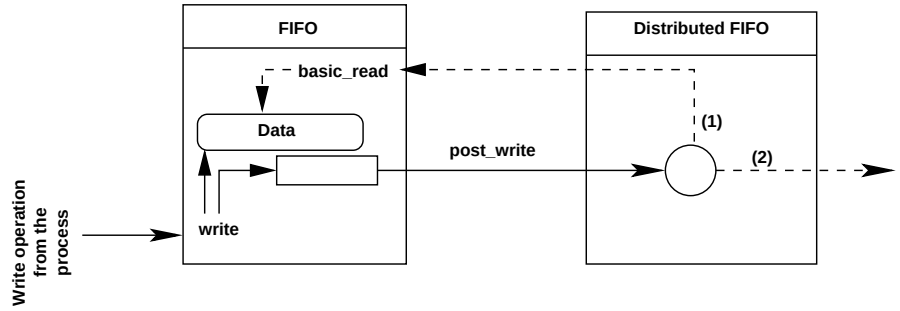


Figure 7: Write and post-write operations in the distributed FIFO

subcase of the process network model [1, 2] provides more dynamicity, where it is possible to migrate a process or to replace it by another, and our goal is to support these two dynamicity aspects in the general model. To achieve this, two difficulties must be overcome:

- How to retrieve the internal state of the processes, and how to resume the computation.
- How to retrieve the contents of the local FIFO queues.

Moreover, we are interested in the use of the process network model as an execution model for the applications of multidimensional signal processing based on Array-OL [12], which is a programming language dedicated to systematic signal processing.

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. Lecture Notes in Computer Science.

[2] Abdelkader Amar, Pierre Boulet, and Jean-Luc Dekeyser. Towards distributed process networks with CORBA. *Parallel and Distributed Computing Practice on Algorithms*, 2003. Special Issue on Parallel and Distributed Computing Practice on Algorithms, to appear.

[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. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, June 2000. ACM Press.

[5] Erwin de Kock. Multiprocessor mapping of process networks: A jpeg decoding case study. In *International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[6] 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.

[7] 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.

[8] Edward A. Lee. *Overview of the Ptolemy Project*. University of California, Berkeley, March 2001.

[9] Object Management Group, Inc., editor. *Common Object Request Broker Architecture (CORBA), Version 2.6*. `http://www.omg.org/technology/documents/formal/corba_iiop.htm`, December 2001.

[10] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[11] Thomas M. Parks and David Roberts. Distributed process networks in java. In *International Workshop on Java for Parallel and Distributed Computing*, Nice, April 2003.

[12] Julien Soula, Philippe Marquet, Jean-Luc Dekeyser, and Alain Demeure. Compilation principle of a specification language dedicated to signal processing. In *Sixth International Conference on Parallel Computing Technologies, PaCT 2001*, pages 358–370, Novosibirsk, Russia, September 2001. Lecture Notes in Computer Science vol. 2127.

[13] 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.

[14] Darren Webb, Andrew Wendelborn, and Julien Vayssière. A study of computational reconfiguration in a process network. In *IDEA'7*, Victor Harbour, South Australia, February 2000.



Unité de recherche INRIA Futurs
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399