Implementing atomic rendezvous within a transactional framework

Jean-Pierre Banâtre, Michel Banâtre, Christine Morin

Implementing Atomic Rendezvous within a Transactional Framework.

Jean-Pierre BANATRE
IRISA/INSA and INRIA-Rennes

Michel BANATRE
IRISA/INRIA-Rennes

Christine Morin
IRISA-Rennes

Campus de Beaulieu, 35042 RENNES cedex (FRANCE)

ABSTRACT

The transaction concept has been widely used as a powerful tool for the reliable structuring and programming of applications in several fields such as databases, distributed systems... In this paper, we address the problem of implementing the CSP rendezvous whithin a transactional framework. In fact, several authors have studied the problem of implementing CSP rendezvous on networks of machines but our transactional approach is original. Their research mainly concentrates on the implementation of a fair non-deterministic choice and assumes a correct functioning of processors and communication media.

In this paper, we propose an efficient transactional implementation of atomic rendezvous in presence of processor failures in a multiprocessor machine. Both atomicity and efficiency are obtained by using special hardware devices : high speed stable storages. .

Key words
CSP, rendezvous, atomicity, transactions, parallelism, stable storage.

1.Introduction.

Remote procedure call (RPC) appears to be a widely used paradigm for providing communication between programs written in a high level langage and running on different nodes [11]. When a remote procedure is invoked, the caller is suspended, the parameters are transmitted across network to the node where the callee is to be executed and after completion of the callee, the results are passed back to the caller which resumes its execution.

There are many attractive aspects to this communication facility : simple semantics, efficient implementation. The major issue faced by the system programmer of a RPC facility is the problem related with node and communication failures. In particular a precise semantics of a call has to be defined which determines policies to be obeyed after detection of a failure [3].

Another important paradigm for providing communication between programs (or processes) in a high level language is the rendezvous [6]. Two processes are involved in a rendezvous, a producer and a consumer. Production of an item and consumption of the same item are synchronized (the producer waits till the consumer is ready and vice versa). When both producer and consumer are ready, the information exchange takes place atomically and if the rendezvous is successful the two processes go on in parallel or execution of both processes is interrupted just before the rendezvous commands. This is the "all or nothing" property characterizing atomicity.

Several authors have studied the problem of implementing CSP rendezvous in a distributed context [2], [4]. They mainly concentrate on implementation of a fair non-deterministic choice and assume a correct functioning of processors and communication media. In this paper, we address the problem of efficient implementation of atomic rendezvous on a multiprocessor machine in presence of failures. Once the rendezvous to be done is chosen (we do not discuss this problem here), we guarantee that either the rendezvous is successful or nothing happens.

In the following, we consider rendezvous as it is defined in CSP, a language for Communicating Sequential Processes, proposed by Hoare [6]. For the purpose of this paper, the following description of input/output commands syntax and meaning is sufficient.

Communications between processes P_i and P_j ($i \neq j$) are expressed by the send and receive commands P_i ! x and P_j ? x respectively. **Output** command P_i ! x (in text of P_j) expresses a request to P_i to receive a value from P_j. **Input** command P_j ? x (in text of P_i) expresses a request to P_j to assign a value to the (local) variable x of P_i. Execution of P_j ? x and P_i ! y is synchronized (P_i **waits** at P_j ? x until P_j is ready at P_i ! y and vice versa) and results in assigning the value of y to x.

Section 2 deals with architectural issues. In particular, a stable storage device which is used intensively in order to implement atomicity is described. The general model of atomic

CH2807-6/89/0000/0119/$01.00 © 1989 IEEE

rendezvous implementation are presented in section 3 and the associate transaction management system is described in section 4. Finally section 5 contains a brief review and discussion.

2. Architectural issues.

2.1. Machine architecture.

Our reference machine is made out of N processors connected via a global bus. Each processor (e.g., M68030) can access a local memory and a special device (stable storage) which will be described later. The processor normally accesses its local memory and its stable storage through a local bus. Figure 1 illustrates this machine architecture.

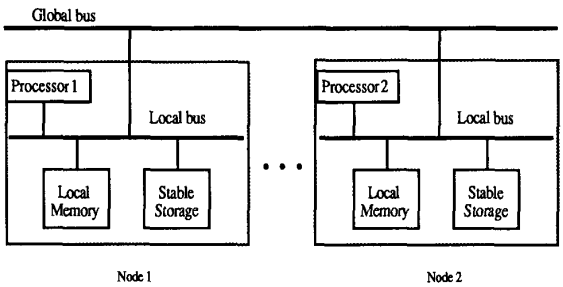

Figure 1 : Machine architecture.

In the following, we will refer to a processor, its local bus, its local memory and stable storage as a **node**.

2.2. Stable storage description.

2.2.1.Hardware architecture.

Stable storage provides memory that has **high probability of surviving processor and communication media failures**. As an abstraction, stable storage has the important property that writes are **atomic**, that is, they either happen or they don't happen. Lampson and Sturgis [8] have proposed an implementation of this abstraction some years ago. Their idea was to use two different disk drives with independent failure processors. Each logical page was represented by two physical pages, one on each disk drive. A write to a logical page involved writing first to one physical page, and then writing to the other physical page. In practice, few real systems implement stable storage with two disks because of the expense in writing to two disks for every update.

The stable storage device used (called SSB) is built from two banks of non-volatile, random access memory [1], each bank consisting possibly of several megabytes of memory (the board has battery backup power in case of power failure). Of course, accesses to the banks are mutually exclusive. Although the algorithms related to atomicity are essentially the same in spirit as Lampson and Sturgis's, our stable storage differs in three ways : (i) the processor writes to one bank, and the object manager internal to the board writes from the first bank to the second, thus freeing the processor from waiting for two write operations ; (ii) the SSB is part of the processor address space, in this case it is necessary to implement a way of controlling secure access to the board, (iii) the SSB provides atomic operations on group of objects (or data structures) which can be sparse in memory. These facilities are provided by the object manager which closely controls any access to an object. A complete description of SSB is given in [1].

2.2.2.Stable storage functionalities.

In order to manage objects (i.e. data structures), several primitive operations are offered by the Stable Storage, let us describe the most useful ones :

- **creat_ stb_obj ([out]stb_obj, [in]size).**

This primitive creates a stable object. The size of the new object is given as argument.

- **destroy _stb _obj ([in] stb _obj).**

The stable object located at address stb_obj is destroyed.

- **read_stb ([in] stb _obj, [out]add).**

The stable object located at address stb_obj is read and transferred into RAM storage at address add.

- **write_stb ([in] stb _obj, [in]add).**

The contents of the object located at address add in RAM storage are written at address stb_obj in stable storage.

- **grp_write_stb ([in]stb_obj_list, [in]x_list).**

This primitive allows the atomic update of a group of objects located in stable storage as explained in the following example.

Example.

Consider three integer stable objects O1, O2, O3 :

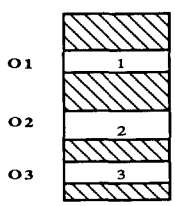

Figure 2 : Stable object group.

The execution of the operation grp_write_stb ((O1, O2, O3), (0,0,0)) produces one of the two following results depending whether it is successful or not.

On fig. 3.1, the operation has been successful and all three objects have been updated. On fig. 3.2, the operation has failed and has produced no effect.

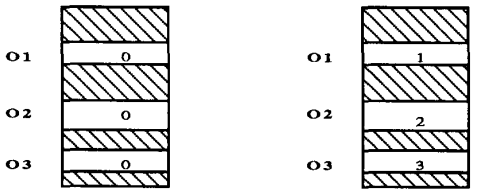

Fig. 3.1 Fig. 3.2

Figure 3: Stable object group update.

In our implementation, the primitive operation grp_write_stb is used for instance to atomically update queues. The object group is then composed of queue elements and pointers. It is also used to update checkpoints. In this case, the object group is constituted of all the objects which represent a process state.

2.2.3.Performance.

Unlike disk access time, accessing this new stable storage is on the order of accessing main memory, which is a big win. For instance, with a stable storage built with high performance disks (15 ms access time), an update operation takes 30 milliseconds for four bytes. The same operation takes only 9 microseconds in SSB. In fact, access time to SSB is only 2.7 times access time to a normal RAM but with a lot of advantages, in particular **write atomicity** and **protection**.

The performances achieved by such a stable storage are comparable with the performances of a normal RAM memory. This observation leads us to think that such a memory could be used to store efficiently such information as checkpoints. Measurements show that the time required for saving an object of more than hundred 32-bits words in SSB is approximatively 2.5 µs per word. So the time required for saving a checkpoint of two hundred bytes (50 words of 32 bits) is 125 µs. This is still quite reasonable compared to the average 0.2 ms necessary for an inter-processor communication via the global bus. Optimizations would even be possible, which would make it possible to reduce the amount of information saved in a checkpoint.

2.3. Fault hypothesis.

The machine is designed to survive two kinds of faults :

- hardware faults, such as the crash of a processor for internal (hardware misfunctioning) or external reasons (power failure...)

- software faults. These faults are detected at the hardware level through their consequences which, most of the time, happen to be an incorrect memory access.

Processors are assumed to be "fail-stop" [13], so they possess the following properties :

(i) After the detection of an error, the processor interrupts its activity (fail-fast property).

(ii) The crash of a processor can be detected by the other processors of the system.

(iii) Every processor can access two kinds of storage facilities : a volatile memory (RAM) and a stable storage.

(iv) The stable storage of a fail-stop processor can always be read by another fail-stop processor.

A crashed node is assumed to be repaired and restarted after a **finite delay**.

The communication medium is considered as unreliable. Actually message loss, duplication and desequencing are possible. However, the contents of a message cannot be altered during a node to node transfer. Finally, the communication medium can be unavailable during a **finite amount of time**.

3. Implementation of atomic rendezvous : general principles.

The implementation of atomic rendezvous relies on the concept of atomic transaction [5], [8]. Actually, using the transaction concept to implement the rendezvous is motivated by simplicity:

- a transaction is associated to every rendezvous,

- well-known commit protocols for atomic actions can readily be applied.

This section introduces the major components of the transactional system which has to be set up in order to solve the problem. We first give a brief description of the hierarchical structure of the system and then show how the rendezvous mechanism is mapped on this structure.

3.1. System structure.

Our system has to provide the transaction model to applications. It is organized in three layers organized as displayed on fig. 4 .

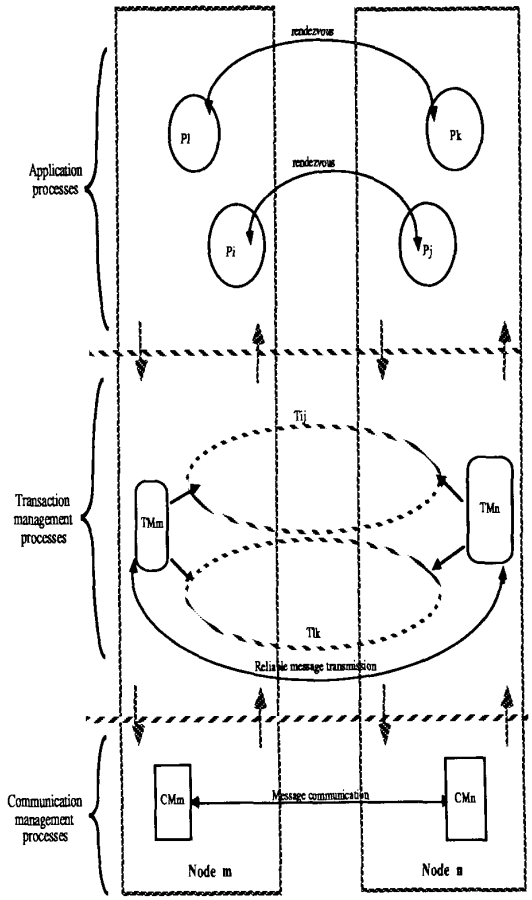

Figure 4 : System organization.

- the application layer where CSP processes communicate with rendezvous.

- the transaction layer which implements the notion of atomic transaction. Transaction management processes exchange informations via a reliable communication system. Every node Ni possesses its unique local transaction manager TMi which may communicate with managers TMj (j≠i) in order to implement atomic actions involving more than one process.

- the communication layer which implements a reliable communication system by using the stable storage facilities. Appropriate protocols are implemented in a cooperative way by processes (called CMi) located on system nodes. A complete description of this communication system can be found in [10].

3.2.Brief sketch of the solution.

Consider two application processes Pi (located on node Ni) and Pj (located on node Nj) (fig. 5).

```
Pi :: [ x:integer ;                    Pj :: [ y:integer ;
        ...                                    ...
    (Si )                                  (Sj )
      Pj ! x                                 Pi ? y
    (Si' )                                 (Sj' )
        ...                                    ...
    ]                                      ]
```

Figure 5 : Two CSP processes.

On this figure Si (resp. Sj) represents the state of Pi (resp. Pj) before execution of the rendezvous, and Si' (resp. Sj') represents the state of Pi (resp. Pj) after the execution of the rendezvous.

An atomic rendezvous possesses the "all or nothing" property :

(i) If a failure occurs during the execution of the rendezvous then Pi (resp. Pj) are backed up to Si (resp Sj). The rendezvous has not been completed.

(ii) If no failure occurs, then Pi (resp. Pj) reaches state Si' (resp. Sj') and the rendezvous has been achieved.

Let **Ctx-Pi** be the current state of process Pi, **Check_Pi** be the last checkpoint of process Pi stored in stable storage and Tij the transaction implementing a rendezvous between processes Pi and Pj. A queue **in_mess**, and a variable **result** are located in stable storage, and shared between application processes and their transaction manager. The implementation of atomic rendezvous between processes Pi and Pj can then be described as follows (fig. 6) :

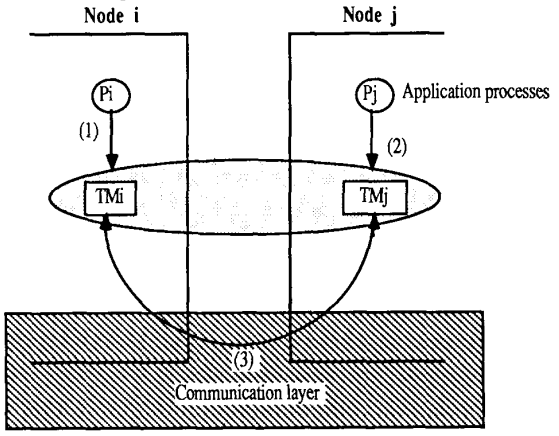

Figure 6 : A rendezvous execution.

Step (1), process Pi reaches the rendezvous operation Pj ! x.

It sends its rendezvous request to its node transaction manager TMi. In fact, Pi's request is stored in TMi's queue in_mess. The following piece of program describes the processing done on behalf of Pi.

```
(a) grp_write_stb(Check-Pi<-Ctx-Pi,in_mess[TMi]
<- [(sender_UID,receiver_UID),info],result<-undefined);
(b) while #result=undefined#
      do
         wait #termination of the atomic transaction associated to
              the rendezvous#
      od;
(c) if #result=failure#
    then
      failure #exception handling has to be performed#
    fi.
```

Instruction (a) updates atomically a group of objects in stable storage. This group is made up of **Check_Pi** (initialized with Ctx_Pi), the queue **in_mess** (initialized with a rendezvous request) and the **result** variable which is undefined while the transaction associated to the rendezvous is not completed.

During the execution of the transaction implementing the rendezvous, processes Pi and Pj are waiting (instruction b).

After the completion of the transaction, the variable **result** is set up to failure or success. In case of failure, Pi and Pj are backed up to state Si and Sj and possibly, a failure exception is signaled to both processes. Actually such an exception mechanism is not present in CSP, it could well be introduced in order to face such exceptionnal situations.

Step (2), process Pj reaches the rendezvous operation Pi ? y. It sends its rendezvous request to the transaction manager TMj and the same processing as above is initiated.

Step (3), the two transaction managers TMi and TMj cooperate to manage the transaction associated to the rendezvous requested by Pi and Pj. They reliably communicate by using the communication layer.

4. Transaction management for rendezvous implementation.

This section details the management of transactions implementing atomic rendezvous. After a short presentation of transaction representation and of data structures used for communications between transaction managers, we describe the behaviour of transaction managers.

4.1. Transaction representation.

The main purpose of a transaction management process is the creation and control of transactions implementing rendezvous.

On a node, a transaction is represented by a descriptor. Several transactions may be active at a given time on the same node and all have to be handled by the same transaction manager. It is then necessary to keep the descriptors of these active transaction in a table. Each transaction is associated to a unique identifier, which is delivered by the **deliver_UID** operation. In fact, timestamping mechanisms [7] are used to logically date every event in the system (e.g. object, transaction or process creation). Every entry in this table is made of three fields :

- the name of a transaction tr_UID,
- the identity of processes involved in the transaction tr_UID,
- a reference to the descriptor represented the transaction tr_UID.

The table and a transaction descriptor are represented in figure 7 :

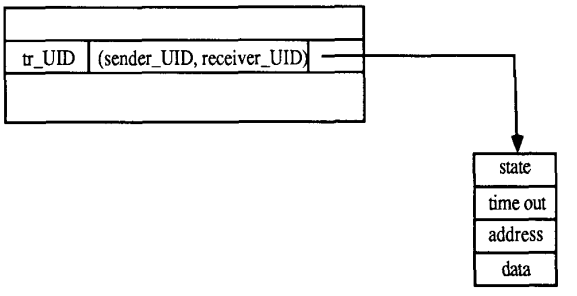

Figure 7 : Transaction representation.

where:

- **state** indicates the state of the transaction :

."undefined", if the transaction is currently being executed,

."ready_to_commit", if the transaction can be locally committed (the overall transaction encompassing two sites) will be committed if both sites are ready to commit,

."impossible to commit", if the transaction cannot be locally committed. The overall transaction encompassing two sites will be aborted,

. "committed", if the transaction is successful,

. "aborted", if the transaction is a failure.

- **time out** defines the maximal time duration granted to the transaction. This parameter is set up by the site which creates the transaction.

- **data** receives the data to be exchanged during the rendezvous.

- **address** contains the address (in the receiver address space) where the above data has to be stored as a consequence of the communication.

The transaction descriptor and the corresponding entry in the table are created and initialized through the invocation of an operation called **create_tr (tr_UID, <param>)**.

4.2.Messages handled by transaction managers.

Each local manager manages three queues (located in stable storage) in order to handle communications :

- an input communication queue, **in_mess**, for storing input messages sent by external transaction managers or stored by local application processes.

- an output communication queue, **out_mess**, for storing output messages directed towards external transaction managers.

- a queue, called **creation_request** used to store rendezvous requests not yet handled by a transaction.

A transaction manager deals with two types of messages : (i) messages from (or to) application processes and (ii) messages from (or to) other transaction managers.

The contents of these messages may be described as follows :

- **Messages of type (i)** contain the identity of the sender and that of the receiver plus the data to be transferred (in case of output) or the address to which the data has to be stored in case of input.

Sender	Receiver	data/address

- **Messages of type (ii)** are concerned with transaction creation and management. The structure of transaction creation messages is the following :

Sender_TM	Receiver_TM	tr_UID	Rdv_Processes

where sender_TM (resp. receiver_TM) is the identity of the sender transaction management (resp. receiver transaction manager). Rdv_process contains the identity of the two processes involved in the rendezvous. tr_UID contains a transaction unique identifier or **skip** when this field is undefined, i.e., the transaction corresponding to Rdv_process is not yet created. The structure of transaction management messages can be described as:

Sender_TM	Receiver_TM	tr_UID	Data

where data may be either the value exchanged during the rendezvous or control informations used to implement commit protocols. These control informations may be : ready_to_commit, not_ready_to_commit, commit, abort.

4.3.Description of a transaction manager.

This section describes the two major issues to be tackled by the transaction manager : transaction creation and transaction termination.

4.3.1.Transaction creation.

The main problem to be tackled concerns the unicity of the transaction associated to a rendezvous. Actually, due to the symmetry of the rendezvous mechanism, it is possible that two processes involved in a rendezvous make a request for a transaction creation at the same time. However only one transaction has to be created. For example, consider two application processes Pi and Pj which have to communicate by rendezvous. In that order requests are made to transaction managers TMi and TMj in order to create the transaction Tij. The creation of Tij amounts to attribute a unique identifier (UID) to this transaction and to initialize appropriate data structures. Transaction managers TMi and TMj have to cooperate in order to ensure the unicity of the UID. Several situations have to be considered depending on the ordering in time of the requests for transaction creation.

case 1:

Process Pi reaches the rendezvous after Pj (fig. 8). TMi is aware of Pj rendezvous request and it receives a rendezvous request from Pi.

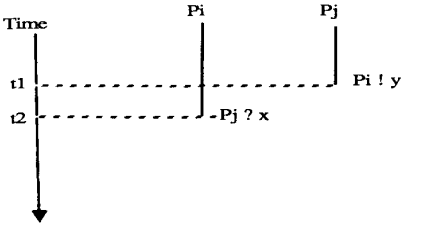

Figure 8 : Pj reaches the rendezvous before Pi.

Process Pi stores the message [(Pi, Pj), info] in in_mess[TMi]. Message [TMj, TMi, , (Pi, Pj)] is already present in creation_request[TMi] as shown in fig. 9.

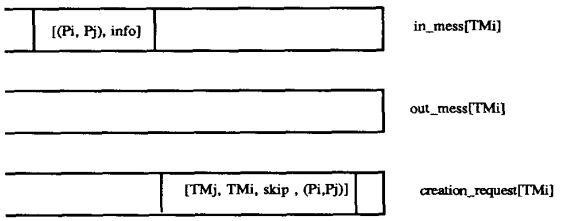

Figure 9 : State of TMi's queues at time t2 (before processingthe rendezvous request).

TMi is the only one to be aware of the rendezvous request of Pi and Pj. So it can execute the deliver_uid operation which gives it an unique identifier UID_Tij for the transaction Tij associated to the rendezvous. Then it locally creates the transaction Tij by invoking the operation create_tr (UID_Tij, in_mess.TMi[j].info). It informs TMj that Tij is created by sending TMj the message [TMi,TMj,UID_Tij,(Pi,Pj)]. TMi's queues in_mess and out_mess are atomically updated by using the **grp_write_stb** operation. Their contents after processing the rendezvous request is shown in figure 10.

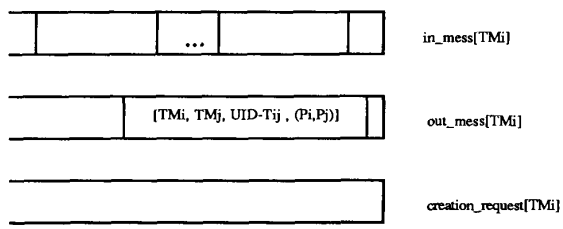

Figure 10 : State of TMi's queues at time t2 (after processing the rendezvous request).

case 2 :

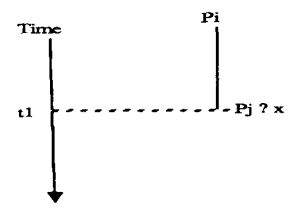

Figure 11 : Pi reaches its rendezvous operation.

Pi (on node i) requests a rendezvous with Pj (on node j). TMi has no information about an hypothetic rendezvous request from Pj (fig. 11).

Process Pi stores in TMi's queue in_mess the message [(PiPj), info] (fig. 12).

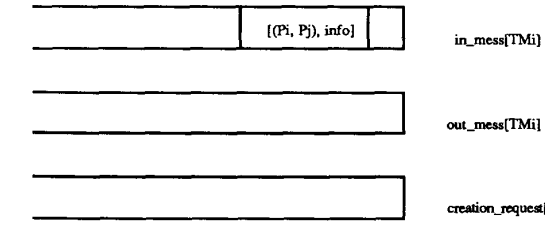

Figure 12 : State of TMi's queues at time t1 (before processing the rendezvous request).

TMi does not know about the rendezvous request from Pj. So it cannot create the transaction Tij. It informs TMj of Pi's rendezvous request by sending it the message [TMi,TMj, skip,(Pi,Pj)]. Pi's rendezvous request is stored in creation_request[TMi] (fig. 13).

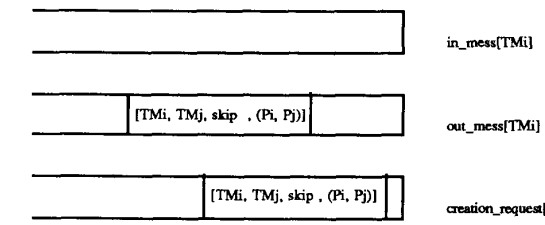

Figure 13 : State of TMi's queues at time t1 (after processing the rendezvous request).

case 3:

Processes Pi and Pj request the rendezvous at the same time.

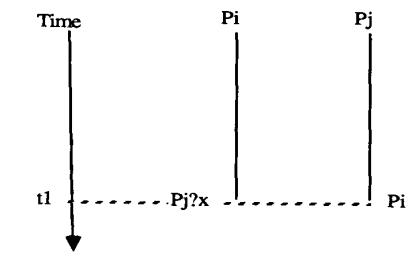

Figure 14 : Pi and Pj reach their respective rendezvous operation at the same time.

We distinguish two cases in our explanation : processing done by TMi (case 3.1) and processing done by TMj (case 3.2).

case 3.1 :

Process TMi receives the message [TMj,TMi,skip,(Pi,Pj)]. The message [TMi,TMj,skip,(Pi,Pj)] is already present in its queue creation_request (fig.15).

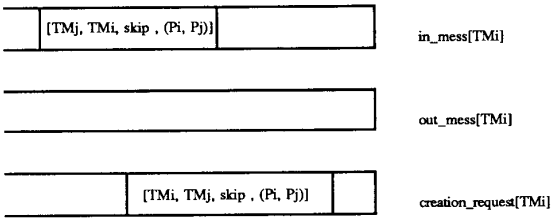

Figure 15 : State of TMi's queues at time t1 (before processing the rendezvous request).

Our problem is to decide if TMi has to provide UID_Tij or to wait that TMj sends it UID_Tij. To solve that problem, we decide that the transaction manager which holds the smallest UID provides UID_Tij. Let us assume that the property UID_TMi<UID_TMj holds. So, TMi is allowed to call the operation deliver_UID which returns UID_Tij associated to the rendezvous in order to locally create the transaction Tij by calling create_tr (UID_Tij,in_mess.TMi[j].info). It informs TMj that the transaction Tij is created by sending him the message [TMi,TMj,UID_Tij,(Pi,Pj)]. Its queues in_mess and out_mess are updated (fig. 16).

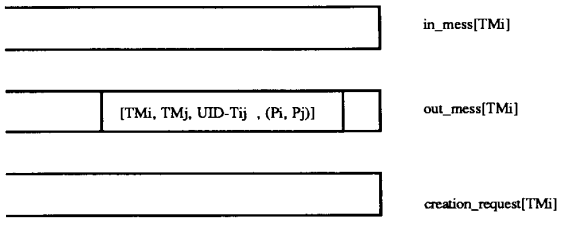

Figure 16 : State of TMi's queues at time t1 (after processing the rendezvous request).

case 3.2 :

We consider here the transaction management process TMj with UID_TMi<UID_TMj. The request [TMj,TMi,skip,(Pi,Pj)] is present in TMj's creation_request queue. TMj receives the message [TMi,TMj,UID_Tij,(Pi,Pj)] (fig. 17).

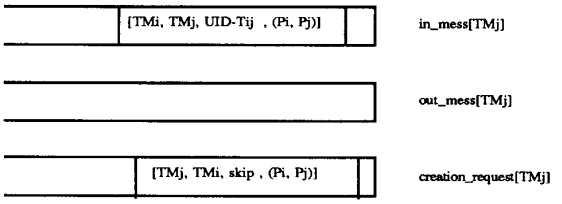

Figure 17 : State of TMj's queues at time t1 (before processing the rendezvous request).

It locally creates the transaction Tij by calling the operation create_tr (UID_Tij, in_mess.TMi[j].info). Its in_mess, out_mess and creation_request queues are updated (fig. 18). There is no information anymore about the current rendezvous between Pi and Pj in TMj's queues since TMj has locally created Tij and knows that TMi did the same.

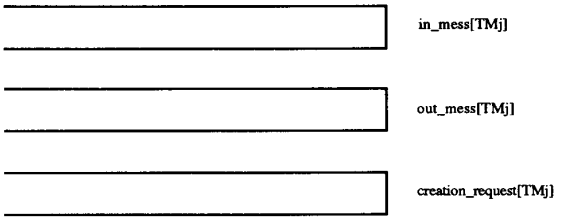

Figure 18 : State of TMi's queues at time t1 (after processing the rendezvous request).

4.3.2.Transaction management.

Two actions are performed in a transaction :

-the effective transmission of the data,

-the atomic termination of the transaction.

Our protocol is very classical, it is a two phases protocol similar to the one described in [LAMP-81]. Consider a transaction Tij between two processes TM_i and TM_j implementing a rendezvous between Pi and Pj. The master process of the transaction **Tij** (let us assume that it is process **TMi**) is the sender of the message **m** exchanged in the rendezvous. The process which receives m is the slave process of Tij (let us assume that it is process **TMj**). Let **desci** (resp. **descj**) be the transaction descriptor of Tij on the node where process Pi (resp. Pj) is located. The management of transaction Tij can be described with the following rules :

Rule 1

When TMi sends m to TMj, it sets a timeout and stores "**undefined**" in desci.state.

Rule 2

When a Tij's timeout is elapsed on TMi (resp. TMj) node then desci.state (descj.state) is changed to "**not_ready_to_commit**".

Rule 3

When TMj receives m, if descj.state has the state "undefined" then, TMj sends a **ready_to_commit** message to TMj and stores "**ready_to_commit**" in descj.state.

Rule 4

When TMj receives m, if descj.state is in "**not_ready_to_commit**" state then TMj sends a **not_ready_to_commit** message to TMi.

Rule 5

When a **ready_to_commit** message is received by TMi, if desci.state is equal to "**undefined**" the transaction Tij is locally committed else it is aborted. A **commit** message (in the former case) or an **abort** message (in the latter case) is sent to the remote process.

Rule 6

When a **commit** (respectively **abort**) message is received by TMj, the rendezvous is locally committed (respectively aborted).

We do not develop in more details protocols dealing with atomicity as they have already been presented elsewhere, [8], [9], [5], [12]...

5.Review and discussion.

This paper has presented an unusual application of the transactional model to the solution of synchronous communication scheme known as rendezvous. The major problem to be tackled was related to the atomicity property of the rendezvous : either the communication happens completely or not at all. The difficulty comes from the symmetric behaviour of the rendezvous which allows either the producer or the consumer to take the initiative of the communication. This difficulty does not arise in a RPC communication scheme where the caller is clearly the initiator of the communication.

We have chosen to solve the problem by applying the transactional model. A transaction is associated to every rendezvous and well-known commit protocols are used to implement atomic termination. Timestamping mechanisms are used to logically date every event in the system, thus permitting appropriate coordination for the creation of a transaction.

In our solution, a key role is played by stable storage devices which provide atomic access to group of objects. This notion of group of objects is absolutely central to our proposal as it allows the atomic creation and update of a checkpoint. In other words, the stable storage provides a hardware implementation of some basic transactions, one of which (used in this proposal) is the atomic update of a group of objects.

The important aspect to be emphasized concerns the efficient implementation of atomic rendezvous. For this purpose, it is clear that a stable storage device is absolutely necessary. However, as typical stable storages are built from disks, performances achieved are quite poor. This is the reason why very few proposals deal with the implementation of atomic operations in general and more specifically with atomic rendezvous. These proposals are generally concerned with data base management systems where the size of data to be processed is important (files) and can be stored on a double-disc stable storage (e.g. [5]). In ARGUS [9], an effort has been made in order to allow the programmer to define its own atomic actions. However, the implementation of atomic actions has not been optimized at all as stable storage is "emulated" on a single disk. Our approach which uses a high speed RAM-based stable storage make it realistic to implement efficiently such communications schemes as atomic remote procedure call or rendezvous.

REFERENCES

[1] J.-P. Banâtre, M. Banâtre, G. Muller.
Ensuring Data Security and Integrity with a fast stable storage,
Proc. 4 th conf. on Data Engineering, Los Angeles, February 1988, pp. 285-293.

[2] A. J. Bernstein.
Output guards and Nondeterminism in "Communicating Sequential Processes",
ACM Trans. Program. Lang. Syst., vol. 2, n° 2, pp. 234-238, April 1980.

[3] A. D. Birrell and B. J. Nelson.
Implementing Remote Procedure Calls,
ACM Trans. on Computer Systems, vol. 2, n° 1, pp. 39-59,
February 1984.

[4] G. N. Buckley and A. Silberschatz.
An Effective Implementation for the Generalized Input-output Construct of CSP,
ACM Trans. Program. Lang. Syst., vol. 5, n° 2, pp. 223-235,
April 1983.

[5] J.N. Gray.
Notes on data base operating systems,
In lecture Notes in Computer Science, Springer Verlag, New York, pp. 393-481, 1978.

[6] C. A. R. Hoare.
Communicating Sequential Processes,
Commun. ACM, vol. 21, n°8, pp. 666-677, August 1978.

[7] L. Lamport.
Time, clocks, and the ordering of events in a distributed system,
Commun. ACM, vol.21, n° 7, pp. 558-565, July 1978.

[8] B. Lampson and H. Sturgis.
Atomic transactions
Lecture Notes in Computer Science, vol. 105, New York : Springer-Verlag, 1981, pp. 246-265.

[9] B. Liskov.
Distributed programming in ARGUS,
Commun. ACM, vol. 31, n° 3, pp.300-312, March 1988.

[10] C. Morin.
Propositions pour la mise en œuvre des multifonctions dans GOTHIC.
Internal report. IRISA. November 1988.

[11] B. J. Nelson.
Remote Procedure Call,
Tech. Rep. CSL-81-9, Xerox PARC, Calif. 1981.

[12] David P. Reed
Implementing Atomic Actions on Decentralized Data.
ACM TOCS 1,1. (Feb. 1983), pp.3-23.

[13] Schneider F.B.
Fail-stop Processors
in Digest of Papers from Spring Compon'83. March,
San-Francisco 1983.