Modelling remote concurrency with Ada.
Case study of symmetric non-deterministic rendezvous.

Claude Kaiser, Christophe Pajault, Jean-François Pradat-Peyre

CEDRIC - CNAM Paris
292, rue St Martin, F-75003 Paris
{kaiser, christophe.pajault, peyre}[at]cnam[dot]fr
http://quasar.cnam.fr/

Abstract. When developing concurrent software, a proper engineering practice is to choose a good level of abstraction for expressing concurrency control. Ideally, this level should provide platform-independent abstractions but, as the platform concurrency behaviour cannot be ignored, this abstraction level must also be able to cope with it and exhibit the influence of different possible behaviours. We state that the Ada language provides such a convenient abstraction level for concurrency description and evaluation, including distributed concurrency. For demonstrating it, we present two new cooperative algorithms based on remote procedure calls which, although simply stated, contain actual concurrency complexity and difficulties. They allow a distributed symmetric non-deterministic rendezvous. One relies on a common server and the second is fully distributed. Both realize a symmetric rendezvous using an asymmetric RPC modelled by Ada rendezvous. These case studies show that Ada concurrency features provide the adequate abstraction level both for describing and evaluating concurrency and for carrying out design decisions.

1 Introduction

1.1 The need of high-level concurrency description

Concurrency is a prolific source of complexity and is a serious cause of errors when developing software. Thus it is a challenge for developers of long-lived, high-quality software that needs reliable software technologies.

Current approaches to software development use patterns or models as a set of guidelines for structuring application specification, design and implementation. Providing significant examples is of prime importance for mastering the additional temporal dimensions of correctness introduced by concurrency, i.e., safety and liveness. Even if you never employ them directly, reading about different special-purpose design patterns can give you ideas about how to attack real problems.

Moreover when developing concurrent software, a proper engineering practice is to choose a good level of abstraction for expressing concurrency control. Ideally, this level should provide platform-independent abstractions [1]. However the concurrency semantics of platforms associated with POSIX standards or with languages like Ada, Java or C# are different and this diversity may influence the correctness of some models or patterns. In [7], we have shown examples where the weak liveness semantics of Java and C# run time causes deadlock in some programs, which nevertheless have been proven safe with the Ada strong liveness semantics. As the platform concurrency behaviour cannot be ignored, the abstraction level should be able to cope with it and to analyse the influence of the different possible behaviours.

1.2 Ada as a concurrency description and modelling language

We state that the Ada language provides such a convenient abstraction level and thus may be used for concurrency description and evaluation, including distributed concurrency. Our statement is based on four assumptions.

First Ada proposes today the most powerful set of high level concurrency features available in an imperative language and its concurrency semantics is well and precisely defined. For expressing cooperation through a shared memory, protected objects can be used together with the requeue statement and with the entry family facility. For analysing communication without a shared memory, the rendezvous together with the use of the requeue statement allows to simulate simply the semantics of a remote procedure call.

Second the behavioural semantics of shared memory platforms used for other languages such as POSIX, C# or Java can be emulated with Ada [7]. Similarly, the remote procedure call, which is used in message passing protocols, can be simulated by Ada task rendezvous.

Third, as Ada concurrency semantics is precisely defined, model programs expressed in Ada can be analysed automatically for detecting correctness deficiencies such as deadlock or starvation. Our tool QUASAR [6], based on slicing, followed by Petri net generation and by model checking of the generated net, is devoted to concurrent Ada programs analysis. It allows evaluating and validating a concurrency model description at the design and specification stage.

Fourth, Ada provides an executable description language. This allows running simulations and testing the concurrency behaviour of programs.

Indeed, our approach that we teach also to our students aims at mixing design and evaluation. We are convinced that this encourages choosing simpler concurrency architectures in order to render them more readable, understandable, and finally easier to validate, maintain, reuse and modify.

This is the most necessary, as our students are not lucid enough about concurrent programming; as many designers they underestimate its difficulties and the need of a language for coping with clear concurrency ideas and structures. Todays programming approaches, whatever the pedantic name they use, are often close to cut-and-paste techniques and lack concurrency analysis.

We have already shown how data sharing paradigms which use the monitor concept [13] can be expressed in Ada and validated while running on platforms with different fairness semantics. We will now deal with remote procedure call.

Cooperative algorithms based on message passing tend to grow complicated especially when they include some form of consensus among participants. It is our statement that Ada is really suitable for expressing them when they rely on remote procedure calls. Thus our presentation focuses on such distributed concurrency. Although simply stated, our case study contains actual concurrency complexity and difficulties. We show that Ada can be used for analyzing them and carrying out design decisions.

2 Representing remote procedure call protocols by Ada tasking

In the following case study, we focus on the use of the remote procedure call concept for expressing, analysing and validating a protocol resolving the symmetrical rendezvous required by processes scattered in a distributed system. It may be useful for installing a peer-to-peer communication and it is an instance of the more general problem of group making in asynchronous distributed systems. The partners do not share a common memory; they communicate only by messages and use remote procedure call.

The case study is modelled in Ada as concurrent tasks that communicate only by rendezvous, without shared variables.

Recall that a concurrency protocol or a distributed application that is modelled in Ada is compiled and analysed as a single program. The first purpose of the model is to express and analyse its concurrency properties. It does not necessarily need to be a distributed program itself. If it were necessary however (for example, for running some simulation programs), the analysed model could be distributed and run on several platforms. Since Ada 95, distributed applications may be programmed with Ada partitions, according to postpartitioning and to the distributed annex choices. Active partitions have no global clock and communicate by asynchronous transfer of messages. Thus tasks are not visible across partitions: Ada has no remote rendezvous between tasks of different partitions, no distributed delay or time management and no distributed task management. This must be coped with and is well mastered by the partition model and the post-partitioning process (also called post-compilation partitioning)[17, 11]. This is an additional advantage of choosing Ada, an executable description language.

2.1 The basic binary rendezvous

The binary rendezvous has been suggested first for CSP [13] and Ada 83 [14]. In a binary rendezvous a communication involves the synchronization of exactly two processes. CSP provides a symmetric, nondeterministic and synchronous communication construct. Synchronous communication requires that both processes involved in a communication be ready to communicate before the communication can proceed. Nondeterministic selection allows a process to participate in one of many possible communication and symmetric communication allows both send and receive commands in a nondeterministic selection construct. Surveys of centralized and distributed CSP binary rendezvous implementations can be found in [3, 19].

2.2 Ada former implementations of a symmetric rendezvous

The Ada rendezvous between tasks is said to be asymmetric since the nondeterministic selection is possible only for receive commands. Moreover during the rendezvous, data may pass in both directions. This leads to an extended rendezvous or a remote invocation construct abstracting a remote procedure call from another task.

A programming challenge is how to implement a symmetric rendezvous using the Ada asymmetric rendezvous. A synchronous communication where a controller task performs an anonymous rendezvous between one producing and one consuming tasks has been given in [20]; this was also named three ways synchronization in the early book on concurrent programming in Ada [4]. This gave us insights for our server solution. We have not yet found any published implementation using the asymmetric Ada rendezvous for non deterministic pairwise choice in distributed systems (not even in the early review of Ada tasking [5]), possibly because it was cumbersome to do it in Ada 83 without the possibility of fixing a caller state when the requeue statement did not exist, and because Ada 83 did not aim at programming distributed applications. However symmetric intertask communication has been proposed as an additional feature of Ada 83 and was not held [10]. We shall show how it can be programmed with Ada 95.

2.3 Specification of a non-deterministic symmetric binary interaction

In distributed applications the binary remote rendezvous is often named binary interaction. For example, peer to peer collaboration starts by a binary interaction which can be performed in a purely decentralized manner directly between network hosts or in an indirect scheme using supernodes as rendezvous servers [2].

Let us now specify the case study that we consider in this paper. The distributed system is made of a set of at least two asynchronous processes that are labelled by distinct Ids. Sometimes a process that considers performing some peer-to-peer communication becomes a candidate partner, and seeks to constitute a pair with another candidate partner. Candidate partners behave all similarly, i.e. their rendezvous is symmetrical (they candidate in the same way, and all have the same capabilities for sending or receiving partners requests), the pair is the result of the non-deterministic interaction between two (or more) candidate partners and its Id values are returned to both successfully chosen candidate partners. We suppose that the partnership ends after a while allowing both processes of the pair to return to the state of possible candidate partners. The absence of candidate partners will not last forever.

To start with, we suppose that processes do not fail. Afterwards, we examine briefly the consequences of some process or communication failures, assuming nevertheless that procedure calls are atomic operations. Recall that distributed applications have to face site crash or message failures as well as absence of correspondants and that solving agreement problems in purely asynchronous distributed systems prone to process failures has been shown to be impossible deterministically [8, 9]. Thus we shall only examine how to increase the probability of non-faulty behaviour.

3 A server for anonymous non-deterministic pairing

We describe now a first solution relying on a centralized server. According to the use of Ada as a description and evaluation language, we first complete the protocol specification using remote procedure call (the RPC acronym will be used now on); then we model it in Ada which allows validating its safety by Quasar and evaluating its performances by simulation runs. We end the protocol analysis by some fault tolerance insights.

3.1 Specification

Each candidate partner calls the server by RPC, communicating its Id and waiting until the pair is notified. The pair notification contains the caller Id. An additional result is returned to paired partners, which is the choice of a leader arbitrary chosen by the server in the pair. The server specification follows:

1. The server is callable by any candidate at any moment, whatever the server state may be.
2. All the calls are registered and a caller is not acknowledged before it is paired.
3. The server waits until it has received two requests, before giving notifications.
4. Notifications of the pair values are sent as soon as possible, i.e. as soon as the server has two not yet acknowledged waiting calls.
5. Both notifications are done before starting preparing another pair.

Multithreading the service could help when notification transmission delays are long. However if the arriving calls are dispatched among the threads, several candidates may wait although they should be paired according to 4. This harmful situation is avoided if solely a unique thread does the pairing service. In case of lengthy transmission delays, the server may require auxiliary tasks controlled by a producer-consumers schema to perform concurrently these notifications. Similarly, other auxiliary tasks may intervene in a producers-consumer schema if registering the calls is lengthy.

Both RPC calls and the server sequence are indivisible actions (when failures will be considered, they should then be atomic).

3.2 Description and modelisation in Ada

A concurrent solution using shared data controlled by a monitor has been modelled with Ada protected objects and implemented also in Java and POSIX [15] and we have shown how to care of weak fairness semantics of Java and POSIX. We present here a solution where the symmetric rendezvous is controlled by an Ada task modelling a remote server called by RPC.

The RPC is modelled by a call to the server task, which exports a unique visible entry. According to Ada, this call blocks the caller until the results have been delivered.

The server has to hold on a first accepted entry call and to wait for a second one and, as soon as its second one is accepted, it has to return out parameters values to both accepted callers. Embedding two accept statements of the same entry is forbidden in Ada. However it is feasible when two embedded accept statements concern an entry and a private entry to which a former call has been requeued. The indivisibility of the server sequence is a property of the accept blocks.

The pairing action is realised as follows. The first accepted calling partner is requeued to a private entry (i.e., an entry not callable by another task). This removes its call from the visible queue and allows accepting another call on this entry. The server then accepts another call to the unique visible entry and the first statement of this accept block is to accept the call that was previously requeued to the private entry. Accept statements are nested and this nesting performs the symmetrical rendezvous as an indivisible action. Once this nesting done, the server exchanges candidate partner parameters and both calls are returned, allowing hereafter a new couple of calling partners to use the server.

```
Nb_Process : constant := N; type Id is range 1.. Nb_Process;

task Server is
  entry Cooperate(X: in Id; X_Other: out Id; Group_Leader: out Id);
private
  entry Waiting(X1: in Id; X_Other1: out Id; Group_Leader1: out Id);
end Server;

task body Server is
begin
 loop -- cyclic server
   accept Cooperate(X: in Id; X_Other: out Id; Group_Leader: out Id) do
      Group_Leader := X; -- server chooses arbitrarily the first member as leader
      requeue Waiting;    -- done for being able to nest two calls of the same entry
   end Cooperate;
   accept Cooperate(X: in Id; X_Other: out Id; Group_Leader: out Id) do
    accept Waiting(X1: in Id; X_Other1: out Id; Group_Leader1: out Id)
     do
      X_Other := X1; X_Other1 := X; Group_Leader := X1;
    end Waiting;
   end Cooperate;
 end loop;
end Server;
```

When implemented in other languages, their RPC semantics and the indivisibilities required has to be compared to Ada solutions in order to behave similarly when concurrency is involved. This concerns especially preventing perturbations caused by other calling candidates and respecting the notification completion as soon as possible and before starting up another pair. For example calling partners should not emit concurrent RPC calls, RPC servers should not process concurrently several rendezvous calls.

This Ada implementation has been validated as deadlock-free by our tool Quasar [6] and running simulation programs is straightforward.

We have not yet found a previously published version of this simple Ada solution. This solution can be extended to group formation, which for example might be useful before starting some grid-computing algorithm.

3.3 Failure considerations

We give some hints just to show that in presence of faults the concurrency problems discussion may go ahead still using Ada as a concurrency design, description and analysis language. Failure considerations lead using atomic procedure calls [16]and atomic actions, which have been devised for Ada [18, 21]. As ending the pairing action requires sending a notification to a pair of processes, this has also to cope with consensus on commit [12].

However according to the impossibility results recalled in section 2.3., there is a nonzero probability that the partners of an announced pair are not exactly two. Suppose that processes A and B have called the server, that A received correctly the notification and that B did not and exits from the RPC (the commit failed). B does not know A and will not answer to it. A is orphan especially if it was chosen as Group_Leader. But as B has not found a partner, it may start a new seek ending with C. B is now paired with both A and C. If the pairing data arrive to C and not to B (suppose B is in a jammy part of the network), B may try again with another partner, say D.

4 A cooperative non-deterministic symmetric rendezvous

We describe now a fully distributed solution relying on process cooperation. Here also we use Ada as a description and evaluation language. First we refine slightly the partner behaviour specification and examine some simplifying choices; they lead to two policies and two versions in each policy; then we model the more reliable solution in Ada and this allows validating its safety by Quasar and testing its performances by simulation runs. We end the protocol analysis by some fault tolerance insights.

4.1 Specification refinement

In this approach, each candidate partner tries to find directly another candidate partner willing also to constitute a pair. In the absence of failure, once a pair is formed, both partners of the pair share the same cooperation knowledge. Each one has registered the decision, i.e. the paired partners names. But each partner is also confident that its partner shares this information. If the pair is (A, B), partner A knows that its partner is B and that B knows that its partner is A. Symmetrically B knows that its partner is A and that A knows that its partner is B. We shall return to this when we examine the effects of failures.

We suppose that a partner can reach two states only in which it can seek a rendezvous. Either it sends a call to another partner which it supposes willing to answer to it (i.e. expected to be in the listening state or on the way to it), or it is listening, awaiting a remote call from any calling partner. The success supposes that while seeking for a pair the candidate partners finally achieve being in different states, one sending, the other one listening. If all partners wait forever in the same state (all listening or all calling), a communication deadlock occurs.

Let us recall that we assume that there is no failure (reliable communication and reliable processes). We shall consider successively two kinds of behaviour for processes that are not candidate partners. At first they do respond to any request and answer whether they are candidate or not. In a second version, they may be non-responding and remain silent, ignoring the request of a candidate partner.

4.2 Local concurrency level

First let us examine whether simultaneous communications or multithreading may help.

Simultaneous communications imply a global decision. Suppose that a candidate partner is allowed to manage concurrently its two communicating states or to seek several partners in parallel when it is calling (for example, broadcasting its request). Thus if it receives successively multiple proposals, it may concurrently start a rendezvous with several other candidate partners which themselves may already have started other rendezvous. Due to these possible transitivity and symmetry, the decision must be global and supposes some complex global serialization.

Local decision concerning two candidates at most. As the final choice concerns only two candidate partners, a global decision can be avoided. Introducing a local serialization of actions and a fixed dissymetry between partners, the choice can rely on a local decision taken by one partner only. This leads to a simpler solution which is presented now. First a candidate partner manages each of its communicating states exclusively and then it is either calling or listening. Thus it examines only one other candidate at a time. Second suppose that a listening partner (candidate or not) is able to execute as an indivisible operation the acceptance of one and only one pairing request followed by the processing of its answer. This listening partner is then able to commit the final decision for both partners: it can accept or refuse to constitute a pair with the candidate calling partner since it knows whether it is itself also willing to pair or not. This is possible if the successive pairing request calls are acknowledged serially and if the calling partner is blocked until the end of the RPC. By chance, this is the semantics of the synchronous remote procedure call and of the Ada rendezvous accept statement. This dissymmetry gives precedence to the listening partner over the calling partner for decision taking.

4.3 Required indivisible actions

For each process X, we introduce the following local variables:

```
Candidate      : Boolean; -- X is requiring a partner
Paired         : Boolean; -- X has got a partner
Partner        : Id;      -- X has got a partner which Id is the value
Next_Neighbor  : Id;      -- Function delivering a process Id, different at each call
Site(X)        : T_Site;  -- Network address of X
```

Let us summarize the concurrency assumptions for candidate processes:

1. Each candidate is either requesting or listening.
2. A requesting candidate sends only one request at a time (there is no calling concurrency).
3. A listening candidate picks and serves its received requests one at a time.
4. The pairing implies that both partners are candidates and that one is requesting and the other is listening.
5. The pairing decision is taken by the listening partner.

Each process X may be remotely called by RPC. This is modelled in Ada by the following entry which is visible by other processes:

```
entry Cooperate (Calling_Partner: in Id; Listening_Partner: out Id;
                 Accepted       : out Boolean);
```

According to the previous assumptions, a cooperating process may run either of the following action sequences, CS1 during the requesting state, or CS2 during the listening state. This can be specified in Ada as follows.

```
(CS1) {Candidate, not Paired }
Requesting a possible candidate partner
  Z := Next_Neighbour;          -- trying Z as candidate partner; Next_Neighbour
                                -- delivers a different Id at each call
  Site(Z).Cooperate(X, Z, OK);  -- calling Z with possibly a time limit
  -- if OK is returned within time limit and is True, then rendezvous(Z, X) has been decided
  -- by Z while X was waiting for the end of this call, therefore X is no longer candidate
  Candidate := not OK;          -- following statements are executed only when the
                                -- call is accepted within the time limit
  Paired := OK;
  if OK then Partner := Z; end if;
  {Requesting success = not Candidate }
```

```
(CS2) {True}
Listening and accepting a remote call
  accept Cooperate(Calling_Partner : in Id; Listening_Partner : out Id;
                   Accepted         : out Boolean) do
   -- request from remote candidate Calling_Partner, accepted with possibly a time limit
   -- returns name X and Accepted to caller Calling_Partner,
   -- if Candidate, the rendezvous(Y, X) is decided by X while Y is waiting for this decision
   -- X no longer Candidate once the rendezvous is decided
   Accepted:= Candidate;
   -- if X is Candidate, it decides to form the pair and returns Accepted = True
   Paired := Accepted;
   if Paired then Partner:= Calling_Partner; Listening_Partner := X;
   end if;
   Candidate := False; -- either X is no longer candidate or was already not candidate
  end Cooperate;
```

4.4 Navigation policies when seeking another candidate

We consider now two policies that may be used by a process for managing both calling and listening states and for navigating in the system when seeking another candidate: polling alternatively these two states or reacting when any one of these two states is triggered.

For simplicity, we suppose that each process is granted an assistant task that is in charge of the seeking policy.

Polling policy When polling, a process assistant loops alternatively listening for a call from any candidate partner and calling a process while changing the called process Id at each cycle. In the first version, the assistant task is supposed to acknowledge remote calls even if its process is not candidate for partnership and in that case to return a negative answer to the request. If all candidates happen to be in the same state, this leads to deadlock; this deadlock probability can be lessened, but not annulled, by using a probabilistic succession of states. Another version for avoiding deadlock is to wait in each state only during a given delay (large enough to allow message transmission - in a distributed system where transmission delays have a known upper limit, this delay can be chosen as twice this limit). As this allows also caring about processes non-responding since they are not candidate, the assistant task needs not necessarily to acknowledge every call when its process is not candidate. However this latter version may lead to livelock, even if livelock probability may be lessened similarly as above.

The kernel of the assistant task body is then the following.

```
-- function Hazard return Boolean; generates a value with some probability distribution
loop                        --polling loop of the assistant task
 case Hazard is
  when True  => CS1; -- possibly requesting during an exponential delay
  when False => CS2; -- possibly listening during an exponential delay
 end case;
 -- possible exit when not Candidate
 if not Candidate then return partner Id to the process; end if;
end loop
```

Reacting policy In this solution, when both states are simultaneously triggered, one state only has to be chosen. This supposes a nondeterministic symmetric selection of send or receive commands, as it is possible with CSP. We have devised such a scheme in Ada. Each process assistant task loops using a select statement, which nondeterministically accepts either a remote call from other candidates (the assistant is then listening) or a local call which aim is to emit a call to another process (the local process requires its assistant to perform such a call). The local call is performed repeatedly by the candidate partner to its assistant until a partner is found by the assistant, either by calling or by listening, and the assistant task must address each time a different process in order to hit a candidating one.

In the first version, the assistant task is supposed to accept remote calls even when the process is not candidate for partnership and in this latter case to return a negative answer to the request. In this version, it may happen that all candidate assistant tasks have been triggered by local calls for calling a remote partner and that this leads to circular situations such as candidate A requesting candidate B, candidate B requesting candidate C, candidate C requesting candidate A. This can be avoided by ordering processes and forbidding a candidate to call a process having a lower rank (a smaller Id for example). Thus an assistant task calls only processes (in fact it calls the other processes assistant tasks) with higher ranks that its own process. The highest rank process assistant does not emit a request and requeues the local call to a private entry Waiting, which allows it to wait until a successful remote call has to be returned to the process (see below).

In the second version, the assistant task needs no longer be present when its process is not candidate. Thus the assistant task of a candidate partner waits in each state only during a given delay using Ada selective accept with a delay alternative or Ada timed entry call.

In both versions, the assistant task must acknowledge the call of its local process and indicate whether seeking failed or succeeded and in the latter case returning the partner Id. Thus the first operation to do when examining a local call is to examine whether a distant call was successfully accepted since the last local call. Such a success must also forbid new distant call acceptance before its acknowledgement (the corresponding entry guard is set to False) and withdraws calling a new candidate partner.

4.5 The Ada symmetric and non-deterministic cooperative rendezvous protocol

We introduce some additional local entities:

```
Peer_To_Register : Boolean ;      -- assistant task has acknowledged a distant call
function Next_Neighbour(X) : Id ;-- delivers the Id of a Process with a rank greater
                                  -- than X; each call still delivers a different value.
function Top(X) return Boolean;  -- indicates whether process X has the highest rank
entry Waiting;                    -- for requeueing the highest rank process
```

The kernel of the assistant task body is then the following, giving priority to distant call when both local and distant call are triggered.

```
loop forever -- assistant task cyclic behaviour
 select
  accept Local_Call do
   if Peer_To_Register then
    Candidate := False; Peer_To_Register := False;
    -- acknowledges beforehand the local caller when a remote call was
    -- successfully accepted since last local call
   else
    Candidate := True; -- local call is considered as a candidature for pairing
    if not Top(X) then
     CS1; -- for requesting Neighbour(X) when candidate and not Last(X)
    else
     requeue Waiting; -- just wait for acknowledging a remote call
    end if;
   end if;
  end Local_Call;
 or
```

```
  when Peer_To_Register =>
   accept Waiting do
     Candidate := False; Peer_To_Register := False;
     -- acknowledges local caller when a remote call was accepted since last local call
   end Waiting;
  or
   when not Peer_To_Register =>
     accept Distant_Call do -- accepting a remote call
       CS2;
       Peer_To_Register := Accepted;
       -- a partner has been committed; local process has to be informed
     end Distant_Call;
 end select;
end loop;
```

This description holds for Ada concurrency semantics of the task rendezvous. When the protocol is implemented using other languages or platforms, their RPC semantics and the required indivisibilities must be confronted with Ada choices.

The complexity of this cooperative protocol, the possible concurrency simplifications and the resulting algorithm are easy to express and to analyse using ADA. This shows the expressive power of Ada for concurrency problems

Two policies, polling and reactive, and two versions in each policy have been devised. The full Ada solution of the first version of the reactive policy is given in Annex and its implementation has been analysed as deadlock-free by our tool Quasar [6].

The second version with delayed call or accept is suitable for an asynchronous network with bounded transmission delays, but it is prone to communication uncertainty in a purely asynchronous distributed system (in this latter, when a called process does not answer, the caller doesnt know whether the called process is non responding since it is not a candidate or the candidate called process has sent an answer which is very late to arrive, so any delay may be erroneous since there is no bounded transmission delay).

The polling versions are never absolutely safe (they are prone to deadlock or to livelock) and have some probability of failure.

4.6 Failure considerations

Again we give just some hints to show that the concurrency errors due to the presence of faults may still be considered using Ada as a description and analysis language. Faulty processes or variable network delays may be simulated with abort and timed rendezvous and delayed entry.

Anew, failure considerations lead using atomic procedure calls and atomic actions in order to reduce the faults outcome. As all communications are point to point, the consensus on commit is not necessary this time. However when some messages are not received either by failure or when the waiting delays are too short for network propagation or for processor overload, a candidate partner may still be associated with any number of other candidates in a dissymetrical association. This again may be simulated in Ada when requesting candidates use timed entry calls and when listening candidates use delayed accepts. Recall that this may also lead to livelocks.

5 Conclusion

In the introduction, we have stated that Ada can be used for concurrency description and evaluation. We pointed out that its concurrency features are powerful and have been settled at a convenient abstraction level, and that this allows expressing most of the useful concurrent algorithms as well as emulating other language constructs or semantics.

In a former paper we have shown how the use of Ada protected objects allowed to describe and validate a monitor based implementation and to derive it for Java and POSIX safe implementations.

In this paper we have designed step by step and analysed RPC based concurrent and distributed protocols. The first one, a server protocol, is so simple that it is directly apprehensible in Ada. The second, a cooperative protocol, was described by parts and all, including the global architecture which is the most delicate part, were able to be expressed in Ada. With these protocols, we have also fulfilled twice the programming challenge of implementing a symmetric rendezvous using the Ada asymmetric rendezvous. These case studies emphasise the suitability of Ada as a domain-specific language for distributed concurrency description and evaluation.

These protocols, either based on protected objects or tasking rendezvous, are programmed by our students as a starting step of the chameneos game [15] which they have to implement, validate and simulate.

Our final claim is that Ada concurrency programming richness is largely underestimated and its capabilities not yet fully understood, especially by designers of new languages. With this presentation we would also like to contribute pointing out its power and its elegance.

References

1. AADL workshop, 2005. www.axlog.fr/R_d/aadl/workshop2005_fr.html.
2. S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.
3. R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Trans. Softw. Eng.*, 15(9):1053–1065, 1989.
4. A. Burns. *Concurrent programming in Ada.* Cambridge University Press, New York, NY, USA, 1985.
5. A. Burns, A. M. Lister, and A. J. Wellings. *A review of Ada tasking.* Springer-Verlag New York, Inc., New York, NY, USA, 1987.
6. S. Evangelista, C. Kaiser, J-F. Pradat-Peyre, and P. Rousseau. Quasar: a new tool for analyzing concurrent programs. In *Int. Conf. on Reliable Software Technologies, Ada-Europe'03*, volume 2655, pages 166–181, Toulouse, FR, 2003. Springer-Verlag.
7. S. Evangelista, C. Kaiser, J-F. Pradat-Peyre, and P. Rousseau. Comparing Java, C# and Ada monitors queuing policies: a case study and its Ada refinement. *Ada Lett.*, XXVI(2):23–37, 2006.
8. F. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distrib. Comput.*, 16(2-3):121–163, 2003.

9. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
10. N. Francez and S. A. Yemini. Symmetric intertask communication. *ACM Trans. Program. Lang. Syst.*, 7(4):622–636, 1985.
11. F. Gasperoni. Programming distributed systems, 2003.
12. J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.
13. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
14. J. D. Ichbiah. Preliminary Ada reference manual. *SIGPLAN Not.*, 14(6a):1–145, 1979.
15. C. Kaiser and J.F. Pradat-Peyre. Chameneos, a concurrency game for Java, Ada and others. In *ACS/IEEE Int. Conf. AICCSA'03*. IEEE CS Press, 2003.
16. K-J. Lin and J. D. Gannon. Atomic remote procedure call. *IEEE Trans. Softw. Eng.*, 11(10):1126–1135, 1985.
17. L. Pautet and S. Tardieu. GLADE User Guide, 2000.
18. A. B. Romanovsky, S. E. Mitchell, and A. J. Wellings. On programming atomic actions in Ada 95. In *Ada-Europe '97: Proceedings of the 1997 Ada-Europe Int. Conf. on Reliable Software Technologies*, pages 254–265. Springer-Verlag, 1997.
19. F. B. Schneider. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst.*, 4(2):125–148, 1982.
20. D. Le Verrand. *Le langage Ada. Manuel d'évaluation.* Dunod, 1982.
21. A. Wellings and A. Burns. Implementing atomic actions in Ada 95. *IEEE Trans. Softw. Eng.*, 23(2):107–123, 1997.

Annex: the cooperative non-deterministic symmetric rendezvous program.

This runnable Ada program contains the cooperative protocol in the first version of the reactive policy together with a simulation where Nb_Process processes require each a non-deterministic rendezvous Nb_Trial times. This simulation may not terminate when all processes but one have ended after their successful Nb_Trial rendezvous. The remaining lonely process cannot find a pairing partner. This runnable program can be downloaded at:
http://quasar.cnam.fr/files/concurrency_papers.html.

```
with Text_IO; use Text_IO;

Procedure Cooperative is
  Nb_Process : constant := 9; type Id is range 1..Nb_Process;
  Nb_Trial   : constant := 6; -- number of rendezvous requested by each process
      --- set of Assistants --
  task type T_Assistant is
   entry Get_Id       (Y: in Id);
   entry Local_Call   (X: in Id; X_Other: out Id; Group_Leader: out Id;
                       Accepted : out Boolean); -- local call
   entry Distant_Call(X: in Id; X_Other: out Id; Group_Leader: out Id;
                       Accepted : out Boolean); -- remote call
  private
   entry Waiting      (X: in Id; X_Other: out Id; Group_Leader: out Id;
                       Accepted : out Boolean); -- for Id'last Process
  end T_Assistant;
  Assistant: array(Id) of T_Assistant;
```

```
    --- Assistant task body ---
task body T_Assistant is
  Ego               : Id;                       -- Caller_Id
  Partner           : Id;                       -- the result of search
  Peer_To_Register  : Boolean := False;  -- partner found by CS2 through an
                                          -- accepted Distant_call
  Candidate         : Boolean := False;  -- searching a partner
    --- Process neighbourhood management ---
  Current : Id := Id'Last; -- used for managing Next_Neighbour
     -- Next_Neighbour provides a neighbour name which is always larger than
     -- the caller's name (this avoids deadlock due to circular calls)
  function Next_Neighbour return Id is
  begin
    if Current = Id'Last then Current := Ego + 1;
    else Current := Current + 1; end if;
    return Current;
  end Next_Neighbour;
  function Top(X : in Id) return Boolean is -- X has the highest rank
  begin return X = Id'Last; end Top;
  Y : Id;  -- records a value returned by Next_Neighbour
    --- end of neighbourhood management ---

begin
  -- attaching each Assistant to a different Process
  accept Get_Id(Y: in Id) do Ego := Y; end Get_Id;

  -- cyclic Assistant Ego waiting for a request from a remote process or from a local call
  loop
   select
    -- CS1 : FIRST MUTUALLY EXCLUSIVE ACTION : local call to propagate to Next_Neigbour
    accept Local_Call(X: in Id; X_Other: out Id; Group_Leader:
                              out Id; Accepted : out Boolean) do
     -- a new partner may have been already found and has to be registered
     if Peer_To_Register then
      X_Other := Partner ; Group_Leader := Ego;
      Accepted:= True; -- a partner has been found and registered;
                        -- reset Assistant state
      Peer_To_Register := False;
     else
      Candidate := True; -- a local request is made for searching a partner
        -- calls a neighbour process, hoping it might be a partner assume: every
        -- assistant that is called will answer positively or negatively to remote call
      if not Top(Ego) then
        -- calls a neighbour holding a name strictly bigger than Ego
        Y := Next_Neighbour; -- each time a different neighbour process
        Assistant(Y).Distant_Call(X,X_Other,Group_Leader,Accepted);
        if Accepted then
         Candidate := False; -- a partner is found, reset Assistant state
        end if;
      else
       requeue Waiting; -- X holds the biggest name, it never calls a neighbour
      end if;
     end if;
    end Local_Call;

   or

    -- used only by Assistant(Id'Last) for returning the partner name
    when Peer_To_Register =>
     accept Waiting(X: in Id; X_Other: out Id; Group_Leader: out Id;
      Accepted : out Boolean) do
      -- a new partner has called, was accepted and then has to be registered
      X_Other := Partner ; Group_Leader := Ego;
      Accepted:= True; -- a partner has been registered; reset Assistant state
      Peer_To_Register := False;
     end Waiting;

   or
```

```
        -- CS2 : SECOND MUTUALLY EXCLUSIVE ACTION : waiting for a distant call
        -- barrier forbids accepting a new distant call before acknowledging the previous one
        when not Peer_To_Register =>
         accept Distant_Call(X: in Id; X_Other: out Id; Group_Leader: out
             Id; Accepted : out Boolean) do
           Partner := X; X_Other := Ego; Group_Leader := Ego;
           Accepted := Candidate; -- the pair is accepted only if process Ego is seeking
           Peer_To_Register := Candidate; -- for triggering its local registration
           Candidate := False; -- whatever the answer, process Ego is no longer seeking
        end Distant_Call;
      or
        terminate;
      end select;
    end loop;           -- end of cyclic Assistant code
  end T_Assistant;      -- end of Assistant body

        --- set of processes ---
  task type T_Process is
   entry Get_Id(Y : in Id);
   entry Start_Peering(Pilot : in Id; Copilot : in Id; Leader : in Id);
   entry Finish_Peering(Copilot: in Id; Pilot : in Id; Leader : in Id);
  end T_Process;
  Process: array(Id) of T_Process;

        --- Process task body ---
  task body T_Process is
      Ego, Partner, Leader: Id;
      Done                : Boolean := False;

begin
   -- giving each Process a different name
   accept Get_Id(Y: in Id) do Ego := Y; end Get_Id;

   -- each process loops seeking a partner just for recording its name and the rendezvous
   -- peer-to-peer asymmetric communication is simulated only
   for I in 1.. Nb_Trial  loop
    -- get a partner
      loop
        Assistant(Ego).Local_Call(Ego, Partner, Leader, Done);
        -- repeats request until a partner is found
        delay(0.001);
        exit when Done;
      end loop;
      Put_line("Process" & Id'Image(Ego) & " is paired with " &
                Id'Image(Partner) & ". Leader is " & Id'Image(Leader));
      if Ego = Leader then
       -- sends initializing RPC
        Process(Partner).Start_Peering(Ego, Partner, Leader);
        accept Finish_Peering(Copilot : in Id; Pilot : in Id;
                              Leader : in Id); -- waits until peering ends
      else
        accept Start_Peering(Pilot : in Id; Copilot : in Id;
                             Leader : in Id);  -- waits partner call
        delay(1.0); -- simulates peer interchange and processes corresponding activity
        Put_line("PROCESS" & Id'Image(Ego) & " acts as COPILOT while " &
                 "PEERING with" & Id'Image(Partner) & " as PILOT");
        -- peer exchange ends; sends releasing RPC
        Process(Partner).Finish_Peering(Ego, Partner, Leader);
      end if;
    end loop;
  end T_Process;

        --- allocating names to tasks ---
begin
   for I in Id loop Assistant(I).Get_Id(I) ; end loop;
   for I in Id loop Process(I).Get_Id(I) ; end loop;
end Cooperative;
```