On "alien call" protocol of remote procedure calls and on connecting virtual machines into one virtual supercomputer

Bolesław Ciesielski, Grażyna Mirkowska, Andrzej Salwicki, Oskar Świda

***Abstract*—We present an original protocol of remote procedure calls. It deals with active objects. Its name "alien call" is to underline the fact that methods of one active object can be called from other (alien) objects. It is a protocol since the execution of the callee's method is done in cooperation with the thread of caller. Each active object may be a client (caller) and a server (callee) too. The remote procedure calls may act as rendez-vous of two active object or as interruption of the callee's thread. Methods can change their character dynamically from private to public to private...**

The second part of the message tells that by establishing connections among virtual machines one can quickly and efficiently construct a (virtual) distributed computer. As a consequence our model applies to concurrent programming as well as to distributed programming. Moreover, many mixed models ofprogramming are easy to realize. Suppose that we have accounts on three computers. One can allocate four active objects on computer A, three active objects on computer B and 7 active objects on computer C. All of them cooperate not knowing where its partners are alocated. Modifications of the configuration can be done in couple of minutes.

These concepts were validated in Loglan'82 object oriented programming language. They have however a general and universal character.

***Index Terms*—concurrent execution, distributed execution, object programming, remote procedure call, parallel virtual machine,**

I. Introduction

We present an original protocol of cooperation among active objects. It distinguishes from the other approaches: monitors, rendez-vous, message passing. All these mechanisms can be easily defined by the proposed protocol.

The ideas we present are of general and universal character. They may be adapted in various environments. The methodology of programming active objects ... was validated by an implementation in Loglan"82 programming language. Consequently the examples will be given in this language. Those who prefer the jargon of C++, Java, C# etc. may wish to consult the web page where the same examples are written in pseudo-Java code.

II. Terminology

Before going into details we need to fix the notions.

active object – an object with a list of instructions to be executed, see thread,(note, a regular object has nothing to do)

thread – it is sequence of instructions associated with an object, in Java, Thread is the name of a class such that any object instanceof Thread has a seqence of instructions to be executed,

specification of method – it is what in other places people call a heading of the method i.e. a list of parameters and their types and the type of result,

III. Assumptions

We assume that programs and systems of programs are written in one object-oriented programming language $\mathcal{L}$. Next, we assume that the language $\mathcal{L}$ admits one predefined class **process** (The name is of no importance. call it **thread** if you wish so) The objects of this class and of classes derived from it will be called *active objects*. Each active object o has the following properties:

- object o has a thread i.e. a list of instructions to be executed,
- object o is either in *passive* state or in *active* state,
- the object may enter an active state (In this state the instructions of the thread are executed concurrently with the instructions of other threads. The main program is also a thread.)
- the active object may enter a passive state. In a passive state only ...(e.g. when the command suspend() is executed),
- each method of the object is either *enabled* or *disabled*. Initially all the methods of any active object are disabled.
- instruction enable ⟨list of methods⟩ causes that all the methods from the list become enabled. (One may believe the enabled methods are public.)
- instruction disable ⟨list of methods⟩ causes that all the methods from the list become disabled. (Analogously, one may believe the disabled methods are private). The status of a method may change from disabled to enabled to enabled ...
- instruction accept ...
- alien call instruction –

The effect of synchronized alien call

$$\langle \underbrace{\textbf{call}\ X.m(arg_1,...,arg_n)}_{\text{in caller } Y} \parallel \underbrace{\textbf{accept}\ m}_{\text{in callee } X} \rangle.$$

One should conceive as a one instruction executed jointly by both processes X and Y.

IV. Scenario of active object

Creation of an active object:

Elaboration of the object expression `new`

`MyProcess(parameters)` returns an object o of `type Myprocess`. Hence, the execution of the assignment instruction

```
z := new MyProcess(parameters)
```

leads to a new configuration where, the set of existing objects is augmented by the object o, object o is the value of the variable `z`. One may say also, the object o is pointed out (is referenced to) by the variable `z`. Remark, the object may be allocated on one computer and the variable `z` may be on another computer. An active object o after it has been created, remains in the state PASSIVE. Another active object, owner of variable `z`, such that the value of `z` is the object o may activate the . Object o of name z becomes ACTIVE when another object executes the command `resume(z)`. An active object may execute command `suspend()` and enter the state PASSIVE.
TERMINATION: an active object may reach the end of its thread, for example it may reach **`end`** `Myprocess`. In this case the active object is killed and deallocated. For the object cannot be activated again and its resources being private will never be accessible from outside.
AWAITING – an active object may enter the state AWAITING if it awaits for a partner object to jointly execute a procedure instruction. It happens if either the current object begins execution of alien call of a procedure or if the object begins execution of the instruction accept (see below).

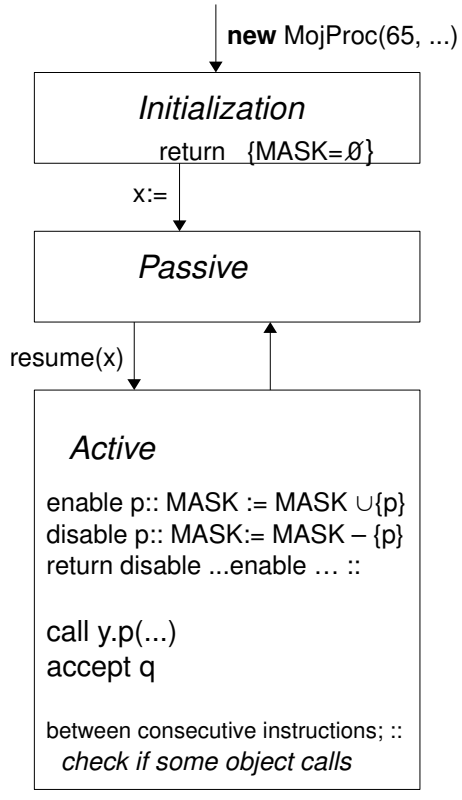

Fig. 1. The scenario of active object

V. ALIEN CALL PROTOCOL

We shall illustrate the protocol by a series of pictures.

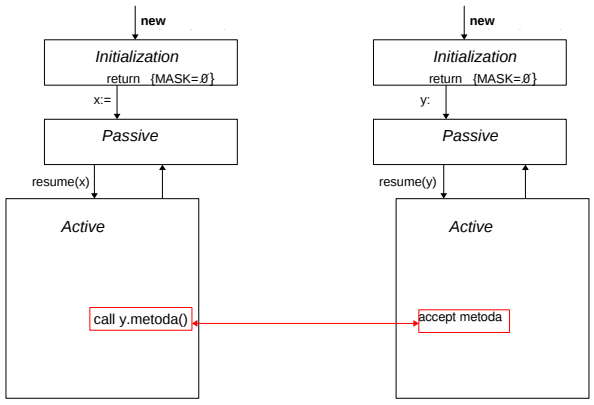

Fig. 2. Protocol of alien call part1

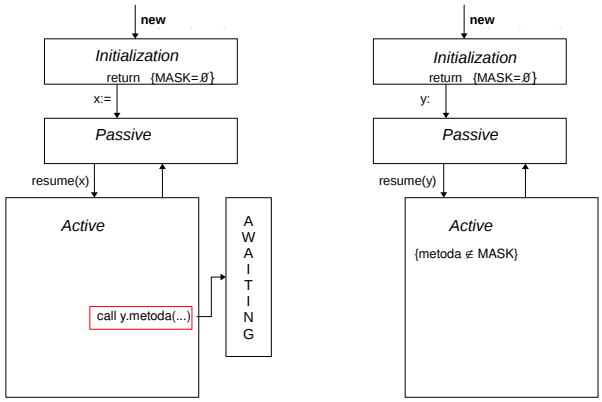

Fig. 3. Protocol of alien call part 2

VI. PROPERTIES

In this section we attempt to describe the properties of active objects from the point of view of a user.

- P1 An active object is created and memorized when an assignment instruction `z:=new Myprocess()` is executed. Note, creation of an active object without assignment has no sense, for the newly created object will become a garbage immediately.
- P2 The newly created object will be allocated on a computer indicated by the value of the first parameter. The value 0 tells that new active object will be allocated and run on the same computer (concurrency).
- P3 **Mutual exclusion**. If several active objects simultaneously execute alien procedure calls of one active object o (a callee), then only one at the time may execute it. The following algorithmic formula expresses the mutual exclusion of n parallel alien calls. The formula abstracts from the possible other threads.

$$\Box \parallel_{i=1}^{n} [o.m_{i_j}; R_i]\alpha \Leftrightarrow$$
$$\bigvee_{k=1}^{n} \Box \overline{o.m_{k_j}}; [\parallel_{i=1, i\neq k}^{n} [o.m_{i_j}; R_i] \parallel R_k]\alpha$$

The expression $\overline{o.m_{k_j}}$ denotes the body of the method

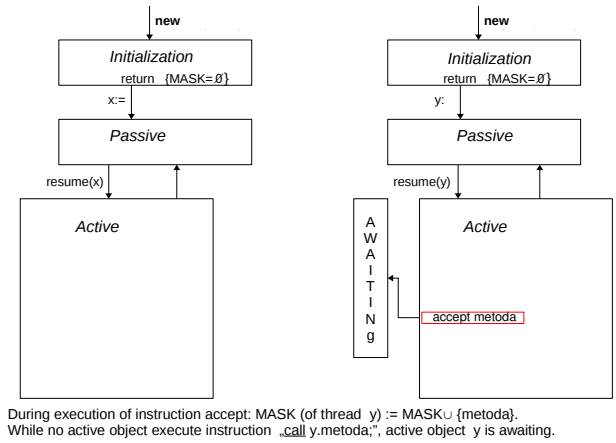

Fig. 4. Protocol of alien call part 3

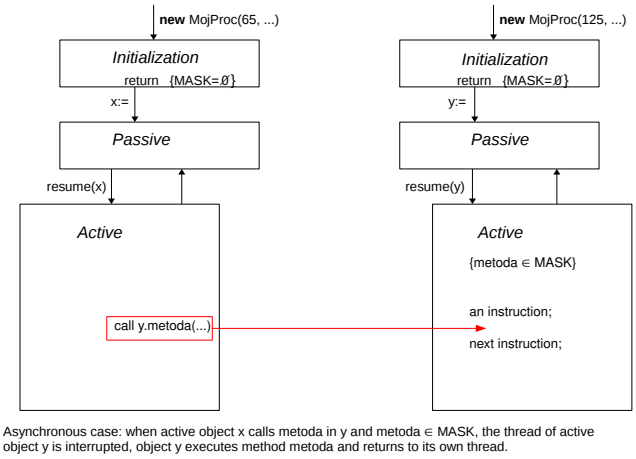

Fig. 5. pProtocol of alien call part 4

m in object o, modified by the actual parameters.

P4 **Dynamic public/private methods**. Each method of an active object may be public in one moment and private in another one. A method m is public when its name is in the MASK of the object, when its name does not belong to the MASK, the method is private.

P5 When one active object calls a method m of another active object and the method m is in the MASK then we have an effect of *interruption*. The callee interrupts its own work and executes a service for the caller object.

P6 When one active object calls a method m of another active object and the method m is not in the MASK and the callee executes the instruction accept with the name m on the list, then we have an effect of *meeting*. The caller object and the callee object meet and execute the called method jointly.

P7 Each active object may be once a client calling a procedure in a remote active object and in another moment it can be a server, ready to serve one of its procedures to other active objects.

P8 **Distributed concurrency is true fair concurrency**.

The programs like [p:=false ‖ while p do x := x+1] always terminate.

VII. Examples

We begin with an example showing a couple of threads printing in the screen.

Listing 1. Three not synchronized processes print on screen

```
program first;
     unit writer: process(node: integer,
                                s: string);
          var i: integer, A: arrayof char;
     begin
       A := unpack(s);
       return;

       for i := lower(A) to upper(A)
       do
          write(A(i));
       od
     end writer;
   var w1, w2, w3: writer;
begin
    w1 := new writer(0, "aaaaaaaaaaaaaaaa");
    w2 := new writer(0, "bbbbbbbbbbbbbbbb");
    w3 := new writer(0, "cccccccccccccccc");
    resume(w1);  resume(w2);  resume(w3);
end
```

Perhaps you guessed the image on the screen shows a mixture of letters 'a', 'b' amd 'c'. This is so because the threads w1, w2 and w3 compete for the screen – their common resource. Our next example shows how to define a semaphore, the tool for synchronization of processes. A semaphore s is an active object with three methods: pass, free and fin. The methods have empty bodies. The thread of the semaphore s repeats instructions `accept pass,fin; accept free, fin;` until one of clients execute the command `call s.fin`. If all clients follow the same scheme `call s.pass; critical section; call s.free`, then no interleaving of commands of critical sections is posible.

Listing 2. Implementation of semaphores using the alien procedure call

```
program Second;

unit binarySemaphore: process(node:integer);
   unit pass: procedure;
   end pass;
   unit free: procedure;
   end free;
   unit fin: procedure;
   begin
      bol := false;
   end fin;
   var bol: boolean;
begin
  bol := true;

  return;
  enable   fin;
  while bol do
     accept pass;
     accept free
  od;
end  binarySemaphore;

 unit  writer:  process (node:integer,
```

```
    nr:integer, s: string, sem: aSemaphore);
   var i: integer,
       A: arrayof char;

   unit fin: procedure;
   begin
   end fin;
 begin
   A:=unpack(s);
   return;

   call sem.pass;
   for i := lower(a) to upper(a)
   do
      write(a(i));
   od;
   writeln;
   call sem.free;
   accept fin;
 end writer;

 var s: aSemaphore, w1, w2: writer,
     i: integer;

 begin
  s := new aSemaphore(0);
  resume(s);
  w1:= new writer(0,1,"aaaaaaaaaaaaa",s);
  w2:= new writer(0,2,"bbbbbbbbbbbbb", s);
  writeln("press Enter");
  readln;
  resume(w1);
  resume(w2);
  call w1.fin; call w2.fin;
  call s.fin;
 end Second
```

The following example is more interesting. We shall analyse it more closely.

Listing 3. A spooler

```
unit queue:class(type element; size:integer);
(* The auxiliary class implementing
   queues with a limited capacity.
   The class is parameterized by the element
   type and the maximum queue size *)

  unit insert:procedure(e:element); ...
   (* insert element into the queue *)
  unit delete:function:element; ...
   (* remove the first element *)
  unit isempty:function:boolean; ...
    (* check if the queue is empty *)
  unit isfull:function:boolean; ...
    (* check if the queue is full *)

end queue;
...
unit spooler:process;
 var
  Q:queue,          (* queue of files *)
  f:filename,

 unit print:procedure(f:filename);
 begin
   call Q.insert(f);
   if Q.isfull
   then
     return disable print
   fi;
 end print;

begin
  Q := new queue(filename, 50);
  return;
  do
    disable print;
    if Q.isempty
    then
      accept print
    fi;
    f := Q.delete;
    enable print;

    (* send the file f to the printer *)
    ...
  od
end spooler;
```

Two questions arise:

1) Suppose several active objects simultaneously require printing by executing
   call s.print(f) ‖ call s.print(f')
   commands in two objects p and q. Can we assure that no request will be lost or improperly queued?
2) Suppose that the spooler takes a file from the queue to be sent to a printer and simultaneously one or more processes require printing of their files. Can we assure that files will be printed without interleaving their contents and in proper order?

These questions find the following answers.

Lemma

No request will be lost and the requests will be handled as first-in first-out policy requires.

The positive answer to the first question is founded on the alien call protocol. For it will be impossible that two activation records of procedure print coexist. ROZWINAC?? As concerns the second question: it is sure that the operation Q.delete will not interfere with any operation Q.insert. It is so because we disabled the operation print before attempting to execute operations delete and isempty.

VIII. A THESIS ON ALIEN CALL

Thesis

Any known mechanism of synchronization and/or communication is expressible in terms of alien call with low cost.

We can not prove the thesis. Instead, like in the case of Church thesis, we can present numerous arguments witnessing the thesis. Our proof is by natural induction instead of mathematical one. Semaphores.

Critical regions, ‖ Monitors. ‖

IX. DISCUSSION

Semaphore, critical regions, monitors, ... all tools of concurrent programing are efficiently definable by alien call. Also all concepts known in distributed programming like space of tuples(LINDA), cooperation of sequential process in execution of assignment instructions (CSP), cooperation of two tasks in execution of a piece of code(ADA), message passing etc. are defined by alien call easily and efficiently. Comparison with

CORBA. CORBA[**?**] is a methodology of coupling object built in various programming languages and residing on different computers. CORBA has to deal with many problems: one, the internal presentation of objects may vary, second, objects should present its methods to any other object – and there are different formats of specification of the methods.

Java RMI: is similar to CORBA with one simplification. In Java RMI all objects are defined through some Java classes. All objects have the same, uniform format. The remaining assumption of CORBA are retained. Namely, the communication is to be established between objects of unknown classes. The RMI requires many preparatory stepslike: creation of stubs and skeletons of classes. We aknowledge that there is some progress: the stubs and skeletons may be produced by the javac compiler meking an additional compilation by rmic compiler not necessary. However the methodology of Java RMI remains heavier than necessary.

In our proposal we stress that a distributed application is one program with perhaps classes distributed over network. Hence the compiler may perform the static semantic analysis of a program without difficulties. We propose to avoid marshalling of object over network as much as possible. We accept the idea of transmitting a clone of an object to another active object, but the class of active objects of a distributed program

X. A CONJECTURE

We do not know whether the alien call protocol may be efficiently implemented in Java, C#, or a similar programming language. It seems that when an object o is runnable and executes instructions of its thread then no method of the object o can be called from another thread and executed. Any answer, whether a positive one or a negative one will be appreciated. The author of the answer will gain a prize of a box of wine or its equivalent. The precise formulation requires some space and consists of an interface and an example.

A. Specification

We ask whether it is possible to declare a class implementing the interface given below. Observe that the comment makes an integral part of the specification.

Listing 4. Specification of Active Objects

```
interface ActiveObjects
  extends Runnable {
    /* initially, a new ActiveObject is
       passive, the set of Enabled
       methods is empty. */
    /* methods changing status */
    void resume(SpecifiAO o) {}
    void stop() {}
   /* the instruction stop hangs the
      execution of the thread,
      the instruction resume(o) resumes
      execution of the thread o. */
    /* methods changing Mask */
    void enable(String m) {}
    void disable(String m) {}
    /* methods of collaboration */
    void accept(String m) {}
    void alienCall() {}
/* Let ActiveObjects be a class imple-
          menting this specification.
  Let P be a class extending
            the class ActiveObjects.
  Let o be an active object of
                        the class P.
  Consider the instructions of the thread o.

  A) the effect of enable: the methods m of
                  the thread become enabled.
     Enabled := Enabled + {m}
  B) the effect of disable
     Enabled := Enabled - {m}
  C) the effect of accept
     The instruction accept will be executed
     in cooperation of an instruction of
                alienCall, see the point E.
     It means that another active object of
     a class derived from ActiveObject must
     begin to execute
         alienCall(o, m, params)
  D) the effect of an alien call
     D1) the object o
must be in state Active,
         the method m must be Enabled.
  E) When a rendez-vous of two Active Objects
     is reached then
     e1) the parameters of alien call of the
       method m are transferred to the object o,
     e2) the called object o executes the me-
        thod m with with the actual parameters
        obtained from the callee object.
     e3)
  F) Asynchronous alien call
     When a caller process encounter an alien
     call instruction and the callee process is
     active and the method is enabled then the
     callee interrupts the execution of its
     thread and executes alien call instruction
     - see E)



 */
}  // end of interface
```

B. example

kjfkjsfjf Benton, Cardelli, xxx Modern concurrency abstractions for C#

Journal ACM Transactions on Programming Languages and Systems (TOPLAS) TOPLAS Homepage archive Volume 26 Issue 5, September 2004 ACM New York, NY, USA table of contents doi¿10.1145/1018203.1018205