Ada 95 — Remote Call Interface*

Dariusz Wawrzyniak
darek@cs.put.poznan.pl

22 maja 2001

1 Zdalne wywoływanie procedur

Procedury i funkcje zdalne w języku Ada 95 tworzy się poprzez zadeklarowanie ich w jednostce bibliotecznej kategorii `Remote_Call_Interface`. Procedury te udostępniane są i tym samym wykonywane przez partycję aktywną, w skład której wchodzi dana jednostka biblioteczna. Z perspektywy modelu klinet-serwer partycja taka jest serwerem procedur zdalnych, zadeklarowanych we wchodzących w jej skład jednostkach bibliotecznych.

Kategoryzacja jednostki bibliotecznej odbywa się poprzez umieszczenie odpowiedniej pragmy. W przypadku jednostek udostępniających procedury zdalne jest to
`pragma Remote_Call_Interface` (patrz listing 1).

Listing 1: Specyfikacja pakietu procedur zdalnych

```
package Server is
   pragma Remote_Call_Interface;

   procedure Service(arg: in out Positive);
end Server;
```

Z punktu widzenia klienta wywołanie procedury zdalnej niczym się nie różni od wywołania procedury z jakieś jednostki skonsolidowanej z programem lub procedurą klienta. Listing 2 pokazuje wywołanie procedury `Service` z pakietu `Server`. O sposobie wykonania tego wywołania decyduje kategoria pakietu `Server` (listing 1) oraz lokalizacja odopowiednich jednostek bibliotecznych w poszczególnych partycjach systemu rozproszonego. Opis przydziału jednostek bibliotecznych do partycji zaprezentowany jest w przykładowym pliku konfiguracyjnym dla narzędzia `gnatdist` (listing 3), które jest częścią środowiska Glade — implementacji aneksu E standardu Ada 95.

Listing 2: Przykład wywołanie procedury zdalnej z pakietu `Server`

```
with Server;
with Text_IO; use Text_IO;

Procedure main is
   value: Positive := 34;
begin
   Server.Service(value);
   Put_Line("Result obtained:" & Integer'Image(value));
end main;
```

*W przypadku wykrycia jakichkolwiek błędów proszę o mail na podany adres.

Listing 3: Przykład pliku konfiguracyjnego dla `gnatdist`

```
configuration Messenger is

   -- pragma Starter (None);
   -- uruchamianie reczne

   pragma Boot_Server ("tcp", "localhost:5556");

   Msg_Client: Partition;
   Msg_Server: Partition := (Server);
   Procedure main is in Msg_Client;
end Messenger;
```

2 Asynchroniczne wywołanie zdalne

Procedurę asynchroniczną towrzy się z użyciem pragmy `Asynchronous` (listing 4). Ponieważ klient wywołuje procedurę asynchroniczną, ale nie czeka na jej zakończenie, asynchorniczne procedury zdalne nie mogą mieć parametryów typu *out* i *in out*. Z tego samego powodu nie można również definiować funkcji asynchronicznych.

Listing 4: Specyfikacja asynchronicznej procedury zdalnej

```
package Server is
   pragma Remote_Call_Interface;

   procedure Service(arg: Positive);
   function Get_result return Positive;
   pragma Asynchronous(Service);
end Server;
```

Listing 5 pokazuje przykładowe rozwiązanie problemu przekazywania wyniku wykonania procedury asynchronicznej, jeśli taka konieczność istnieje. Wynik wykonania asynchronicznej procedury zdalnej umiszczany jest w zmiennej `result`, dostępnej wewnątrz pakietu `Server`. Wartość tej zmiennej może zostać odczytana przez zdalne wywołanie funkcji `Get_result`. Wywołanie funkcji `Get_result` jest oczywiście synchroniczne, asynchroniczna jest tylko procedura `Service`, jak wynika z linii 6 na listingu 4.

Listing 5: Przekazywanie wyników asynchronicznych procedur zdalnych

```
package body Server is
   result: Positive;

   procedure Service(arg: Positive) is
   begin
      Delay(5.0);
      result := arg + 10;
   end Service;

   function Get_result return Positive is
   begin
      return result;
   end;
end Server;
```

W celu uwypuklenia pewnych istotnych aspektów synchronizacji klienta i serwera założono, że wykonanie procedury `Service` zajmuje jakiś znaczący okres czasu. Czas ten w powyższym przykładzie jest symulowany przez instrukcję `Delay` (linia 6) i wynosi 5 sekund. Ponieważ klient uzyskuje sterowanie zaraz po wywołaniu procedury, może się zdarzyć, że wywoła funkcję `Get_result`, zanim wykonywanie procedury się zakończy i właściwy wynik jej wykonania będzie dostępny. Przykład takiej sytucji został przedstawiony na listingu 6. Procedura `Service` wywoływana jest dwa razy — z parametrem aktualnym 34 w linii 6 i z parametrem aktualnym 20 w linii 10. O wynik pierwszego wywołania klient upomina się po 7 sekundach (linia 8), wywołując funkcję `Get_result`, której wartość zwrotna jest parametrem atrybutu `Image` typu `Integer` i tym samym stanowi część łańcucha znaków wyświetlanego przez procedure `Put_Line`. Ponieważ wykonanie procedury `Service` zajmuje 5 sekund, należy domniemać, że po 7 skundach zwłoki po stronie klienta jej wynik będzie już dostępny. W przypadku drugiego wywołania klient upomina się o wynik natychmiast (linia 11), ale otrzymuje wynik wykonania poprzedniej procedury, ponieważ funkcja `Get_result` wykonywana jest po stronie serwera współbieżnie z procedurą `Service`. Dopiero kolejne wywołanie funkcji `Get_result` (linia 13) zwraca właściwy wynik.

Listing 6: Przykład wywołania asynchronicznej procedury zdalnej

```
  with Server;
  with Text_IO; use Text_IO;
3
  Procedure main is
  begin
6    Server.Service(34);
     Delay(7.0);
     Put_Line("Result obtained:" & Integer'Image(Server.Get_result));
9
     Server.Service(20);
     Put_Line("Result obtained:" & Integer'Image(Server.Get_result));
12   Delay(7.0);
     Put_Line("Result obtained:" & Integer'Image(Server.Get_result));
  end main;
```

3 Zwrotne wywołanie zdalne

Problem z odbiorem wyników wywołania asynchronicznej procedury zdalnej, jaki pojawia się w przykładzie w roz. 2, polega na zagwarantowaniu klientowi, że otrzymuje on wynik wykonania ostatnio wywołanej procedury zdalnej lub — jeśli wynik nie jest jeszcze dostępny — zablokowaniu procesu klienta do momentu uzyskania tego wyniku. Potrzebny byłby zatem dodatkowy mechanizm blokowania dostępu do zmiennej `result` w pakiecie `Server`. Jeśli odpowiednia blokada na dostęp do zmiennej `result` zakładana byłaby w procedurze `Service` zaraz po rozpoczęciu jej wykonywania, to nie ma gwarancji, że zostanie ona założona zanim klient odzyska sterowanie i wywoła funkcję `Get_result`. Blokadę taką musiałby zatem zakładać klient przez synchroniczne wywołanie jakieś dodatkowej procedury zdalnej, co jednak może stawiać pod znakiem zapytania sensowność wywołań asynchronicznych, które stają się dość skomplikowane. Poza tym powstaje dodatkowy problem — błędów w programach klientów lub ich wykonaniu. Jeśli jakiś proces klienta założy blokadę, a później jej nie zdejmie ze względu na błąd w programie lub niespodziewane zakończenie procesu, powstaje istotny problem *osieroconych obliczeń*.

Alternatywnym podejściem mogłoby być „próbkowanie" serwera, czyli sprawdzanie co jakiś czas dostępności wyniku. Rozwiązanie to wymaga jednak przygotowania mechanizmu identyfikacji obliczeń (np. zastosowanie tzw. *epok*) i ma poważną wadę w postaci *aktywnego czekania*,

które oznacza marnotrawienie czasu procesora.

Niezależnie od sposobu rozwiązania problemu synchronizacji, w opisanych podejściach serwer byłby zobowiązany do utrzymywania wyników obliczeń tak długo, aż nie zostaną odebrane przez klientów. W przypadku wywołań procedury przez różnych klientów mogłoby to rodzić istotny problem buforowania, co w konsekwencji — przy ograniczonym buforze — mogłoby wymagać tymczasowego zaprzestania wykonywania procedur zdalnych w celu ochrony przed przepełnieniem bufora. Oznaczałoby to, że jeden proces kliencki, nie odbierając wyników obliczeń, pośrednio paraliżowałby działania innych procesów, korzystających z tego samego serwera.

Właściwym rozwiązaniem byłoby podejście, w którym klient musi przygotować miejsca na wyniki i przekazać informację o tym serwerowi. Poszczególne partycje w systemie rozproszonym mogą działać w osobnych przestrzeniach adresowych, należy więc w ogólności założyć, że nie mają one dostępu do wspólnej przestrzeni adresowej. Wskazaniem zatem serwerowi miejsca na wyniki obliczeń nie oznacza jeszcze, że serwer będzie miał do niego dostęp. Klient musiałby zatem przygotować również odpowiedni mechanizm dostępu, który umożliwiłby serwerowi odpowiednie umieszczenie wyników. Jeżeli mechanizmem dostępu do usług serwera są procedury zdalne, to można spróbować wykorzystać ten mechanizm również w celu umożliwienia serwerowi przekazania wyników na stronę klienta.

Mechanizm, w którym serwer wywołuje procedury zdalne, wykonywane po stronie klienta, nazywany jest *wywołaniem zwrotnym*. W kontekście rozważanego problemu zwrotne wywołanie zdalne umożliwia serwerowi przekazania wyniku wykonania asynchronicznej procedury zdalnej przez wywołanie funkcji udostępnionej zdalnie przez klienta. Wskaźnik do procedury zwrotnej jest przekazywany przez klienta, jako jeden z argumentów wywołania zdalnej procedury serwera. W ten sposób serwer dowiaduje się nie tyle o miejscu na wyniki obliczeń, przygotowanym przez klienta, ile uzyskuje precyzyjną informację o destępnym mechanizmie (w tym przypadku zdalnej procedurze), który ma wykorzystać w wiadomy mu sposób bez wnikania w szczegóły implementacji.

W skład mechanizmu wywołania zwrotnego wchodzą dwa pakiety: pakiet klienta, którego specyfikację przedstawia listing 7 i pakiet serwera, którego specyfikację przedstawia listing 8.

Listing 7: Specyfikacja pakietu klienta

```
package Client is
   pragma Remote_Call_Interface;

   procedure Put_result(arg: Positive);
   procedure Remote_call_wrapper(arg: Positive);
end Client;
```

Listing 8: Specyfikacja pakietu serwera

```
package Server is
   pragma Remote_Call_Interface;

   type RR_reference is access procedure (arg: Positive);

   procedure Service(arg: in Positive; rrf: in RR_reference);
   pragma Asynchronous(Service);
end Server;
```

Przykładowa implementacja tych pakietów pokazana jest na listingu 9 i 10. W celu przekazania wyników obliczeń tuż przed zakończeniem procedury `Service` wywoływana jest procedura wskazywana przez parametr `rrf` (linia 5 na listingu 9). Jej wykonanie polega na wyświetleniu odpowiedniego komunikatu z wynikiem (linia 7 na listingu 10).

Listing 9: Implementacja pakietu serwera

```
package body Server is
   procedure Service(arg: in Positive; rrf: in RR_reference) is
   begin
      Delay(5.0);
      rrf.all(arg+10);
   end Service;
end Server;
```

Listing 10: Implementacja pakietu klienta

```
with Server;
with Text_IO; Use Text_IO;

package body Client is
   procedure Put_result(arg: Positive) is
   begin
      Put_Line("The result is "& Positive'Image(arg));
   end Put_result;

   procedure Remote_call_wrapper(arg: Positive) is
   begin
      Server.Service(arg, Put_result'Access);
   end Remote_call_wrapper;
end Client;
```

W głównym programie klienta (listing 11) przetwarzanie po wywołaniu procedury zdalnej symulowane jest przez pętlę wyświatlającą co sekundę odpowiedni komunikat (linie 8–11 i 16–19 na listing 11). W wyniku wykonania tego programu pomiędzy komunikatami wyświetlanymi przez procedurę `Main` nastąpi wyświetlenie komunikatu z wynikiem wykonania procedury zdalnej.

Listing 11: Przykładowy program klienta

```
with Server;
with Client;
with Text_IO; use Text_IO;

Procedure main is
begin
   Client.Remote_call_wrapper(34);
   for i in 1..10 loop
      Put_Line("Waiting for the result");
      Delay(1.0);
   end loop;
   Put_Line("Result obtained");

   Client.Remote_call_wrapper(20);
   Put_Line("Waiting for the result");
   for i in 1..10 loop
      Put_Line("Waiting for the result");
      Delay(1.0);
   end loop;
   Put_Line("Result obtained");
end main;
```

W procedurze `Main` w dalszym ciągu nie wiadomo jednak, czy w danej chwili uzyskano już wynik wykonania procedury zdalnej `Service`. W ramach wykonania procedury zwrotnej

`Put_result` można by ustawiać jakąś flagę, która świadczyłaby o uzyskaniu tego wyniku, jednak konieczność oczekiwania na jego uzyskanie wymagałaby ciągłego sprawdzania flagi, czyli aktywnego czekania. Właściwy mechanizm synchronizacji musiałby działać tak, żaby próba dostępu do wyniku zakończyła się zablkowaniem procesu do czasu jego uzysakania, jeśli nie został on już wcześniej przekazany.

Realizację takiego mechanizmu pokazano w implementacji pakietu `Client` na listingu 12. W realizacji wykorzysany został obiekt chroniony `Result_buffer`, do którego dostęp możliwy jest przez procedurę `put` lub wejście `get`. Podstawą synchronizacji jest zmienna `flag` obiektu chronionego. W zależności od wartości tej zmiennej wejście `get` jest otwarte lub zamknięte (linia 36).

Listing 12: Synchronizacja klienta z wykonaniem procedury zwrotnej

```
with Server;
with Text_IO; Use Text_IO;

package body Client is
   protected Result_buffer is
      procedure put(v: in Positive);
      entry get(v: out Positive);
      private
         flag: Boolean := False;
         value: Positive;
   end Result_buffer;

   procedure Put_result(arg: Positive) is
   begin
      Result_buffer.put(arg);
   end Put_result;

   procedure Remote_call_wrapper(arg: Positive) is
   begin
      Server.Service(arg, Put_result'Access);
   end Remote_call_wrapper;

   function Get_result return Positive is
      tmp: Positive;
   begin
      Result_buffer.get(tmp);
      return tmp;
   end;

   protected body Result_buffer is
      procedure put(v: in Positive) is
      begin
         value := v;
         flag := True;
      end;
      entry get(v: out Positive) when flag is
      begin
         v := value;
         flag := False;
      end;
   end Result_buffer;
end Client;
```