

Applications Concurrentes :

Conception,

Outils de Validation

ACCOV_B

Chapitre 6

OUVERTURES

**AUTRES ASPECTS DE LA
PROGRAMMATION DE LA CONCURRENCE**

Interruptions
Transfert de contrôle asynchrone par abort
Structuration, placement, construction
Distribution

INTERRUPTIONS

PRINCIPES RETENUS POUR ADA 95

- le modèle d'interruptions retenu pour Ada 95 vise à retenir les éléments communs à la plupart des architectures matérielles et aux modèles normalisés comme Posix 1003.1.
- l'efficacité du traitement de l'interruption est obtenue en permettant au mécanisme câblé d'interruption d'invoquer directement le traitement logiciel d'interruption.

RAPPEL SUR LA GESTION DES INTERRUPTIONS

Un niveau d'interruption peut être dans l'un des états

- **désarmé** : il n'accepte aucune demande d'interruption ("disable")
- **armé** : il accepte et mémorise une demande ("able")
On peut armer et désarmer un niveau par programme
- **masqué** : la demande est mémorisée mais non délivrée ("masked")
- **démasqué** : la demande est mémorisée et transmise
On peut masquer et démasquer un niveau par programme
- **attente** : le niveau a reçu une demande mais il y a plus prioritaire que lui ("pending")
- **élu** : la demande d'interruption a requisitionné le processeur pour exécuter un traite-interruption ("interrupt handler")

CHOIX DE ADA 95

Le traite-interruption est une procédure d'un objet protégé.

protected ALARME is

 procedure REPONSE; -- c'est le traite-interruption

 pragma ATTACH_HANDLER(REPONSE, IT_ALARME);

end ALARME;

protected body ALARME is

 procedure REPONSE is

 begin

 ... -- le code de la procédure traite-interruption

 end REPONSE;

end ALARME;

Ici IT_ALARME identifie le niveau d'interruption concerné.

La procédure de l'objet protégé est directement appelée par le mécanisme matériel d'interruption et partage des données avec des tâches ou d'autres traite-interruptions.

Selon la sémantique des objets protégés, dès qu'il a commencé son exécution, le traite-interruption ne peut plus être bloqué. Par ailleurs, si les données partagées sont accédées par une tâche appelant une procédure ou une fonction de l'objet protégé, l'interruption doit rester en attente pour respecter l'exclusion mutuelle d'accès à l'objet protégé.

TRAITE-INTERRUPTIONS

Un traite-interruption peut être défini statiquement (à la compilation) en utilisant un pragma (transmission d'information au compilateur) comme dans l'exemple ci-dessus. Il peut aussi être défini dynamiquement en utilisant le pragma `INTERRUPT_HANDLER` et la procédure `ATTACH_HANDLER`. L'exemple précédent devient :

```
protected ALARME is
  procedure REPONSE; -- c'est le traite-interruption
  pragma INTERRUPT_HANDLER(REPONSE);
end ALARME;
```

```
protected body ALARME is
  procedure REPONSE is
    begin
      ... -- le code de la procédure traite-interruption
    end REPONSE;
end ALARME;
```

```
ATTACH_HANDLER(ALARME.REPONSE, IT_ALARME);
```

On note l'utilisation du même terme `ATTACH_HANDLER`

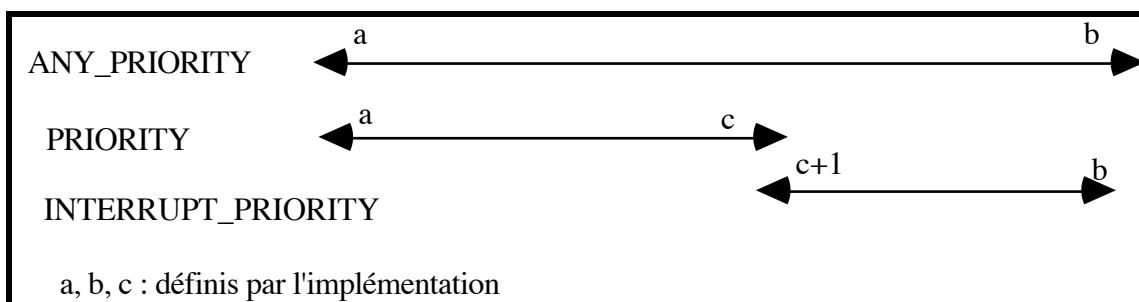
Les pragmas `ATTACH_HANDLER` et `INTERRUPT_HANDLER` ont pour rôle de prévenir le compilateur que l'on a un traite-interruption et qu'il doit générer du code pour une procédure qui peut être appelée par le mécanisme câblé d'interruption :

- il faut stocker l'interrupt-id associé car on peut avoir besoin de masquer l'interruption
- le retour d'interruption ne se code pas comme un retour de procédure
- la pile à utiliser dépend de l'architecture physique (pile spéciale pour les interruptions ou pile de la tâche interrompue)
- cependant la procédure traite-interruption peut aussi être appelée par une tâche; il faut alors la traiter en conséquence.

PRIORITES ET INTERRUPTIONS

PRIORITES

Les tâches et les interruptions ont des priorités et on peut les faire se chevaucher comme cela existe pour certaines architectures physiques;
 subtype ANY_PRIORITY is INTEGER range selon_implémentation;
 subtype PRIORITY is ANY_PRIORITY
 range ANY_PRIORITY'first..selon_implémentation;
 subtype INTERRUPT_PRIORITY is ANY_PRIORITY
 range PRIORITY'last + 1..ANY_PRIORITY'last;



PRIORITÉS DES TÂCHES

La priorité initiale d'une tâche est définie par le pragma :
 pragma PRIORITY(x)
 où x est une valeur_dans_PRIORITY)
 ou encore quand on veut le chevauchement, par le pragma :
 pragma INTERRUPT_PRIORITY(x)
 où x est une valeur_dans_INTERRUPT_PRIORITY

INTERRUPTIONS

Le paquetage Ada.Interrupts contient les opérations pour associer les interruptions aux traite-interruptions créés par le programmeur :
 ATTACH_HANDLER, EXCHANGE_HANDLER...

Les noms des interruptions sont regroupés dans le paquetage fils Interrupts.Name car les noms des interruptions seront définis par l'implémentation et sont du type Interrupt_ID.

INTERRUPTS

Interrupts are modelled as parameterless protected procedures of library level protected objects.

A protected object with such an interrupt handler must have a ceiling priority of range `SYSTEM.INTERRUPT_PRIORITY`.

The predefined package `ADA.INTERRUPTS` contains constants and types for naming interrupts and subprograms for attaching interrupts to handlers dynamically using procedure calls.

The procedures are `ATTACH_HANDLER` and `DETACH_HANDLER`.

The child package : `ADA.INTERRUPTS.NAMES` contains the name of implementation defined interrupts.

INTERRUPTS EXAMPLE

```
with Ada.Calendar, Ada.Interrupts; use Ada.Calendar, Ada.Interrupts;
```

```
protected type TIMER is
```

```
  entry WAIT_FOR_TICK;
```

```
  procedure HANDLE_TIMER_INTERRUPT;
```

```
  pragma Interrupt_Handler(HANDLE_TIMER_INTERRUPT);
```

```
  function GET_CLOCK return TIME;
```

```
private
```

```
  MY_CLOCK : TIME;
```

```
  TICK_OCCURRED : BOOLEAN := FALSE;
```

```
end TIMER;
```

```
protected body TIMER is
```

```
  entry WAIT_FOR_TICK when TICK_OCCURRED is
```

```
  begin
```

```
    TICK_OCCURRED := FALSE;
```

```
  end WAIT_FOR_TICK;
```

```
  procedure HANDLE_TIMER_INTERRUPT is
```

```
  begin
```

```
    ..... -- Update the clock with an operation of TIME type.
```

```
    TICK_OCCURRED := TRUE;
```

```
  end HANDLE_TIMER_INTERRUPT;
```

```
  function GET_CLOCK return TIME is
```

```
  begin
```

```
    return MY_CLOCK;
```

```
  end GET_CLOCK;
```

```
end TIMER;
```

```
MY_TIMER : TIMER;
```

```
ATTACH_HANDLER
```

```
  (MY_TIMER.HANDLE_TIMER_INTERRUPT,  
   Names.TIMER_INT_ID);
```

ASYNCHRONOUS TRANSFER OF CONTROL

A new form of select statement is available to support asynchronous transfer of control.

Such "asynchronous select statements" are composed of two parts :

* **triggering alternative**

a delay statement or entry call (may be followed by a sequence of statements)

* **abortable part - sequence of statements**

On entering the select statement the triggering alternative and abortable part are both executed "in parallel".

If the triggering alternative succeeds before the abortable part is complete this is abandoned, and the sequence of statements following the triggering statement executed.

If the abortable part completes, the triggering alternative is abandoned
For compile simplicity, there must be no accept in the abortable part.

EXAMPLE 1 : TERMINAL INTERRUPT

```
loop
  select
    TERMINAL.WAIT_FOR_INTERRUPT; -- this is an entry call
    PUT_LINE("Interrupted");
  then abort -- This will be abandoned upon terminal interrupt
    PUT_LINE("-> ");
    GET_LINE(COMMAND, LAST);
    PROCESS_COMMAND(COMMAND(1..LAST));
  end select;
end loop;
```

EXAMPLE 2 WITH TIME-OUT

```
select
  delay 5.0; -- this is a delay statement;
             -- there exists also a "delay until deadline_date"
  PUT_LINE("Calculation does not converge");
then abort -- This calculation should finish in 5.0 seconds;
           -- if not, it is assumed to diverge.
  HORRIBLY_COMPLICATED_RECURSIVE_FUNCTION(X, Y);
end select;
```

Contrainte : pas d'accept dans *abortable part

EXAMPLE 3 : CONTROL OF MODE CHANGE USING ALSO ENTRY FAMILIES

```

type SYSTEM_MODE is (NORMAL, MODE2, MODE3, ...);

protected MODE_MANAGER is
  procedure SET_MODE(M : SYSTEM_MODE);
  function GET_MODE return SYSTEM_MODE;
  entry WAIT_FOR_MODE_CHANGE(SYSTEM_MODE)
    (NEW_MODE : out SYSTEM_MODE); -- entry family
private
  CURRENT_MODE : SYSTEM_MODE := NORMAL;
end MODE_MANAGER;

protected body MODE_MANAGER is

  procedure SET_MODE(M : SYSTEM_MODE) is
  begin
    CURRENT_MODE := M;
  end SET_MODE;

  function GET_MODE return SYSTEM_MODE is
  begin
    return CURRENT_MODE;
  end GET_MODE;

  entry WAIT_FOR_MODE_CHANGE
    (for OLD_MODE in SYSTEM_MODE) -- index in entry family
    (NEW_MODE : out SYSTEM_MODE)
    when OLD_MODE /= CURRENT_MODE is
  begin
    NEW_MODE := CURRENT_MODE;
  end WAIT_FOR_MODE_CHANGE;
end MODE_MANAGER;

-- note the use of an index for entry family for protected objects;
-- possible in Ada 95, not in Ada 83; never for task entries in Ada.

```

PROGRAMMING MODE CHANGE

```
task body TYPICAL_TASK is
  MY_MODE : SYSTEM_MODE := NORMAL;
  NEW_MODE : SYSTEM_MODE;
begin
  loop
    select
      MODE_MANAGER.WAIT_FOR_MODE_CHANGE
        (MY_MODE)(NEW_MODE); -- this is an entry call
      MY_MODE := NEW_MODE;
    then abort -- this will be abandoned upon mode change
      case MY_MODE is
        when NORMAL => ... -- Do Normal Mode work
        when MODE2 => ... -- Do Mode 2 work
        when MODE3 => ... -- Do Mode 3 work
      end case;
    end select;
  end loop;
end TYPICAL_TASK;
  -- note that mode change is controlled by another task or by an interrupt
using SET_MODE(M : SYSTEM_MODE)
```

NOTIONS DE STRUCTURATION ET DE PLACEMENT

OUTILS DE GESTION DE CONCURRENCE DANS LES SYSTEMES

processus : objet actif (= processeur logique)
ressource : objet passif, partagé ou non,
accès en exclusion mutuelle ou accès concurrent
(= mémoire logique contenant et classant des informations)

CONSTRUCTION DE L'ARCHITECTURE LOGIQUE

Sous-programmes et modules (ou paquetages en Ada) servent à construire, structurer et composer le tout. C'est un minimum suffisant pour l'architecture logique.

Mais on peut avoir besoin de particulariser certains éléments de l'analyse opérationnelle et de faire apparaître dans l'architecture logique :

des ensembles de processus ou des ensembles de ressources qui ont une cohésion fonctionnelle forte et individualisable (traduit par les notions d'acteur, de serveur, de noeud virtuel...),

On peut vouloir spécifier que les objets d'un tel ensemble soient placés sur le même composant de l'architecture physique répartie (d'où les notions de site, de noeud virtuel, de partition).

Ces ensembles sont aussi souvent des unités pour la tolérance aux pannes, donc des candidats à être redondants et placés sur des composants physiques différents.

Sont utilisés dans ce but des termes et concepts comme

les acteurs de Chorus : regroupement de processus (activités), de ressources (mémoire virtuelle, portes de communication), placement et unités de redondance (acteurs doublés)

les serveurs (du schéma client-serveur) qui gèrent un processus si le service est séquentiel, qui gèrent un groupe de processus si le service est concurrent et le serveur multiprogrammé (un processus par requête de client), qui permettent une implantation répartie avec un serveur à distance

les noeud virtuels ou les partition en Ada 95

Noter la différence entre

- une architecture logique associée à un système d'exploitation (Posix) ou choisie pour une plate-forme comme ADACS (SEMA Group)

ou associée à une approche style CORBA

qui fixent des choix et des interfaces ("API" spécifiques)

- un langage

Le langage doit permettre de mettre en place plusieurs architectures, d'où l'absence de choix figé pour le regroupement et la possibilité de construire des regroupements et des interfaces. Il faut y associer une discipline collective (méthode) de programmation.

RAPPEL DE TECHNIQUES DE CONSTRUCTION DE SYSTEMES (ARCHITECTURE LOGIQUE)

- **regroupement d'entités logiquement proches (packaging)**
- **masquage d'information avec définition d'une interface pour l'utilisateur d'une entité et, séparément, définition de l'implantation de cette entité non accessible à l'utilisateur (information hiding)**
- **programmation par objet, avec composition, extension d'objet par héritage, surcharge des opérateurs sur les objets et choix du bon opérateur soit à la compilation soit à l'exécution (object-oriented programming, inheritance, overload resolution)**
- **construction de systèmes à partir d'un grand nombre d'entités compilées séparément et conservées en bibliothèque, avec un contrôle de type au niveau des interfaces des entités compilées séparément**
- **programmation par classes d'objets régies par relation d'héritage**
- **construction avec des programmes écrits dans des langages différents**
- **accès à la structure de l'architecture physique pour la gestion des périphériques ou des interruptions (low level programming)**
- **récupération propre des erreurs qui se produisent pendant l'exécution (exception handling)**
- **distribution et placement d'entités compilées pour fonctionner ensemble dans une application répartie**
- **limitation des différences entre un programme distribué et un programme non distribué**

DISTRIBUTION ET PLACEMENT EN ADA 95

- un programme ADA est un ensemble de PARTITIONS qui peuvent être exécutées en parallèle
- les partitions sont **ACTIVES** ou **PASSIVES**
- chaque partition active a un espace d'adressage qui lui est propre
- un utilisateur peut placer explicitement, en les nommant, des unités de bibliothèque (sous-programmes, paquetages et unités génériques) dans une partition active; les unités appelées par ces unités nommées par l'utilisateur sont placées automatiquement par le compilateur
- si des unités de bibliothèque ont des variables d'état, il y a une copie de ces variables dans chaque partition qui l'utilise et ces copies évoluent indépendamment dans chaque partition
- la communication entre partition active est réalisée via des paquetages qui sont placés dans chaque partition communicante pour définir l'interface de l'appel à distance ("RCI" : "remote call interface") et implémenter l'appel de procédure distante correspondante ("RPC")
- une partition passive n'a aucun processus ("thread of control")
- une ou plusieurs partitions actives peuvent être placées sur un site processeur (dans l'architecture physique, c'est un élément qui fournit du calcul et de la mémoire)
- une ou plusieurs partitions passives peuvent être placées sur un site mémoire (dans l'architecture physique, c'est un élément qui ne fournit que de la mémoire, adressable par un ou plusieurs sites processeurs; utile pour gérer des multiprocesseurs, en particulier avec des accès mémoire non uniforme ("NUMA architecture": "non uniform memory access architecture"))
- on peut placer plusieurs copies d'une partition active sur des sites différents (une partition est une unité de bibliothèque) pour assurer plus de concurrence ou pour la tolérance aux pannes
- une partition peut être avortée et rechargée dynamiquement à tout moment, sans arrêter le reste du programme.
- toute référence à des données ou tout appel de sous-programme d'une partition à une autre est appelé accès distant ("remote access")
- l'accès distant entre partitions actives est réalisé par des paquetages de bibliothèque (avec des contraintes de visibilité) qui exécutent l'appel de procédure à distance ("RPC" : "remote procedure call"); on peut avoir des **RPC asynchrone** (pragma Asynchronous dans le paquetage qui définit le RCI) pour des procédures avec paramètres "in" seulement.

DISTRIBUTION IN ADA95

ADA95 DISTRIBUTION MODEL

The Ada95 distribution model is based on the notion of active and passive partitions

Every Ada program is made up of one or more partitions

The language only describes the structure and relationship between partitions.

It does not specify how library units :

- (logical) processing units (with private memory)
- shared memory modules
- active partitions
- passive partitions

are aggregated for distribution

Two kinds of partitions are recognized.

Active Partitions

- * correspond to the old notion of an executable main program
- * execute on a processor
- * are composed of independent library units
- * communicate with each other by remote procedure calls only
- * remotely callable procedures are defined in remote call interface package (RCI's)
- * may be both clients and servers of remote calls
- * share variable only by encapsulating them in a passive partition
- * each elaborate independently
- * each have a separate environment task

Passive Partitions

- * provide direct access to data (and possible code) shared between active partitions
- * are mapped to shared memory modules in the architecture
- * may only exist in some programs
- * have no thread of control of their own
- * may not contain tasks or protected objects with entries

CATEGORIZATION PRAGMAS

The program partitioning rules are defined in terms of different kinds of packages :

- * **Preelaborate**
- * **Pure**
- * **Remote_Types**
- * **Shared_Passive**
- * **Remote_Call_Interface (RCI)**
- * **"normal"**

Packages of the first five kinds are identified by the inclusion of a pragma with the corresponding name.

A package which does not have a categorization pragma is a "normal" package

Preelaborate library unit

- * can be elaborated without execution of code at run time (in Ada declarations are elaborated and statements are executed. Thus the elaboration of a declaration makes the things being declared come into existence and then evaluate and assign initial value to them)
- * must not execute a statement and must not require creation of an object of a task type, a protected type with entry declarations or a controlled type

Pure packages

- *are preelaborated packages with further restrictions (they must be freely replicated in different partitions without introducing type inconsistencies. For example a type declared in a pure package is considered to be a single declaration, irrespective of how many times the package is replicated; at contrary, a type declared in a normal package provides a new type declaration every time the package is replicated in a distinct partition)
- * variable and named access types are not allowed unless nested within a subprogram, generic subprogram, task unit or protected unit

Remote_Types packages

- *are preelaborated packages that must not contain any variable declaration within the visible part (a type declared in a Remote_Types package is considered to be a single declaration, as with Pure packages)

Shared_Passive Packages

- * **may only be assigned to passive partitions**
- * **must be preelaborable (i.e. its successful elaboration must not depend on the prior elaboration of the bodies of other units)**
- * **may depend only on pure and shared passive packages**
- * **are not allowed to contain:**
 - **task objects, task types,**
 - **protected objects or types which have entries**
 - **access types designating class wide types.**

Remote_Call_Interface Package

- * **defines the interface between active partitions.**
 - **Its body exists only within a single partition.**
 - **All other occurrences will have stubs allocated.**
- * **must have a preelaborable specification**
- * **can only have children if they reside in the same partition**
- * **must have a specification which depends only on pure, shared passive and other RCI packages**
- * **can not contain the following declaration in their visible part :**
 - **variable (to ensure no remote data access)**
 - **limited type (as tasks and protected types)**
 - **nested generic type declaration**
 - **subprograms with parameters of limited types unless they have Read and Write attribute**
 - **subprograms with access parameters**
 - **access types**

PARTITION COMPOSITION RULES

The restrictions placed on the different categories of library unit do not define partitions.

The partitions are defined in an implementation specific way after the constitute units have been compiled, and must be consistent with the rules outlined above.

Each RCI package and shared package that is to be present in a distributed system must be assigned to exactly one partition by the user.

Only shared passive package may be assigned to passive partitions.

Child units of RCI packages must be assigned to the same active partition as their parent

POST COMPILATION PARTITIONING

The attribute `Partition_ID` may be applied to an RCI or a shared passive package and returns a unique value (partition ID).

This provides run time identification of the partition in which the named package was elaborated

Prior to building a partition, all RCI and shared passive packages must be assigned to a partition using the syntax defined by the implementation

The following entities are then linked into the partition by the linker *
the package explicitly assigned to the partition

- * the units on which these depend (if any)

- * the main program for the partition (if any)

Note that RCI or shared passive packages assigned to other partitions are not included in a partition

Environment Tasks

Once loaded, an active partition is elaborated independently by its environment task (every active partition has its own)

```
task Environment_Task;
task body Environment_Task is
  -- the environment declarative part.
  -- the library items mapped to a partition are defined here
begin
  - main program, if there is one, is called here
end;
```

After elaboration, active partitions execute independently except to communicate

PARTITION COMMUNICATION SYSTEM

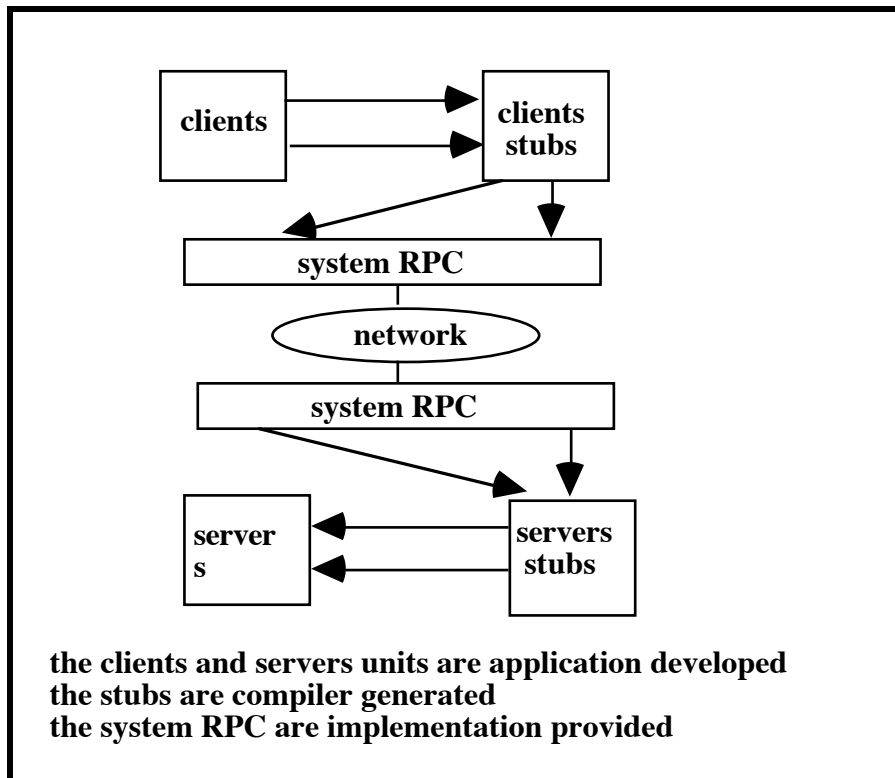
Active partitions communicate by remote procedure calls to the subprograms defined in their RCI packages.

When the implementation detects a dependency between on an RCI package assigned to a different partition it inserts the appropriate stubs to implement a remote call RPCs are defined to have "at most once semantics", and are supported by the notional Partition Communication System (PCS)

In addition to the normal blocking RPCs, it is also possible to define asynchronous RPCs using the pragma Asynchronous

This indicates that calls on the procedure may return without waiting for the call to complete

It may only be applied to procedures with "in" parameters.



```
with Ada.Streams;
package System.RPC is
```

```
  type PARTITION_ID is range 0 .. Max; --implementation_defined;
```

```
  procedure PARTITION(
    PTN : in PARTITION_ID;
    ASSIGN : LIBRARY_UNITS); --possible implementation
```

```
  COMMUNICATIONS_ERROR : exception;
```

```
  type Params_Stream_Type
    (INITIAL_SIZE : Ada.Streams.Stream_Element_Count) is
    new Ada.Streams.Root_Stream_Type with private;
```

```
  procedure READ (
    STREAM : in out Params_Stream_Type;
    ITEM    : out Ada.Streams.Stream_Element_Array;
    LAST    : out Ada.Streams.Stream_Element_Offset);
```

```
  procedure WRITE (
    STREAM : in out Params_Stream_Type;
    ITEM    : in Ada.Streams.Stream_Element_Array);
```

```
-- Synchronous call includes RESULT parameter
```

```
  procedure DO_RPC (
    PARTITION : in PARTITION_ID;
    PARAMS     : access Params_Stream_Type;
    RESULT     : access Params_Stream_Type);
```

```
-- Asynchronous call
```

```
  procedure DO_APC (
    PARTITION : in PARTITION_ID;
    PARAMS     : access Params_Stream_Type);
```

```
-- the handler for coming RPCs
```

```
  type RPC_RECEIVER is access procedure (
    PARAMS     : access Params_Stream_Type;
    RESULT     : access Params_Stream_Type);
```

```
  procedure ESTABLISH_RPC_RECEIVER(
    PARTITION : in PARTITION_ID;
    RECEIVER  : in RPC_RECEIVER);
```

```
private
```

```
  -- ... implementation defined
```

```
end System.RPC;
```

```

package TAPES_PKG is -- EXAMPLE OF DISTRIBUTED SERVICE
  pragma PURE;
  type TAPE is abstract tagged limited private; ....
  procedure COPY
    (FROM, TO: access TAPE; NUM_RECS : Natural) is abstract;
  procedure REWIND (T : access TAPE) is abstract;
  -- More operation
end TAPES_PKG;

with TAPES_PKG;
package NAME_SERVER is
  pragma Remote_Call_Interface;
  type TAPE_PTR is access all TAPES_PKG.TAPE'CLASS;
  function FIND (NAME : STRING) return TAPE_PTR;
  procedure REGISTER (NAME : STRING; T : TAPE_PTR);
  procedure REMOVE (T : TAPE_PTR);
  -- More operations
end NAME_SERVER;

package TAPE_DRIVER is ... end TAPE_DRIVER;
  -- Declarations are not shown, they are irrelevant here

with TAPES_PKG, NAME_SERVER;
package body TAPE_DRIVER is
  type NEW_TAPE is new TAPES_PKG.TAPE with ...
  procedure COPY
    (FROM, TO: access NEW_TAPE; NUM_RECS: Natural) is
    begin. . end COPY;
  procedure REWIND (T: access NEW_TAPE) is
  begin. . end REWIND;
  -- Objects remotely accessible through NAME_SERVER operations
  TAPE1, TAPE2 : NEW_TAPE;
begin
  NAME_SERVER.REGISTER ("NINE TRACK", TAPE1'ACCESS);
  NAME_SERVER.REGISTER ("SEVEN_TRACK", TAPE2'ACCESS);
end TAPE_DRIVER;

with TAPES_PKG, NAME_SERVER;
procedure TAPE_CLIENT is
  T1, T2 : NAME_SERVER.TAPE_PTR;
begin
  T1 := NAME_SERVER.FIND ("NINE TRACK");
  T2 := NAME_SERVER.FIND ("SEVEN_TRACK");
  TAPES_PKG.REWIND (T1);
  TAPES_PKG.REWIND (T2);
  TAPES_PKG.COPY (T1, T2, 3);
end TAPE_CLIENT;

```

Notes pour l'exemple

le paquetage **TAPES_PKG** définit une classe

NAME_SERVER est serveur de noms pour tout le programme distribué

TAPE_DRIVER est un paquetage "normal" qui se lie dynamiquement au serveur de noms et cela permet d'ajouter déplacer ou retirer des bandes sans changer le code du client

TAPE_CLIENT fait appel à des opérandes T1 et T2 d'un type "remote access-to-class-wide". Cela suffit au compilateur pour savoir générer les talons ("stub") nécessaires aux appels de RPC

Exemple de déclaration de partition

```
PARTITION(PTN => 1, ASSIGN => (1 => NAME_SERVER));
```

```
PARTITION(PTN => 2, ASSIGN => (1 => TYPE_DRIVER));
```

```
PARTITION(PTN => 3, ASSIGN => (1 => TAPE_CLIENT));
```

