Sistemas Distribuidos de Tiempo Real

Apuntes: TEMA 3

Por: J. Javier Gutiérrez gutierjj@unican.es
http://www.ctr.unican.es/

Grupo de Computadores y Tiempo Real, Universidad de Cantabria

Sistemas distribuidos de tiempo real

PARTE II: Modelos de distribución

- **TEMA 3. Modelo de distribución de Ada**
- **TEMA 4. Modelo de distribución de CORBA y RT-CORBA**

DSA - Distributed Systems Annex
Anexo E de Ada 95

El DSA de Ada 95 proporciona un modo flexible para distribuir programas Ada en plataformas con múltiples procesadores

Definición Ada de sistema distribuido: interconexión de

- **uno o más nudos procesadores (recurso con capacidad computacional y de almacenamiento)**
- **y cero o más nudos de almacenamiento (que sólo tiene capacidad de almacenamiento, direccionable por uno o más nudos procesadores)**

Definición de programa distribuido:

- **una o más particiones que ejecutan independientemente (excepto cuando se comunican) en un sistema distribuido**

DSA - Distributed Systems Annex Anexo E de Ada 95 (cont.)

Al proceso de mapear las particiones en los nodos de un sistema distribuido se llama configuración de las particiones del programa

La distribución se basa en los conceptos de:

- particiones de programa
- y llamadas a procedimiento remoto (RPCs)

El lenguaje no proporciona interfaces ni semántica que permita usar el DSA para aplicaciones distribuidas con requisitos de tiempo real

- aunque en las aplicaciones se puede usar el anexo D

Particiones

Definición:

- Una partición es una entidad que puede ejecutar en paralelo con cualquier otra, posiblemente en un espacio de direcciones diferente, y posiblemente en un computador diferente

Tipos de particiones:

- Activas: contienen threads, que pueden ser un programa principal o alguna tarea
- Pasivas: no contiene un thread de control propio; los datos y subprogramas son accesibles para las particiones activas

`D'Partition_ID` es un atributo que devuelve un entero que identifica la partición en la que D ha sido elaborado

Categorización de unidades de biblioteca

Categorías jerarquizadas de las unidades de biblioteca:

- *Pure*: puede depender sólo de otras unidades declaradas Pure
- *Shared Passive*: puede depender sólo de otras unidades Shared Passive o Pure
- *Remote Types*: puede depender de otras unidades Remote Types o de las anteriores (el cuerpo de la librería no tiene restricciones)
- *Remote Call Interface*: puede depender de otras unidades Remote Call Interface o de las anteriores (el cuerpo de la librería no tiene restricciones)
- *Normal*: sin restricciones

Categorización de unidades de biblioteca (cont.)

La categorización se realiza mediante pragmas:

- pragma `Pure`
- pragma `Shared_Passive`
- pragma `Remote_Types`
- pragma `Remote_Call_Interface`

Se permite a la implementación definir nuevos pragmas de categorización

Unidades Shared Passive

Se utilizan para manejar datos globales compartidos entre particiones activas

Una partición pasiva puede contener unidades de biblioteca categorizadas como Pure o Shared Passive

No puede contener declaraciones de access a tipos class-wide, tipos tarea, o tipos protegidos con entries

Sintaxis:

```
pragma Shared_Passive [(nombre de unidad de biblioteca)]
```


Unidades Shared Passive (cont.)

```
package SharedObjects is

   pragma Shared_Passive;

   Max : Positive := 10;

   type Index_Type is range 1 .. Max;
   type Rate_Type is new Float;

   type Rates_Type is array (Index_Type) of Rate_Type;

   External_Synchronization : Rates_Type;
```

Unidades Shared Passive (cont.)

```
   protected Internal_Synchronization is
      procedure Set
         Index : in Index_Type;
         Rate  : in Rate_Type);
      procedure Get
         Index : in Index_Type;
         Rate  : out Rate_Type);
   private
      Rates : Rates_Type;
   end Internal_Synchronization;

end SharedObjects;
```


Unidades Remote Types

Pensada para la definición de tipos que se usen en la comunicación entre particiones activas.

Sintaxis:

```
pragma Remote_Types [(nombre de unidad de biblioteca)]
```

No debe contener declaraciones de variables en la parte visible

Un tipo access declarado en la parte visible de estas unidades de biblioteca se llama tipo access remoto

- este tipo será un access a subprograma o un access general a un class-wide limited private

Unidades Remote Types (cont.)

Cualquier puntero no remoto que forme parte de un tipo remoto debe especificar los atributos de usuario `Read` y `Write`

Restricciones al uso de access remotos a subprogramas:

- un valor de un access remoto a un subprograma será convertido sólo a otro access remoto a un subprograma
- un valor de un access remoto a un class-wide será convertido sólo a otro access remoto a un class-wide
- el prefijo del atributo `Access` que contiene un valor de un tipo access remoto a un subprograma denotará estáticamente a un subprograma remoto

Transmisión de estructuras dinámicas

```
with Ada.Streams; use Ada.Streams;
-- Lista enlazada
package StringArrayStream is
   pragma Remote_Types;

   type List is private;
   procedure Append (L : access List; O : in String);
   function  Delete (L : access List) return String;

private
   type String_Access is access String;

   type Node;
   type List is access Node;
```


Transmisión de estructuras dinámicas (cont.)

```
   type Node is record
      Content : String_Access;
      Next    : List;
   end record;

   procedure Read
      (S : access Root_Stream_Type'Class;
       L : out List);
   procedure Write
      (S : access Root_Stream_Type'Class;
       L : in List);
   for List'Read use Read;
   for List'Write use Write;
end StringArrayStream;
```


Transmisión de estructuras dinámicas (cont.)

```
package body StringArrayStream is
   procedure Read
      (S : access Root_Stream_Type'Class;
       L : out List) is
   begin
      if Boolean'Input (S) then
         L := new Node;
         L.Content := new String'(String'Input (S));
         List'Read (S, L.Next);
      else
         L := null;
      end if;
   end Read;
```

Transmisión de estructuras dinámicas (cont.)

```
   procedure Write
     (S : access Root_Stream_Type'Class;
      L : in List) is
   begin
     if L = null then
       Boolean'Output (S, False);
     else
       Boolean'Output (S, True);
       String'Output (S, L.Content.all);
       List'Write (S, L.Next);
     end if;
   end Write;
   --  [...] Other services
end StringArrayStream;
```


Unidades Remote Call Interface RCI

Pensadas para ser usadas como interfaces para llamadas a procedimientos remotos, RPCs, entre particiones activas

Incluye los siguientes pragmas:

```
pragma Remote_Call_Interface
            [(nombre de unidad de biblioteca)]
```

- los subprogramas declarados en la parte visible de una unidad de biblioteca RCI son subprogramas remotos

```
pragma All_Calls_Remote
            [(nombre de unidad de biblioteca)]
```

- la unidad a la que se aplica debe ser RCI
- obliga a todas las llamadas a bajar al subsistema de comunicación de particiones (Partition Communication Subsystem, PCS)

Unidades Remote Call Interface RCI (cont.)

Restricciones de la parte visible de una unidad RCI:

- no contendrá declaraciones de variables
- no contendrá declaraciones de tipos limited
- no contendrá una declaración genérica anidada
- no contendrá (ni podrá ser) la declaración de un subprograma al que se le aplique el pragma Inline
- no contendrá (ni podrá ser) una declaración de subprograma (o access a subprograma) con parámetros de tipo access, o con parámetros formales de tipo limited a menos que el usuario haya especificado los atributos `Read` y `Write`
- cualquier hijo público será una unidad RCI

Unidades Remote Call Interface RCI (cont.)

Reglas de post-compilación:

- una unidad RCI se debe asignar al menos a una partición
- una unidad RCI cuyo padre es también una unidad RCI se asignará sólo a la misma partición que el padre

Requisitos de la implementación:

- el pragma `All_Calls_Remote` obliga a todas las llamadas a bajar al PCS; las llamadas a estos subprogramas desde dentro de la región declarativa de la unidad son locales y no van a través del PCS

Llamadas a procedimiento remoto

Una RPC (Remote Subprogram Call) es una llamada a un subprograma que se encuentra en otra partición

La partición que origina la llamada se llama partición llamante (calling partition) y la partición que ejecuta y contiene el cuerpo del subprograma se llama partición llamada (called partition)

Las llamadas que retornan antes de que se haya completado la ejecución se llaman APCs (Asynchronous Procedure Calls)

Llamadas a procedimiento remoto (cont.)

Tres maneras de realizar una RPC:

- una llamada directa a un subprograma declarado en una RCI
- una llamada indirecta a través del valor de un access remoto a un tipo subprograma
- una llamada de despacho con un operando controlado designado por el valor de un access remoto a un tipo class-wide

La primera implementa un enlace estático entre la partición llamante y llamada, las dos últimas implementan un enlace dinámico

Llamadas a procedimiento remoto (cont.)

En la ejecución de un subprograma remoto los parámetros se pasan usando una representación de streams:

- ***marshalling***: conversión a stream
- ***unmarshalling***: recuperación de los parámetros del stream

El código que se añade para realizar una llamada remota se denomina ***stub***:

- ***calling stub***: sustituye al cuerpo del subprograma en la partición llamante
- ***receiving stub***: ejecuta el subprograma en la partición llamada

Llamadas a procedimiento remoto (cont.)

La tarea que ejecuta un subprograma remoto se bloquea hasta que finaliza a menos que sea una llamada asíncrona

Las excepciones se propagan del modo normal salvo que la llamada sea asíncrona

Si sucede algún error en una llamada remota que le impida completarse se eleva la excepción `Communication_Error`

Las llamadas remotas con tipos class-wide pueden elevar la excepción `Program_Error` si los parámetros no coinciden con las etiquetas declaradas

Llamadas a procedimiento remoto (cont.)

Requisitos de la implementación:

- La implementación de las llamadas a subprogramas remotos debe ser conforme al PCS (Partition Communication Subsystem) definido en `System.RPC`
- El stub llamante usará los procedimientos `Do_RPC` o `Do_APC`
- El stub de recepción será ejecutado por un `RPC_Receiver`

Llamadas asíncronas

Una llamada asíncrona permite retornar antes de que se complete la ejecución.

Se expresa mediante el siguiente pragma:

```
pragma Asynchronous [(nombre local)]
```

El nombre local puede ser:

- uno o más procedimientos remotos; los parámetros formales deben ser de modo `in`
- el primer subtipo de un tipo access a procedimiento remoto; también parámetros de modo `in`
- el primer subtipo de un access remoto a un tipo class-wide

Partition Communication Subsystem PCS

El subsistema de comunicación de particiones da soporte a la comunicación entre particiones activas en un programa distribuido

Para la implementación se proporciona el paquete definido por el lenguaje `System.RPC` con la siguiente interfaz:

```
with Ada.Streams; -- see 13.13.1
package System.RPC is
   type Partition_ID is range 0 .. implementation-defined;
   Communication_Error : exception;
   type Params_Stream_Type (
     Initial_Size : Ada.Streams.Stream_Element_Count) is new
     Ada.Streams.Root_Stream_Type with private;
```


Partition Communication Subsystem PCS (cont.)

```
   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
   procedure Do_RPC(
      Partition  : in Partition_ID;
      Params     : access Params_Stream_Type;
      Result     : access Params_Stream_Type);
```

Partition Communication Subsystem PCS (cont.)

```
   -- Asynchronous call
   procedure Do_APC(
      Partition  : in Partition_ID;
      Params     : access Params_Stream_Type);
   -- The handler for incoming 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
   ... -- not specified by the language
end System.RPC;
```


Partition Communication Subsystem PCS (cont.)

Comentarios a la interfaz:

- Un valor de tipo `Partition_ID` se usa para identificar la partición
- Un objeto de tipo `Params_Stream_Type` se usa para identificar el subprograma remoto y los parámetros y resultado que se manejan en las operaciones de marshalling y unmarshalling
- Las operaciones `Read` y `Write` sobrescriben las operaciones abstractas para el tipo `Params_Stream_Type`

Do_RPC y Do_APC

Envían un mensaje a la partición activa indicada por el parámetro.

Después de enviar el mensaje `Do_RPC` se bloquea hasta que llega la respuesta o se produce un error de comunicaciones

`Do_APC` retorna nada más enviar el mensaje

Si se aborta una llamada a `Do_RPC` se envía un mensaje de cancelación a la partición llamada para requerir que aborte la ejecución de la operación remota

RPC_Receiver

El procedimiento `Establish_RPC_Receiver` se llama una vez inmediatamente después de la elaboración de la unidad de biblioteca de una partición activa, si la partición contiene una RCI

El parámetro `Receiver` designa un procedimiento *RPC-Receiver* suministrado por la implementación que manejará todas las llamadas remotas recibidas por la partición

La implementación del *RPC-Receiver* debe ser reentrante permitiendo llamadas concurrentes desde el PCS para que se puedan servir llamadas concurrentes dentro de la partición

RPC_Receiver (cont.)

Requisitos de documentación:

- la implementación del PCS debe documentar si el *RPC-Receiver* es invocado desde tareas concurrentes y si hay un límite al número de tareas

Permisos de implementación:

- el PCS puede contener interfaces para el paso de mensajes explícitas
- también puede contener otras interfaces que deberán darse como paquetes hijos del `System.RPC`

Consejo de implementación:

- el PCS en la partición llamada permitirá que múltiples tareas llamen al *RPC-Receiver* con diferentes mensajes y permitirá que se bloqueen hasta que el correspondiente subprograma retorne

Ejemplo 1 : access remoto a class-wide

- Paquete `Tapes` con las declaraciones necesarias de tipo y operaciones primitivas
- Paquete RCI `Name_Server` elaborado en una partición activa separada que proporciona un servicio de nombres (registrar y buscar) al programa distribuido a través de RPCs
- Un paquete normal `Tape_Driver` que se elabora en la partición configurada en el nudo procesador en el que se encuentra conectado el dispositivo *Tape*:
  - las operaciones abstractas se sobrescriben para soportar los dispositivos locales *Tape1* y *Tape2*

Ejemplo 1: access remoto a class-wide (cont.)

- El paquete `Tape_Driver` no es visible a sus clientes pero exporta los dispositivos como objetos remotos a través de `Name_Server`
  - esto permite añadir, borrar o reemplazar dispositivos dinámicamente sin la modificación del código del cliente.
- Un procedimiento `Tape_Client` que hace referencia sólo a declaraciones de los paquetes `Tapes` y `Name_Server`
- Antes de usar por primera vez una *Tape* es necesario identificarla en el `Name_Server` (a partir de ahí, se usa la identificación para acceder al dispositivo)
- Los valores de access `Tape_Ptr` incluyen la información necesaria para completar las operaciones de despacho que resultan de acceder a los operandos `T1` y `T2`

Ejemplo 1: package Tapes

```
package Tapes is
   pragma Pure(Tapes);
   type Tape is abstract tagged limited private;
   -- Primitive dispatching operations where
   -- Tape is controlling operand
   procedure Copy (From, To : access Tape;
                   Num_Recs : in Natural) is abstract;
   procedure Rewind (T : access Tape) is abstract;
   -- More operations
private
   type Tape is ...
end Tapes;
```


Ejemplo 1: package Name_Server

```
with Tapes;
package Name_Server is
   pragma Remote_Call_Interface;
   -- Dynamic binding to remote operations is achieved
   -- using the access-to-limited-class-wide type Tape_Ptr
   type Tape_Ptr is access all Tapes.Tape'Class;
   -- The following statically bound remote operations
   -- allow for a name-server capability in this example
   function  Find     (Name : String) return Tape_Ptr;
   procedure Register (Name : in String; T : in Tape_Ptr);
   procedure Remove   (T : in Tape_Ptr);
   -- More operations
end Name_Server;
```

Ejemplo 1: package Tape_Driver

```
package Tape_Driver is
   -- Declarations are not shown, they are irrelevant here
end Tape_Driver;

with Tapes, Name_Server;
package body Tape_Driver is
   type New_Tape is new Tapes.Tape with ...
   procedure Copy(From, To : access New_Tape;
                  Num_Recs: in Natural) is
   begin
       . . .
   end Copy;
```


Ejemplo 1: package Tape_Driver (cont.)

```
   procedure Rewind (T : access New_Tape) is
   begin
       . . .
   end Rewind;

   -- Objects remotely accessible through use
   -- of Name_Server operations
   Tape1, Tape2 : aliased New_Tape;

begin
   Name_Server.Register ("NINE-TRACK",  Tape1'Access);
   Name_Server.Register ("SEVEN-TRACK", Tape2'Access);
end Tape_Driver;
```

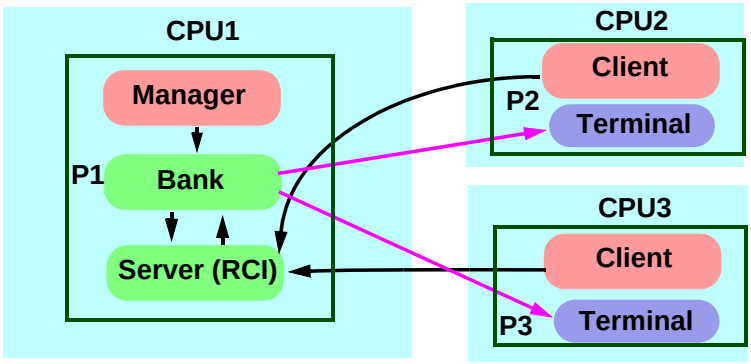

Ejemplo 1: procedure Tape_Client

```
with Tapes, Name_Server;
-- Tape_Driver is not needed and thus not mentioned in the
-- with_clause

procedure Tape_Client is
   T1, T2 : Name_Server.Tape_Ptr;
begin
   T1 := Name_Server.Find ("NINE-TRACK");
   T2 := Name_Server.Find ("SEVEN-TRACK");
   Tapes.Rewind (T1);
   Tapes.Rewind (T2);
   Tapes.Copy (T1, T2, 3);
end Tape_Client;
```

Ejemplo 2: simulación de un banco

Ejemplo proporcionado en la distribución GLADE 2007:



Ejemplo 2: simulación de un banco

Comentarios sobre el ejemplo:

- el banco ofrece un servicio que es controlado
  - localmente por el *Manager*
  - y de forma remota por el *Server* (RCI)
- el *Server* define un puntero a un objeto `Terminal` con la operación de registro; operación que finalmente se realizará en el banco
- cada cliente registra en el *Server* un terminal
- en la operación de transferencia, el banco notifica al cliente destinatario a través del terminal registrado

Ejemplo 2: package Types

```
package Types is
   pragma Pure;

   N_Customer_IDs : constant := 20;
   type Customer_ID is range 0 .. N_Customer_IDs;
   subtype Customer_Type is String;
   subtype Password_Type is String;

   Wrong_Password  : exception;
   Wrong_Customer  : exception;
   Wrong_Donator   : exception;
   No_More_IDs     : exception;

end Types;
```

Ejemplo 2: package Bank

```
with Types; use Types;
with Server; use Server;
package Bank is
   function Balance
     (Customer : in Customer_Type;
      Password : in Password_Type)
      return Integer;
   procedure Create
     (Customer : in Customer_Type;
      Password : in Password_Type;
      Deposit  : in Positive);
   procedure Deposit
     (Customer : in Customer_Type;
      Amount   : in Positive);
```


Ejemplo 2: package Bank (cont.)

```
   procedure Register
     (Terminal : in Terminal_Access;
      Customer : in Customer_Type;
      Password : in Password_Type);
   procedure Transfer
     (Donator  : in Customer_Type;
      Password : in Password_Type;
      Amount   : in Positive;
      Customer : in Customer_Type);
   procedure Withdraw
     (Customer : in Customer_Type;
      Password : in Password_Type;
      Amount   : in Positive);
```


Ejemplo 2: package Bank (cont.)

```
   function Is_Activated (ID : Customer_ID) return Boolean;
   function Get_Customer (ID : Customer_ID) return
                                          Customer_Type;
   function Get_Password (ID : Customer_ID) return
                                          Password_Type;
   function Get_Balance  (ID : Customer_ID) return Integer;

end Bank;
```

Ejemplo 2: package Server

```
with Types; use Types;
with Alarm; use Alarm;

package Server is
   pragma Remote_Call_Interface;
   type Terminal_Access is access all Terminal_Type'Class;
   procedure Register
     (Terminal : in Terminal_Access;
      Customer : in Customer_Type;
      Password : in Password_Type);
   function Balance
     (Customer : Customer_Type;
      Password : Password_Type)
      return Integer;
```


Ejemplo 2: package Server (cont.)

```
   procedure Deposit
     (Customer : in Customer_Type;
      Amount   : in Positive);
   procedure Withdraw
     (Customer : in Customer_Type;
      Password : in Password_Type;
      Amount   : in Positive);
   procedure Transfer
     (Donator  : in Customer_Type;
      Password : in Password_Type;
      Amount   : in Positive;
      Customer : in Customer_Type);

end Server;
```


Ejemplo 2: package Alarm

```
with Types; use Types;
package Alarm is
   pragma Pure;
   type Terminal_Type is abstract tagged limited private;

   procedure Notify
     (Terminal : access Terminal_Type;
      Donator  : in Customer_Type;
      Amount   : in Integer) is abstract;

private
   type Terminal_Type
        is abstract tagged limited null record;
end Alarm;
```

Ejemplo 2: package Message

```
with Alarm; use Alarm;
with Types; use Types;

package Message is
   pragma Remote_Types;

   type Alarm_Terminal is new Terminal_Type
                              with null record;
   procedure Notify
     (Terminal : access Alarm_Terminal;
      Donator  : in Customer_Type;
      Amount   : in Integer);

end Message;
```


Ejemplo 2: package Terminal

```
with Message; use Message;

package Terminal is

   My_Terminal : aliased Alarm_Terminal;

end Terminal;
```


Ejemplo 2: procedure Manager

```
with Bank; use Bank;
with Types; use Types;

...
procedure Manager is
...
begin

   loop
      -- Menú: Create, Load, Print, Quit, Save
   end loop;

end Manager;
```

Ejemplo 2: procedure Client

```
with Server; use Server;
with Message; use Message;
with Terminal; use Terminal;
...
procedure Client is
   ...
begin
   Register (My_Terminal'Access, Customer, Password);
   loop
      -- Menú: Balance, Deposit, Quit, Transfer, Withdraw
   end loop;
end Client;
```


Ejemplo 2: procedure Transfer del package Bank

```
   procedure Transfer
     (Donator  : in Customer_Type;
      Password : in Password_Type;
      Amount   : in Positive;
      Customer : in Customer_Type) is
      ID_1 : Customer_ID := Find (Donator);
      ID_2 : Customer_ID := Find (Customer);
      Term : Terminal_Access;
   begin
      ...
         Term := Accounts (ID_2).Terminal;
         Notify (Term, Donator, Amount);
      ...
   end Transfer;
```