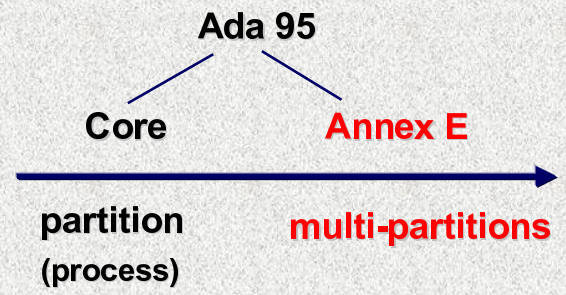
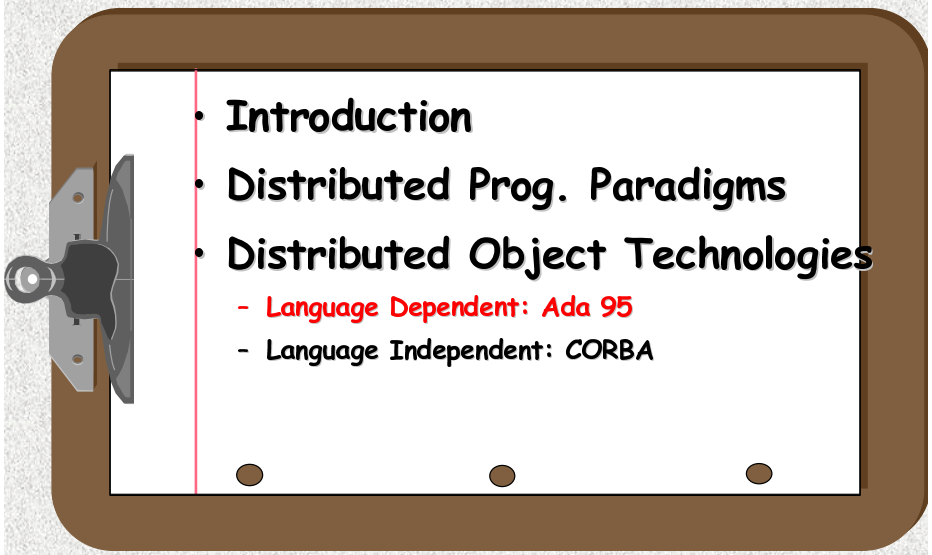
Ada 95 Distributed Programming



A partition comprises one or more Ada packages



Supported Paradigms

- Client/Server Paradigm (RPC)
  - Synchronous / Asynchronous
  - Static / Dynamic
- Distributed Objects
- Shared Memory

Ada 95 Distributed Systems Annex

Remote_Types

- Allows the definition of a remote access types
  - Remote access to subprogram
  - Remote reference to objects
    (ability to do dynamically dispatching calls across the network)

Ada Distributed Application

- No need for a separate interfacing language as in CORBA (IDL)
  - Ada is the IDL
- Some packages categorized using pragmas
  - Remote_Call_Interface (RCI)
  - Remote_Types
  - Shared_Passive (SP)
- All packages except RCI & SP duplicated on partitions using them

Shared_Passive

- A Shared_Passive package contains variables that can be accessed from distinct partitions
- Allows support of shared distributed memory
- Allows persistence on some implementations

Remote_Call_Interface (RCI)

- Allows subprograms to be called remotely
  - Statically bound RPCs
  - Dynamically bound RPCs
    (remote access to subprogram)

Write App

```
package Types is
   type Device is (Furnace, Boiler,...);
   type Pressure is ...;
   type Temperature is ...;
end Types;
```

```
with Types; use Types;
package Sensors is
   function Get_P (D: Device) return Pressure;
   function Get_T (D: Device) return Temperature;
end Sensors;
```

```
with Types; use Types;
with Sensors;
procedure Client_1 is
   P := Sensors.Get_P (Boiler);
```

```
with Types; use Types;
with Sensors;
procedure Client_2 is
   T := Sensors.Get_T (Furnace);
```


Building a Distributed App in Ada 95

1. Write app as if non distributed.
2. Identify remote procedures, shared variables, and distributed objects & categorize packages.
3. Build & test non-distributed application.
4. Write a configuration file for partitionning your app.
5. Build partitions & test distributed app.

Categorize

```
package Types is
   pragma Pure;
   type Device is (Furnace, Boiler,...);
   type Pressure is ...;
   type Temperature is ...;
end Types;
```

```
with Types; use Types;
package Sensors is
   pragma Remote_Call_Interface;
   function Get_P (D:Device) return Pressure;
   function Get_T (D:Device) return Temperature;
end Sensors;
```

```
with Types; use Types;
with Sensors;
procedure Client_1 is
   P := Sensors.Get_P (Boiler);
```

```
with Types; use Types;
with Sensors;
procedure Client_2 is
   T := Sensors.Get_T (Furnace);
```


Remote_Call_Interface

An Example

Partition

```
configuration Config_1 is
   Node_A : Partition := (Sensors);
   Node_B : Partition := (Client_1);
   Node_C : Partition := (Client_2);
end Config_1;
```

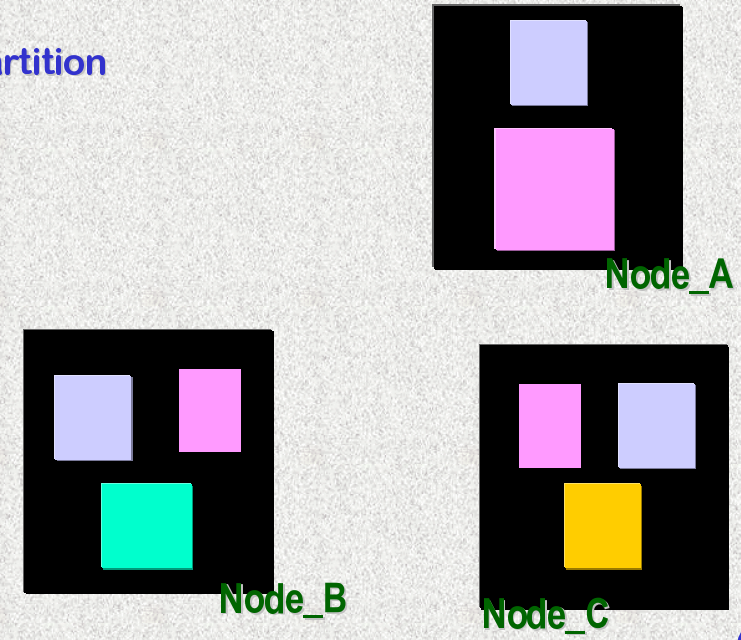

Build & Test

```
package Types is
   pragma Pure;
   type Device is (Furnace, Boiler,...);
   type Pressure is ...;
   type Temperature is ...;
end Types;
```

```
with Types; use Types;
package Sensors is
   pragma Remote_Call_Interface;
   function Get_P (D:Device) return Pressure;
   function Get_T (D:Device) return Temperature;
end Sensors;
```

```
with Types; use Types;
with Sensors;
procedure Client_1 is
   P := Sensors.Get_P (Boiler);
```


Partition



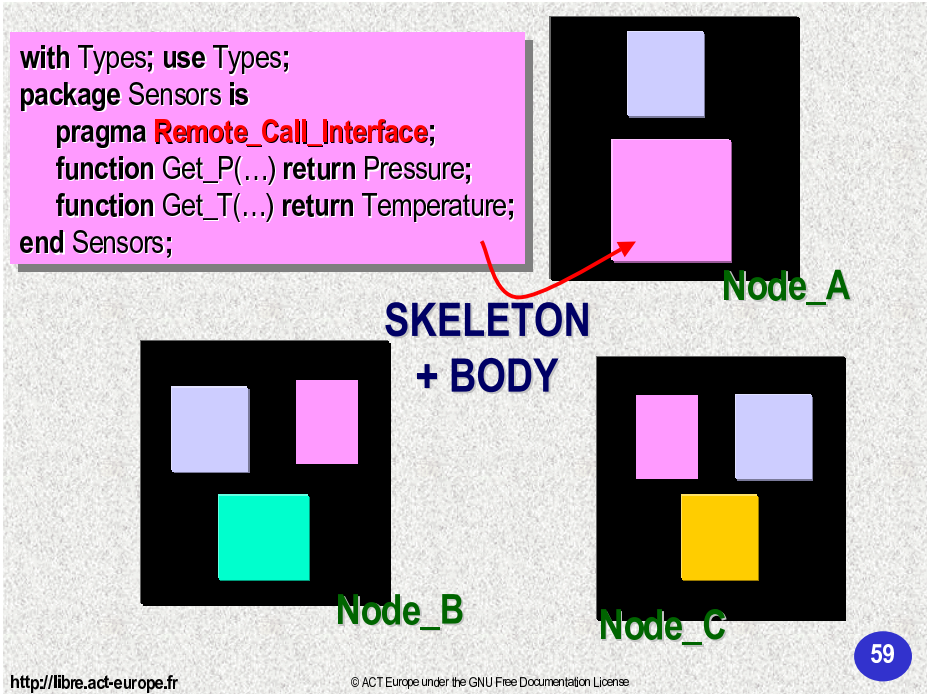
Build & Test

```
package Types is
   pragma Pure;
   type Device is (Furnace, Boiler,...);
   type Pressure is ...;
   type Temperature is ...;
end Types;
```

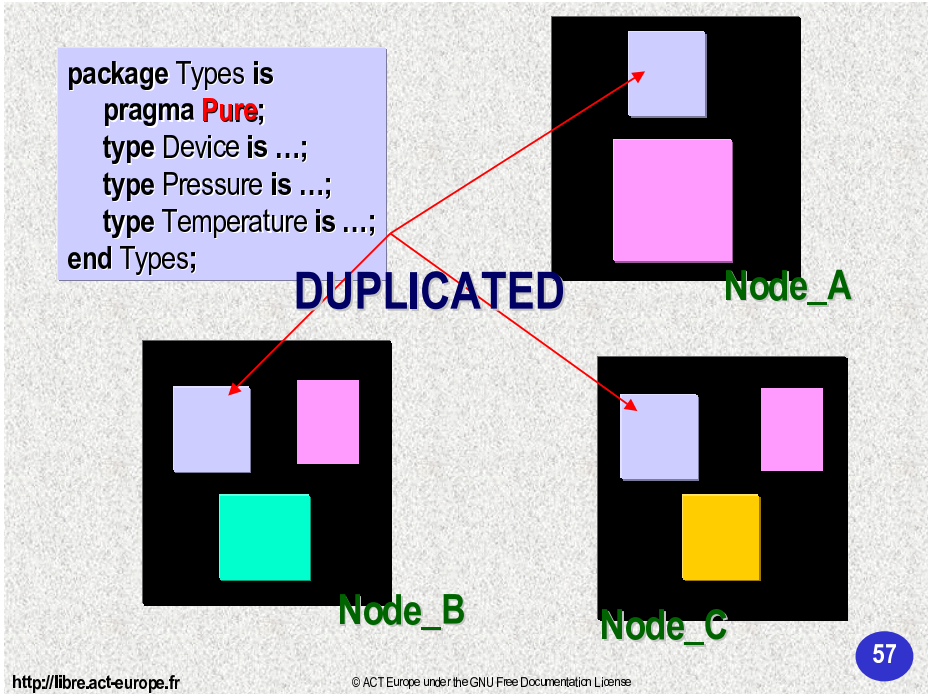
```
with Types; use Types;
package Sensors is
   pragma Remote_Call_Interface;
   function Get_P (D:Device) return Pressure;
   function Get_T (D:Device) return Temperature;
end Sensors;
```

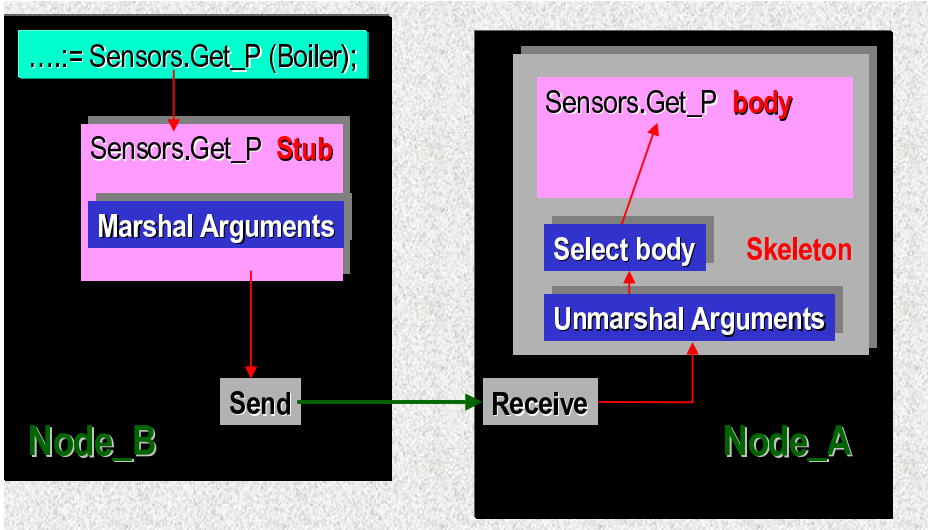
```
with Types; use Types;
with Sensors;
procedure Client_2 is
   T := Sensors.Get_T (Furnace);
```

```
with Types; use Types;
package Sensors is
   pragma Remote_Call_Interface;
   function Get_P(...) return Pressure;
   function Get_T(...) return Temperature;
end Sensors;
```

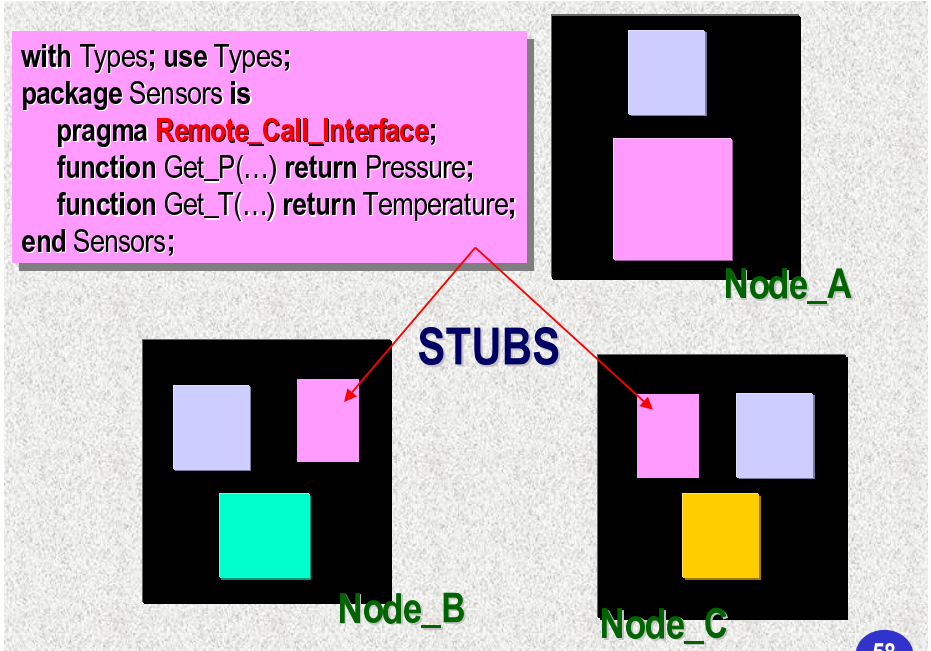



```
package Types is
   pragma Pure;
   type Device is ...;
   type Pressure is ...;
   type Temperature is ...;
end Types;
```






```
with Types; use Types;
package Sensors is
   pragma Remote_Call_Interface;
   function Get_P(...) return Pressure;
   function Get_T(...) return Temperature;
end Sensors;
```



Write App

```
package Alerts is
   type Alert is abstract tagged private;
   type Alert_Ref is access all Alert'Class;
   procedure Handle (A : access Alert);
   procedure Log    (A : access Alert) is abstract;
private
   ...
end Alerts;
```

```
package Alerts.Pool is
   procedure Register (A : Alert_Ref);
   function  Get_Alert return Alert_Ref;
end Medium;
```

```
with Alerts, Alerts.Pool; use Alerts;
procedure Process_Alerts is
begin
   loop
      Handle (Pool.Get_Alert);
   end loop;
end Process_Alerts;
```


Asynchronous Calls

```
with Types; use Types;
package Sensors is
   pragma Remote_Call_Interface;
   ...
   procedure Log (D : Device; P : Pressure);
   pragma Asynchronous (Log);

end Bank;
```

+ returns immediately
+ exceptions are lost
+ parameters must be in


```
package Alerts.Low is
   type Low_Alert is new Alert with private;
   procedure Log (A : access Low_Alert);
private
   ...
end Alerts.Low;
```

```
with Alerts.Pool; use Alerts.Pool;
package body Alerts.Low is
   ...
begin
   Register (new Low_Alert);
end Alerts.Low;
```


Remote_Types

An Example

```
package Alerts.Low is
   pragma Remote_Types;
   type Low_Alert is new Alert with private;
   procedure Log (A : access Low_Alert);
private
   ...
end Alerts.Low;
```

```
package Alerts.Medium is
   pragma Remote_Types;
   type Medium_Alert is new Alert with private;
   procedure Handle (A : access Medium_Alert);
   procedure Log    (A : access Medium_Alert);
private
   ...
end Alerts.Medium;
```



```
package Alerts.Medium is
   type Medium_Alert is new Alert with private;
   procedure Handle (A : access Medium_Alert);
   procedure Log    (A : access Medium_Alert);
private
   ...
end Alerts.Medium;
```

```
with Alerts.Pool; use Alerts.Pool;
package body Alerts.Medium is
   ...
begin
   Register (new Medium_Alert);
end Alerts.Medium;
```


Build & Test

```
package Alerts is
   pragma Remote_Types;
   type Alert is abstract tagged private;
   type Alert_Ref is access all Alert'Class;
   procedure Handle (A : access Alert);
   procedure Log    (A : access Alert) is abstract;
private
   ...
end Alerts;
```

```
package Alerts.Low is
   pragma Remote_Types;
   type Low_Alert is new Alert with private;
   procedure Log (A : access Low_Alert);
private
   ...
end Alerts.Low;
```

```
package Alerts.Medium is
   pragma Remote_Types;
   type Medium_Alert is new Alert with private;
   procedure Handle (A : access Medium_Alert);
   procedure Log    (A : access Medium_Alert);
private
   ...
end Alerts.Medium;
```

```
package Alerts.Pool is
   pragma Remote_Call_Interface;
   procedure Register (A : Alert_Ref);
   function  Get_Alert return Alert_Ref;
end Medium;
```

```
with Alerts, Alerts.Pool; use Alerts;
procedure Process_Alerts is
begin
   loop
      Handle (Pool.Get_Alert);
   end loop;
end Process_Alerts;
```

Categorize

```
package Alerts is
   pragma Remote_Types;
   type Alert is abstract tagged private;
   type Alert_Ref is access all Alert'Class;
   procedure Handle (A : access Alert);
   procedure Log    (A : access Alert) is abstract;
private
```

```
package Alerts.Pool is
   pragma Remote_Call_Interface;
   procedure Register (A : Alert_Ref);
   function  Get_Alert return Alert_Ref;
end Medium;
```

```
with Alerts, Alerts.Pool; use Alerts;
procedure Process_Alerts is
begin
   loop
      Handle (Pool.Get_Alert);
   end loop;
end Process_Alerts;
```

Partition

```
configuration Config_2 is
   Node_AL  : Partition := (Alerts.Low);
   Node_AM : Partition := (Alerts.Medium);
   Node_B   : Partition := (Alerts.Pool);
   Node_C   : Partition := (Process_Alerts);
end Config_2;
```


What Happens When Executing the Distributed Program ?


```
package Alerts.Low is
   pragma Remote_Types;
   type Low_Alert is new Alert with private;
   procedure Log (A : access Low_Alert);
private
   ...
end Alerts.Low;
```

Node_AL

```
package Alerts.Medium is
   pragma Remote_Types;
   type Medium_Alert is new Alert with private;
   procedure Handle (A : access Medium_Alert);
   procedure Log    (A : access Medium_Alert);
private
   ...
end Alerts.Medium;
```

Node_AM

Step 1: A Low_Alert object in Node_AL registers itself with Node_B

```
package Alerts.Pool is
   pragma Remote_Call_Interface;
   procedure Register (A : Alert_Ref);
   function  Get_Alert return Alert_Ref;
end Medium;
```

Node_B

```
with Alerts, Alerts.Pool; use Alerts;
procedure Process_Alerts is
begin
   loop
      Handle (Pool.Get_Alert);
   end loop;
end Process_Alerts;
```

Node_C


```
package Alerts.Low is
   pragma Remote_Types;
   type Low_Alert is new Alert with private;
   procedure Log (A : access Low_Alert);
private
   ...
end Alerts.Low;
```

Node_AL

```
package Alerts.Medium is
   pragma Remote_Types;
   type Medium_Alert is new Alert with private;
   procedure Handle (A : access Medium_Alert);
   procedure Log    (A : access Medium_Alert);
private
   ...
end Alerts.Medium;
```

Node_AM

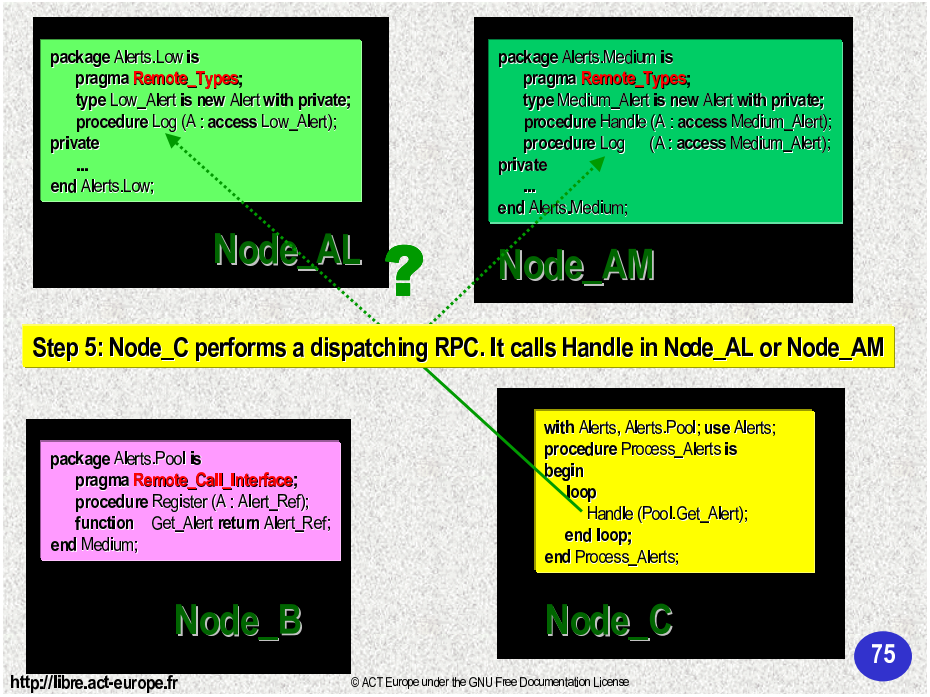
Step 2: A Medium_Alert object in Node_AM registers itself with Node_B

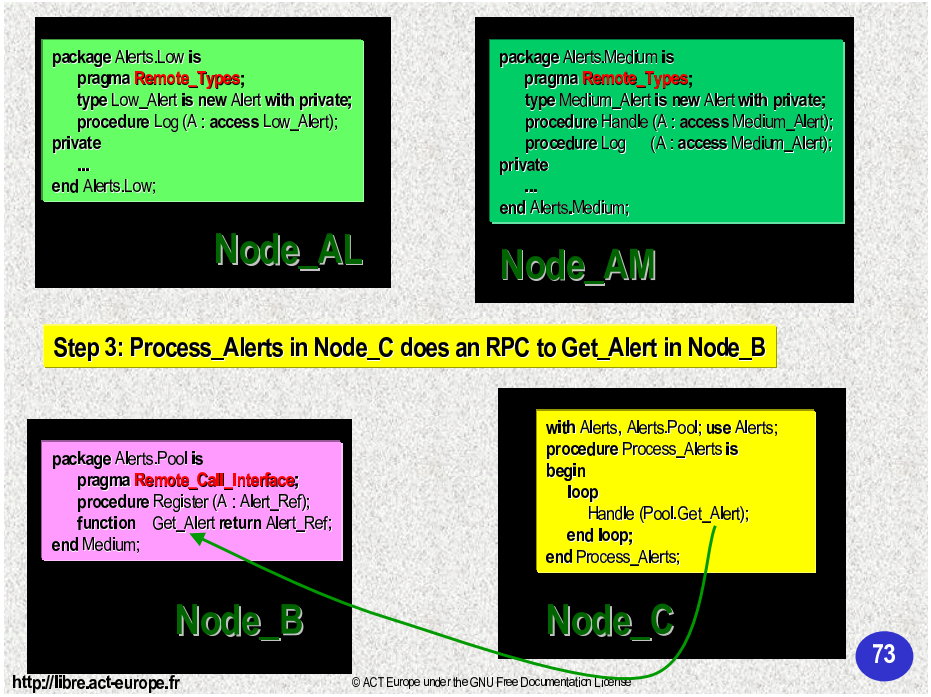

```
package Alerts.Pool is
   pragma Remote_Call_Interface;
   procedure Register (A : Alert_Ref);
   function  Get_Alert return Alert_Ref;
end Medium;
```

Node_B

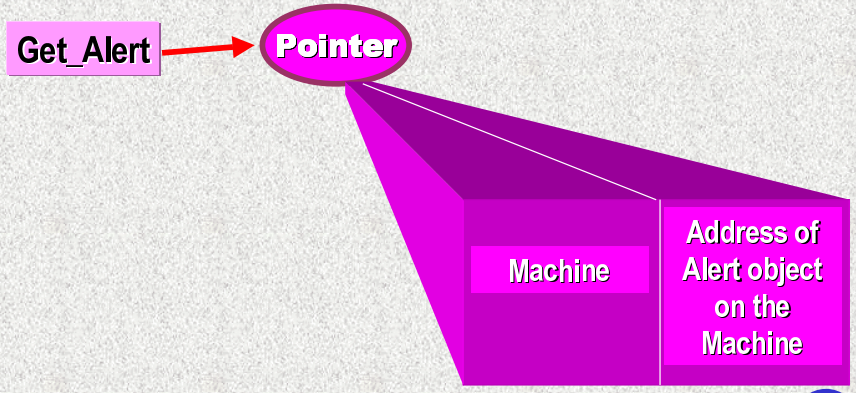
```
with Alerts, Alerts.Pool; use Alerts;
procedure Process_Alerts is
begin
   loop
      Handle (Pool.Get_Alert);
   end loop;
end Process_Alerts;
```

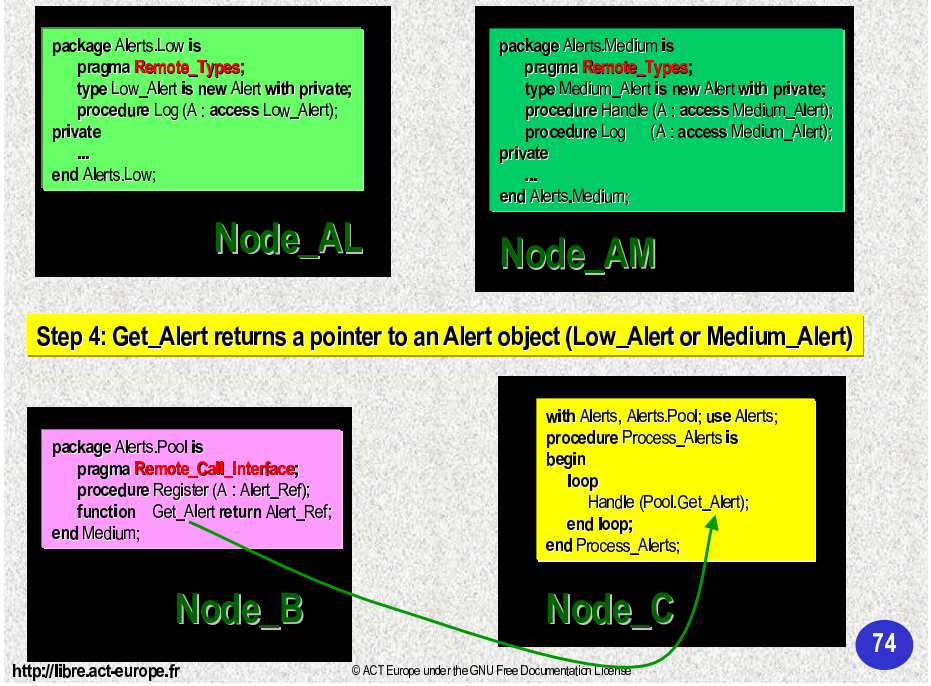
Node_C





What Does Get_Alert Return ?





Remote Access to Class Wide Type

- At compile time:
  - You do not know what operation you'll dispatch to
  - On what node that operations will be executed on