Ada 95 Distributed Programming

- Introduction
- Distributed Prog. Paradigms
  - Language Dependent: Ada 95
  - Language Independent: CORBA

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)
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 partitioning your app.
5. Build partitions & test distributed app.

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

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);

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);
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;
end Medium;

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

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

with Alerts, Alerts.Pool; use Alerts;
procedure Process Alerts is
begin
  loop
    Handle (Pool, Get.Alert);
  end loop;
end Process Alerts;

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

Remote_Types

An Example
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?
Step 5: Node_C performs a dispatching RPC. It calls Handle in Node_AL or Node_AM

What Does Get_Alert Return?

Step 4: Get_Alert returns a pointer to an Alert object (Low_Alert or Medium_Alert)
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