SOFTWARE DEVELOPMENT

Franco Gasperoni

gasperon@act-europe.fr

http://libre.act-europe.fr

© ACT Europe under the GNU Free Documentation License

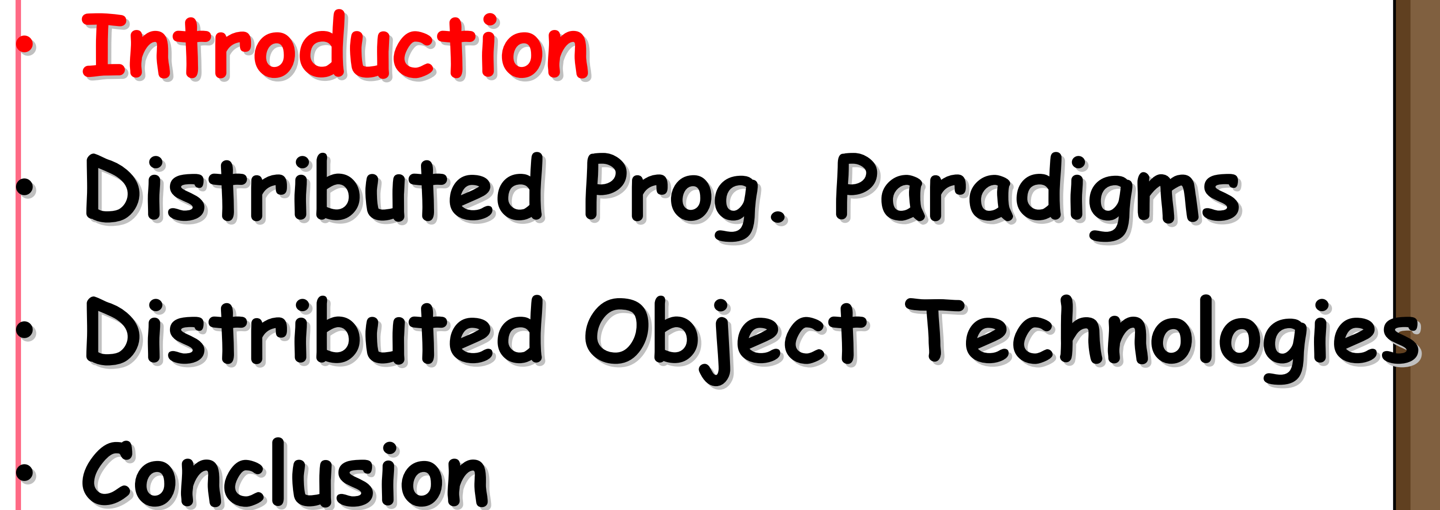
Copyright Notice

- © ACT Europe under the GNU Free Documentation License

- Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; provided its original author is mentioned and the link to http://libre.act-europe.fr/ is kept at the bottom of every non-title slide. A copy of the license is available at:
- http://www.fsf.org/licenses/fdl.html

Programming Distributed Systems

© ACT Europe under the GNU Free Documentation License



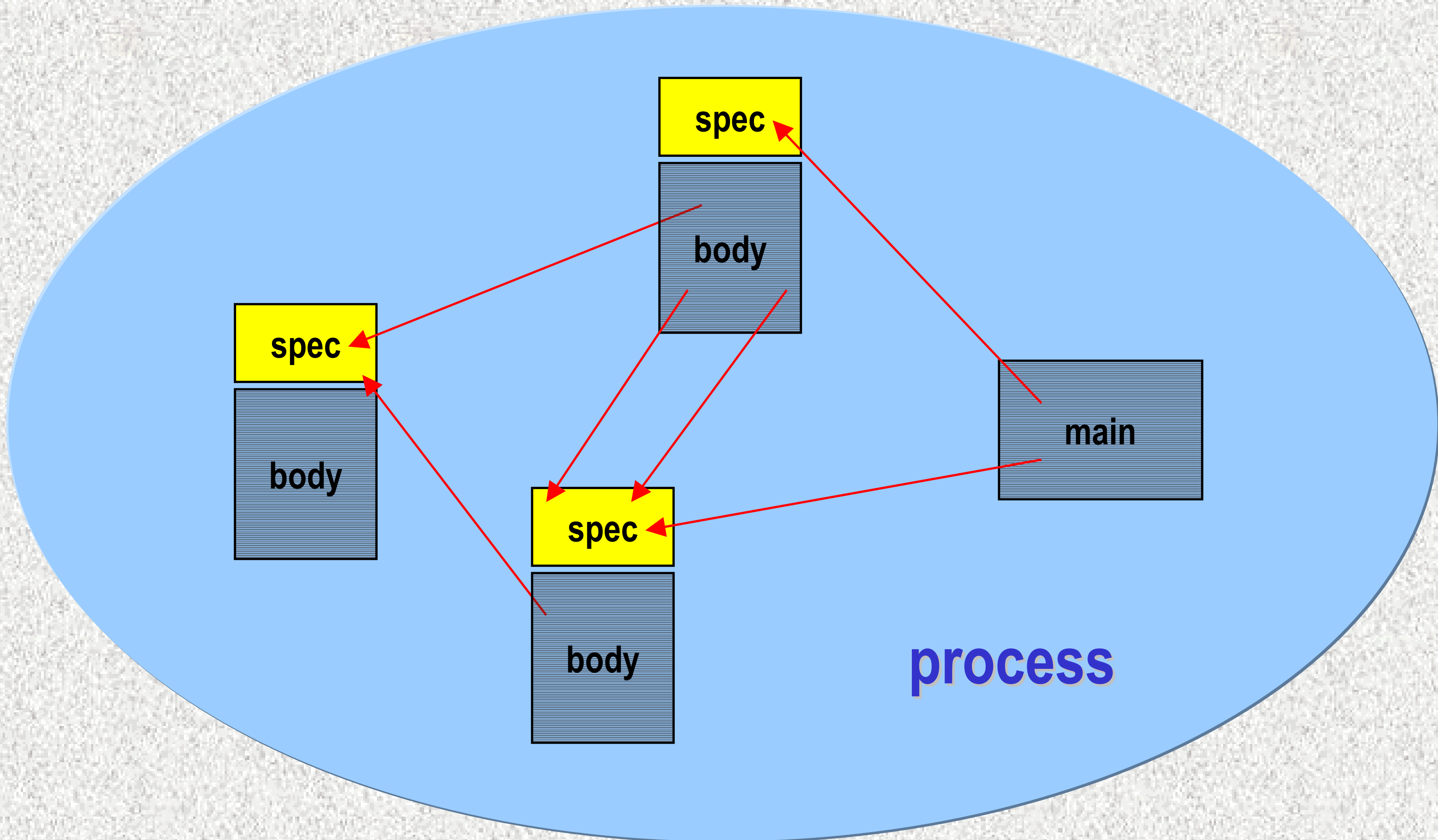
© ACT Europe under the GNU Free Documentation License

- **Non-distributed application** = single process
  - running on a single computer

- **Distributed application** = several communicating processes
  - processes often run on different computers
  - computers are connected through a network

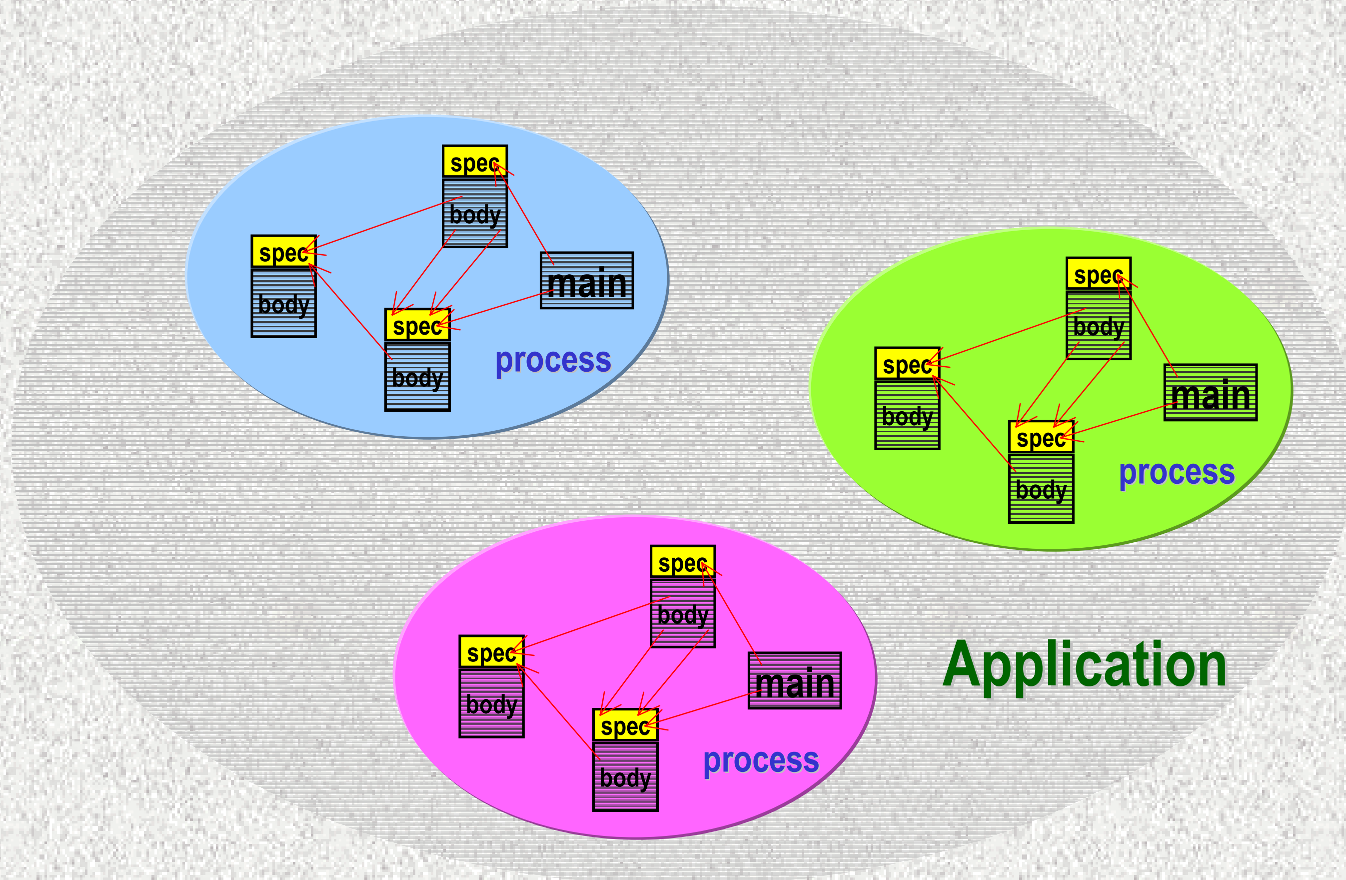
© ACT Europe under the GNU Free Documentation License

Single Process Application



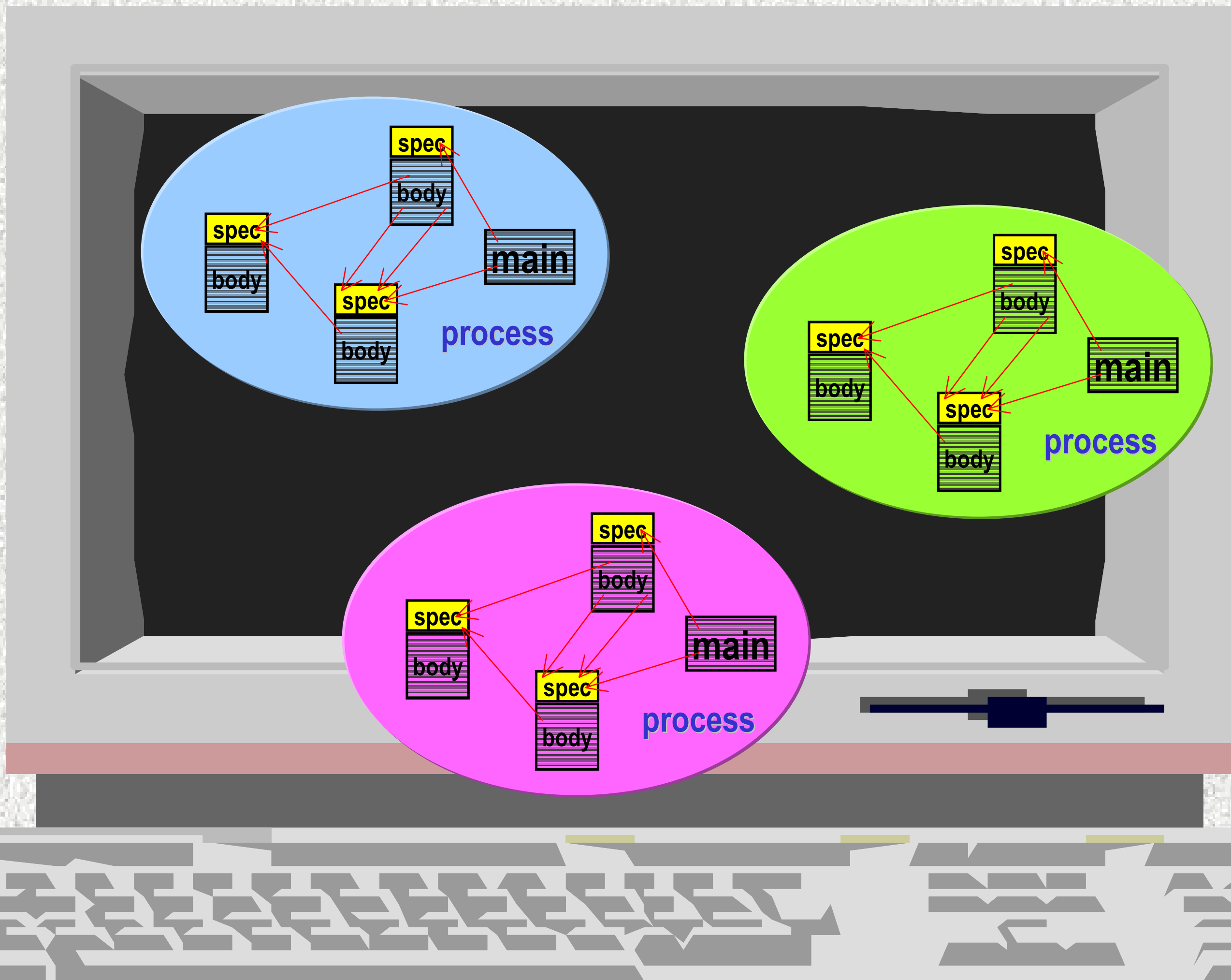
© ACT Europe under the GNU Free Documentation License

Distributed Application

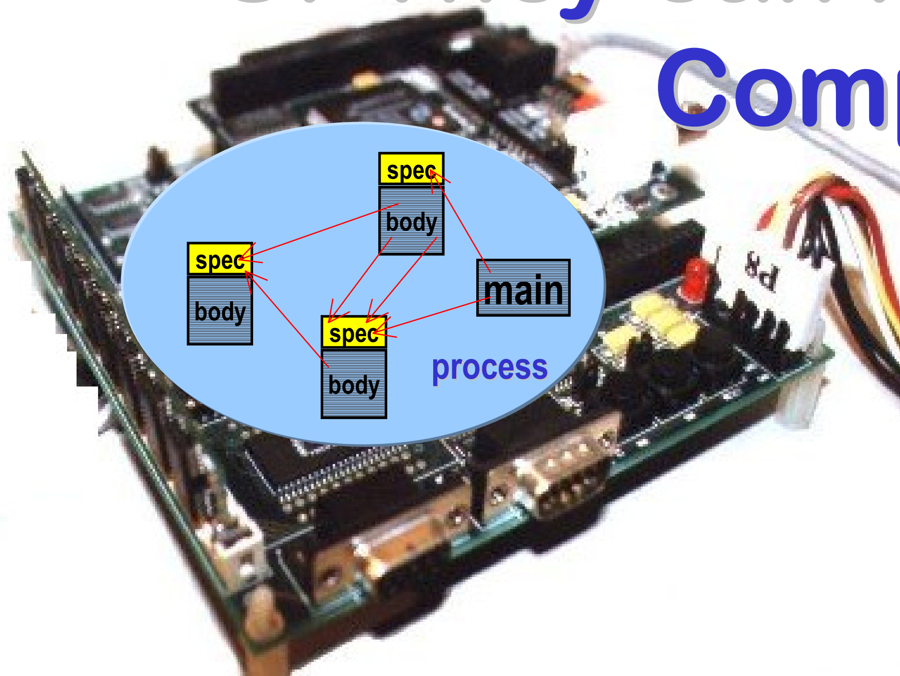


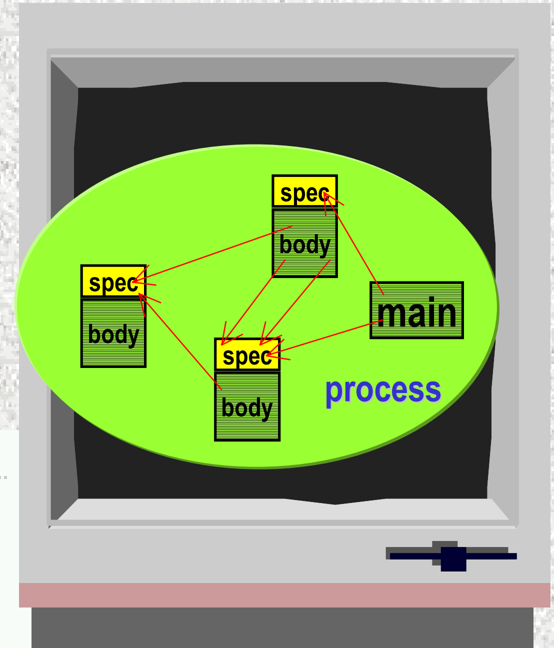
© ACT Europe under the GNU Free Documentation License

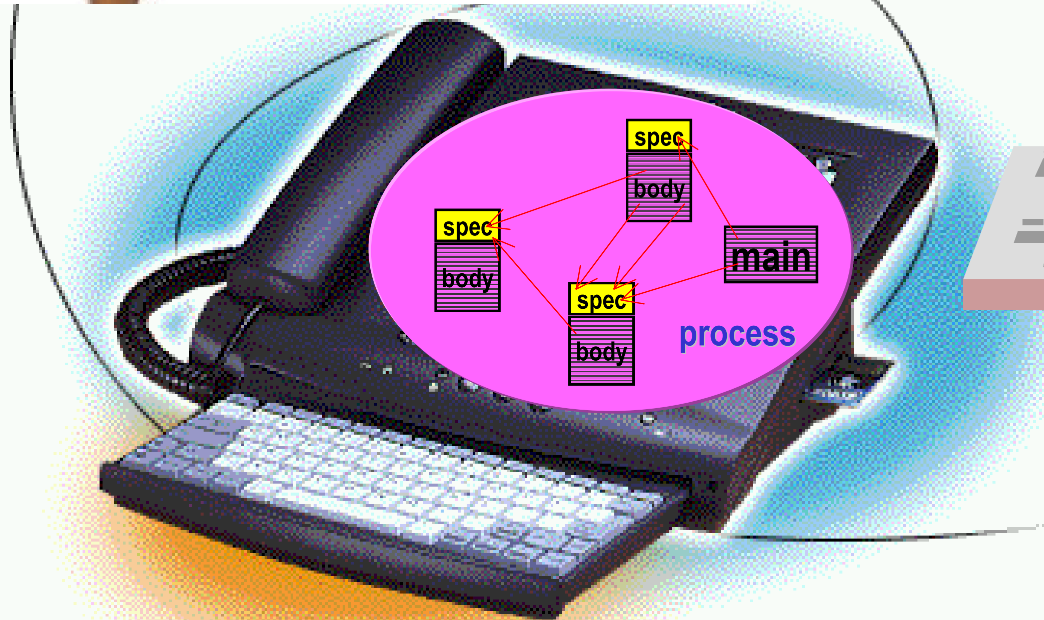
All Processes Can Run on the Same Computer



Or They can run on Different Computers

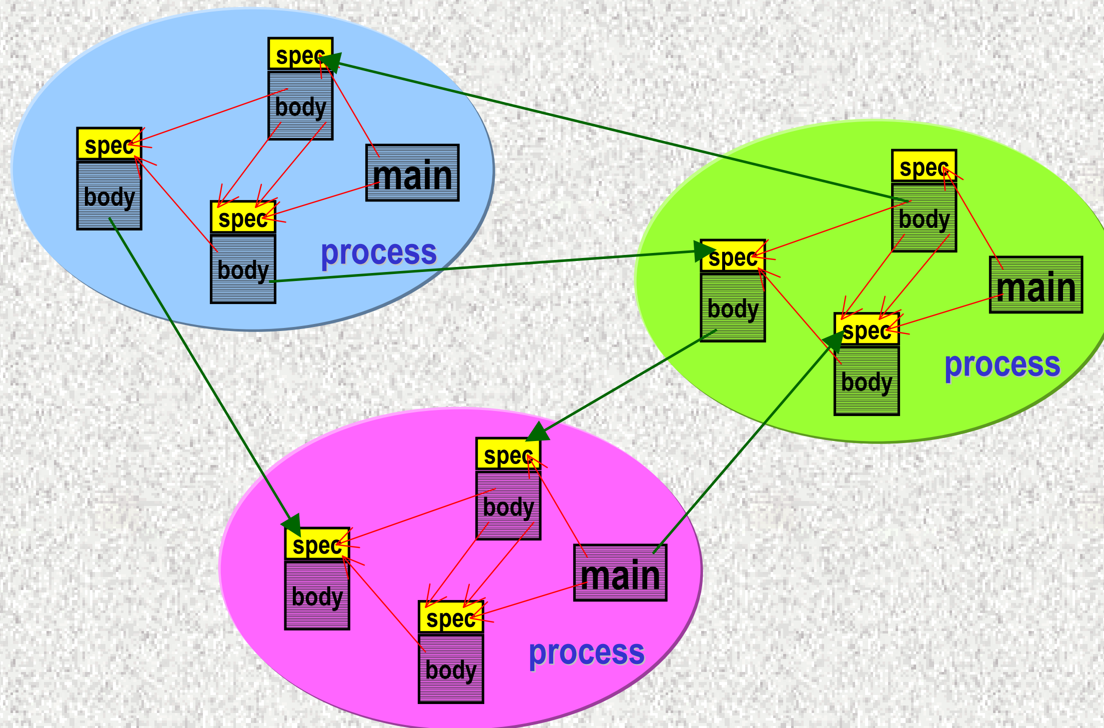




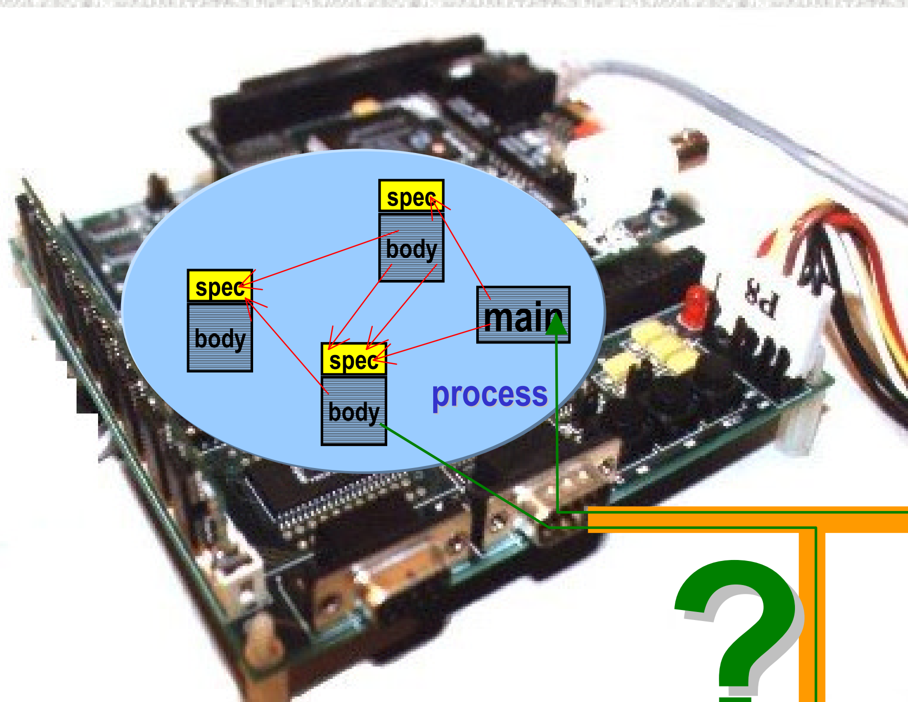


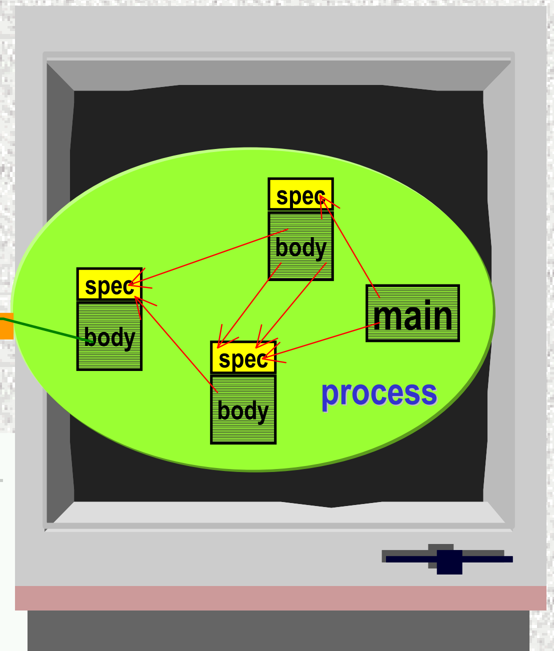
© ACT Europe under the GNU Free Documentation License

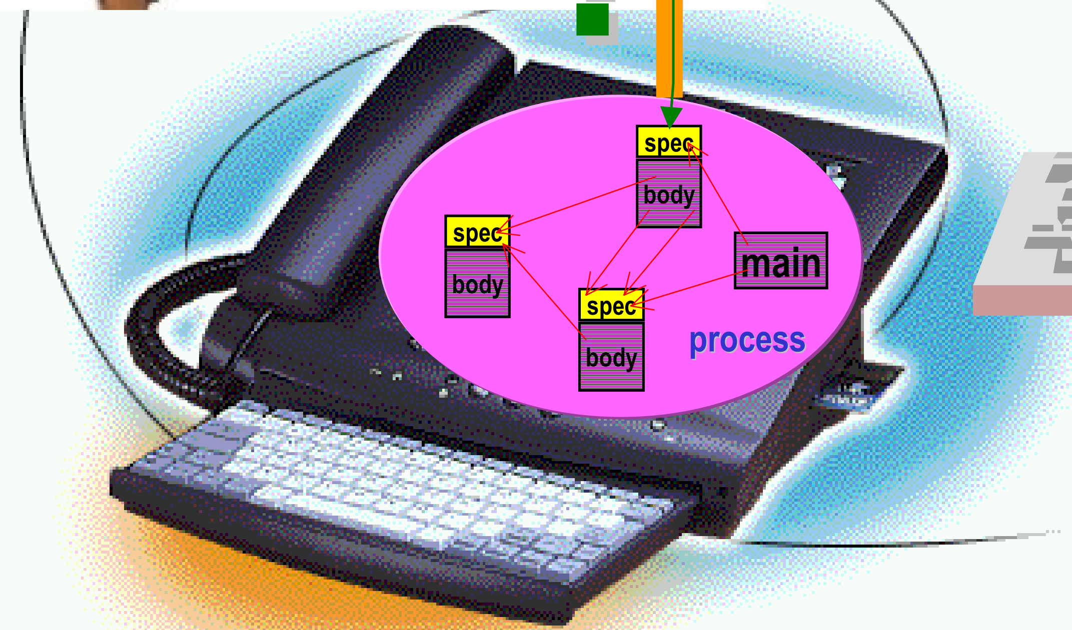
In All Cases This Requires Inter-Process Communication



© ACT Europe under the GNU Free Documentation License







?
?

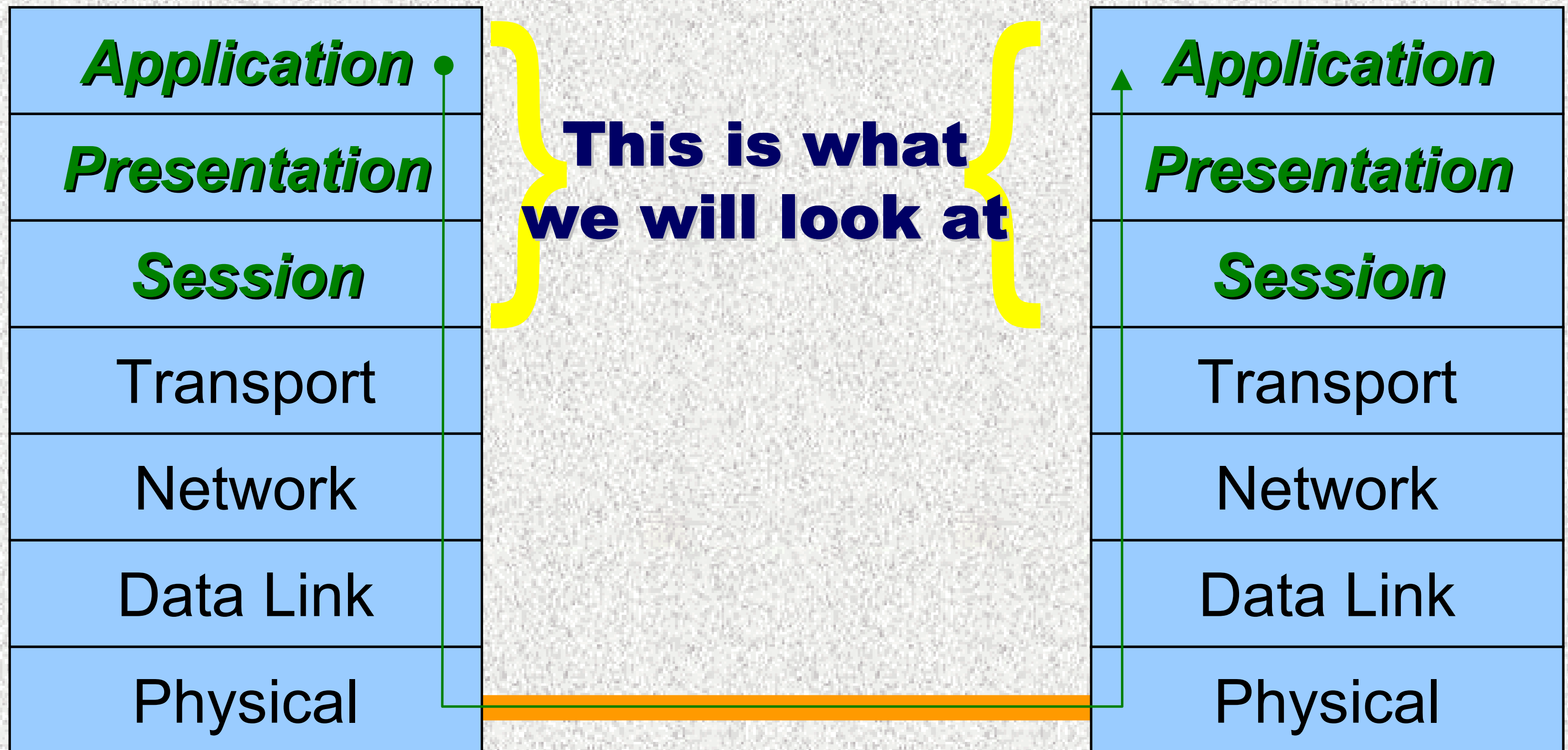
© ACT Europe under the GNU Free Documentation License

The Main Topic of this Lecture

- **How distributed processes communicate at the *programming* level**
- **How the "*software chunks*" of a distributed app can *interact*.**
- **This lecture will NOT teach you how they communicate at the**
  - **physical level**
  - **or protocol level**

© ACT Europe under the GNU Free Documentation License

Remember OSI Layers ?



© ACT Europe under the GNU Free Documentation License

Application	***Telnet, ftp, …***
Presentation	***Sending data in platform indep. manner***
Session	***Establish communication bw processes***
Transport	TCP, UDP, ...
Network	IP, X.25, ...
Data Link	Network drivers
Physical	The wire

© ACT Europe under the GNU Free Documentation License

Why Distributed Apps ?

- Multiuser apps (e.g. e-mail, ftp)
- Sharing data (e.g. www, airline reservation)
- Sharing resources (e.g. printers)
- Fault tolerance
- App may be inherently distributed (cell phones, ATM machines, …)

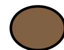
© ACT Europe under the GNU Free Documentation License

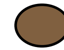
Distributed Prog. Is Hard

- Multiple failure modes
  - each individual process can fail (bugs, machine crash..)
  - the network can go ashtray
- Security issues
  - is someone else listening
- Testing & debugging
- Distributed prog. technologies not fully mature
  - interoperability is still an issue

© ACT Europe under the GNU Free Documentation License

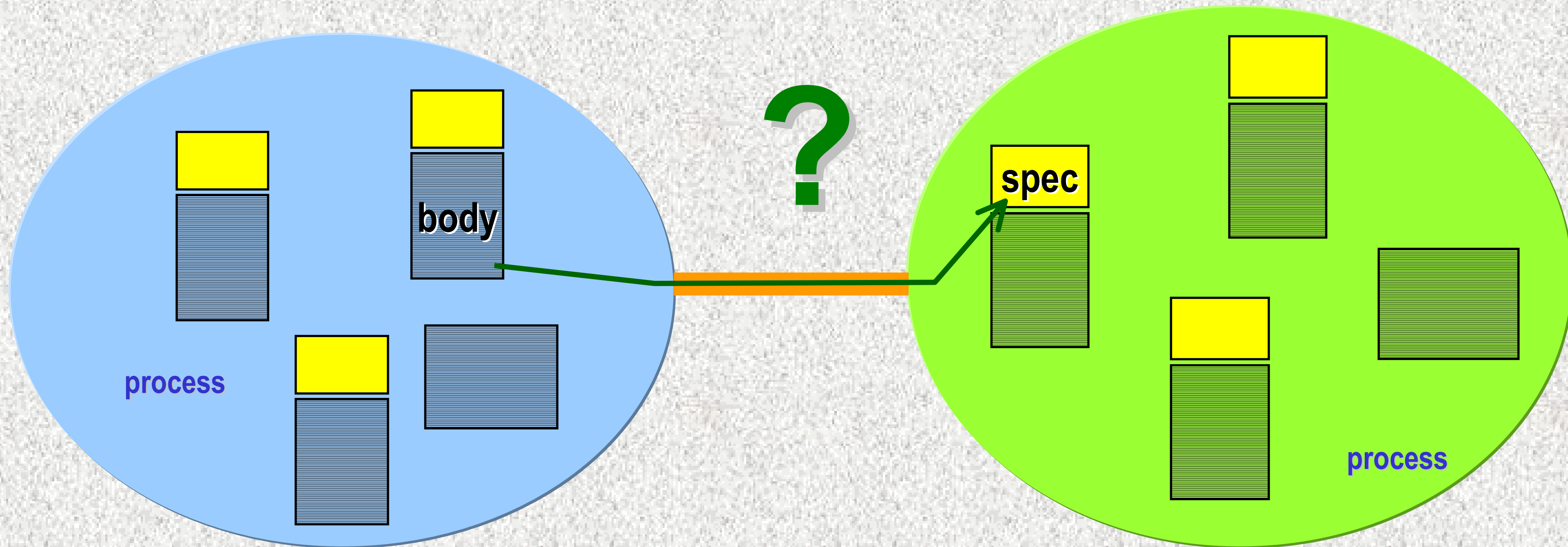
- Introduction
- Distributed Prog. Paradigms
  - Message Sending (Sockets)
  - Remote Procedure Calls
  - Distributed Objects





© ACT Europe under the GNU Free Documentation License

How to Formalize the Notion of an Interface in a Distributed Environment ?



© ACT Europe under the GNU Free Documentation License

How to Formalize the Notion of an Interface in a Distributed Environment ?

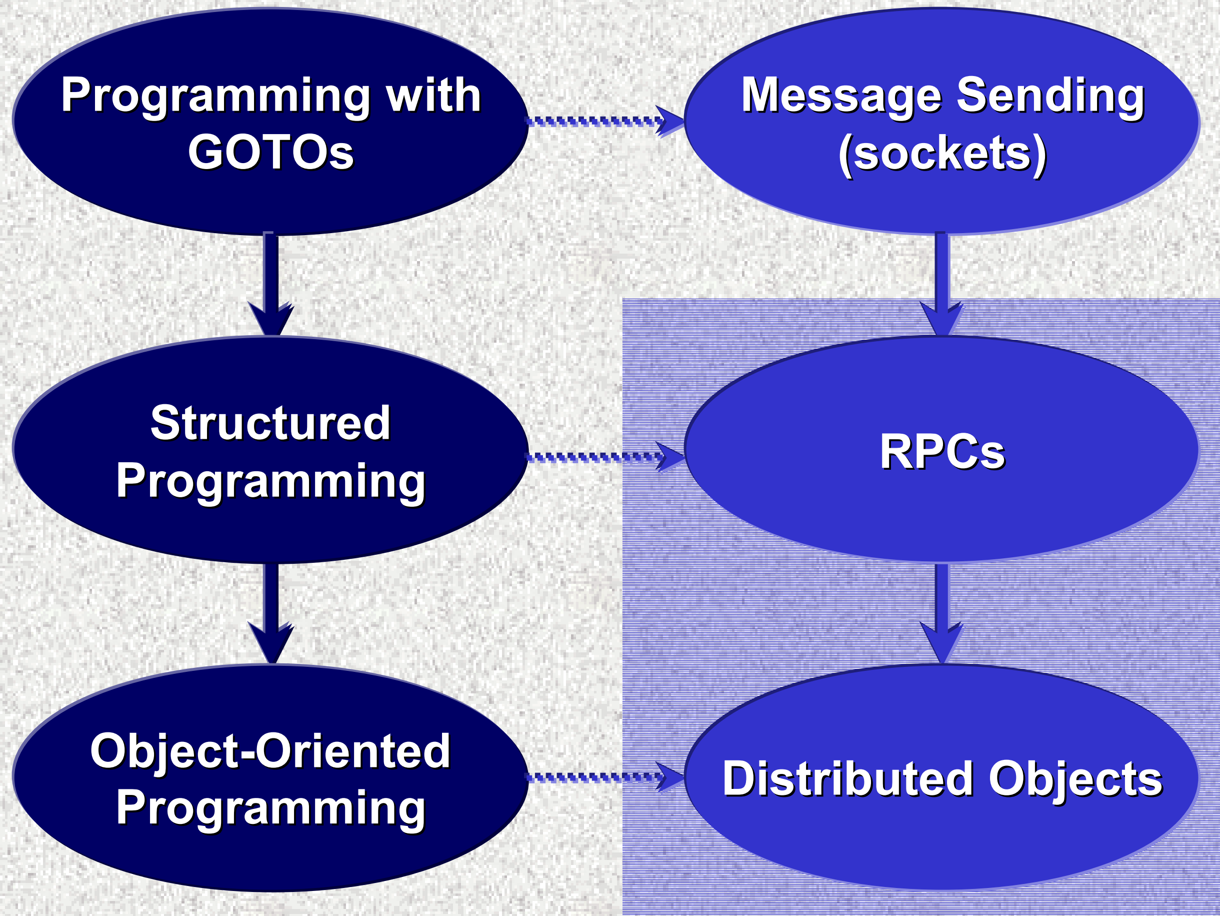
- **Answer 1**: don't formalize it, send a message
  - e.g. sockets

- **Answer 2**: Remote Procedure Call (RPC)

- **Answer 3**: RPCs + Distributed Objects
  - Language dependent: Ada 95, Java RMI
  - Language independent: CORBA, COM/DCOM

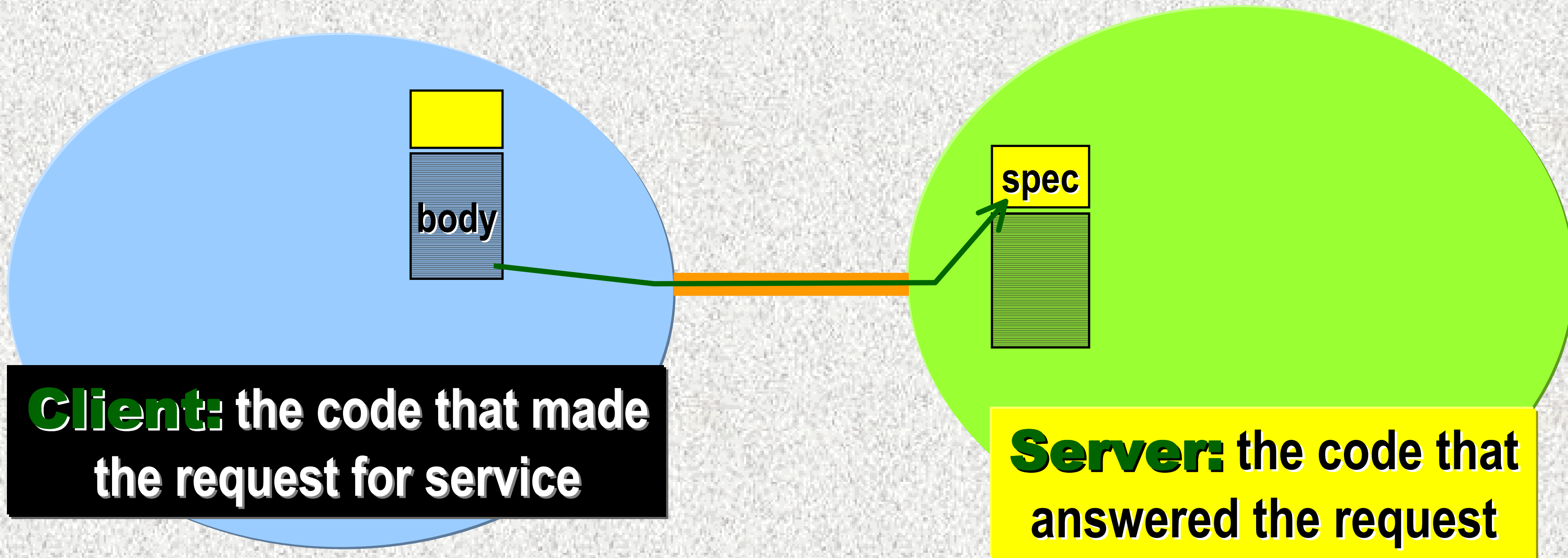
© ACT Europe under the GNU Free Documentation License

A Simple Comparison



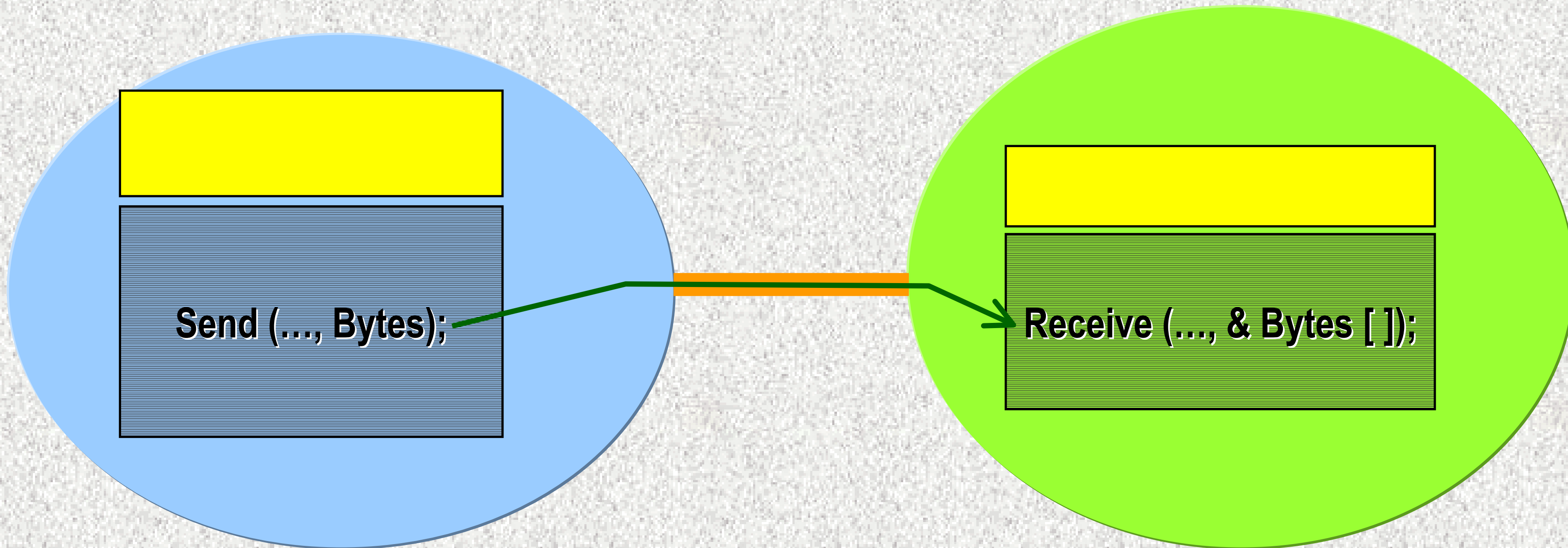
© ACT Europe under the GNU Free Documentation License

Some Terminology



© ACT Europe under the GNU Free Documentation License

Answer 1: Don't Formalize It
Send a Message (e.g. Sockets)



© ACT Europe under the GNU Free Documentation License

Client Process

Server Process

Open socket

Open socket

Wait for connection

Compute

raw bytes

Send bytes

Get bytes

Wait for reply

Compute

Send bytes

Get bytes

raw bytes

Close socket

Close socket

time

time

© ACT Europe under the GNU Free Documentation License

What are the problems

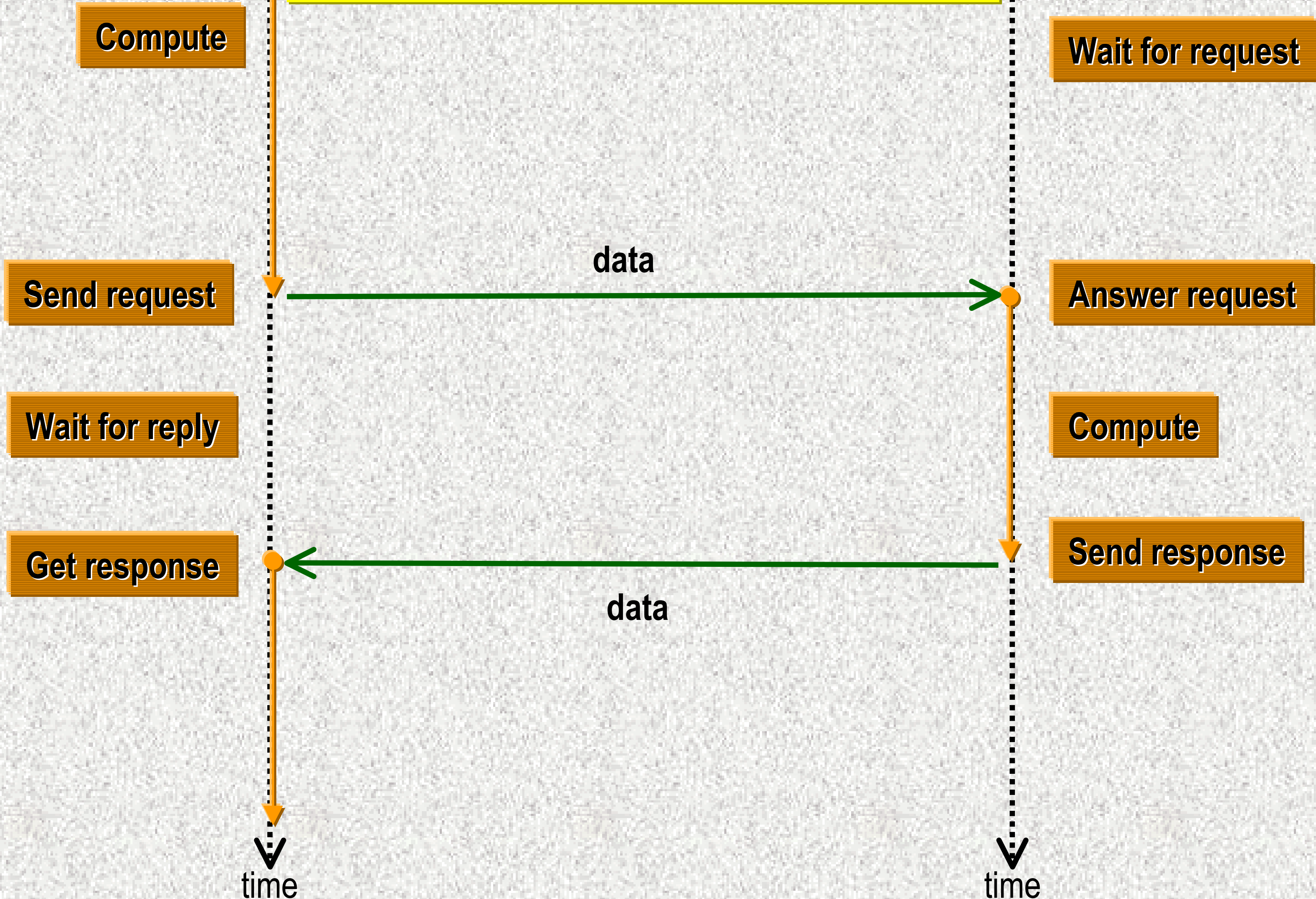
with this approach ?

© ACT Europe under the GNU Free Documentation License

Problems with Sockets/Message Sending

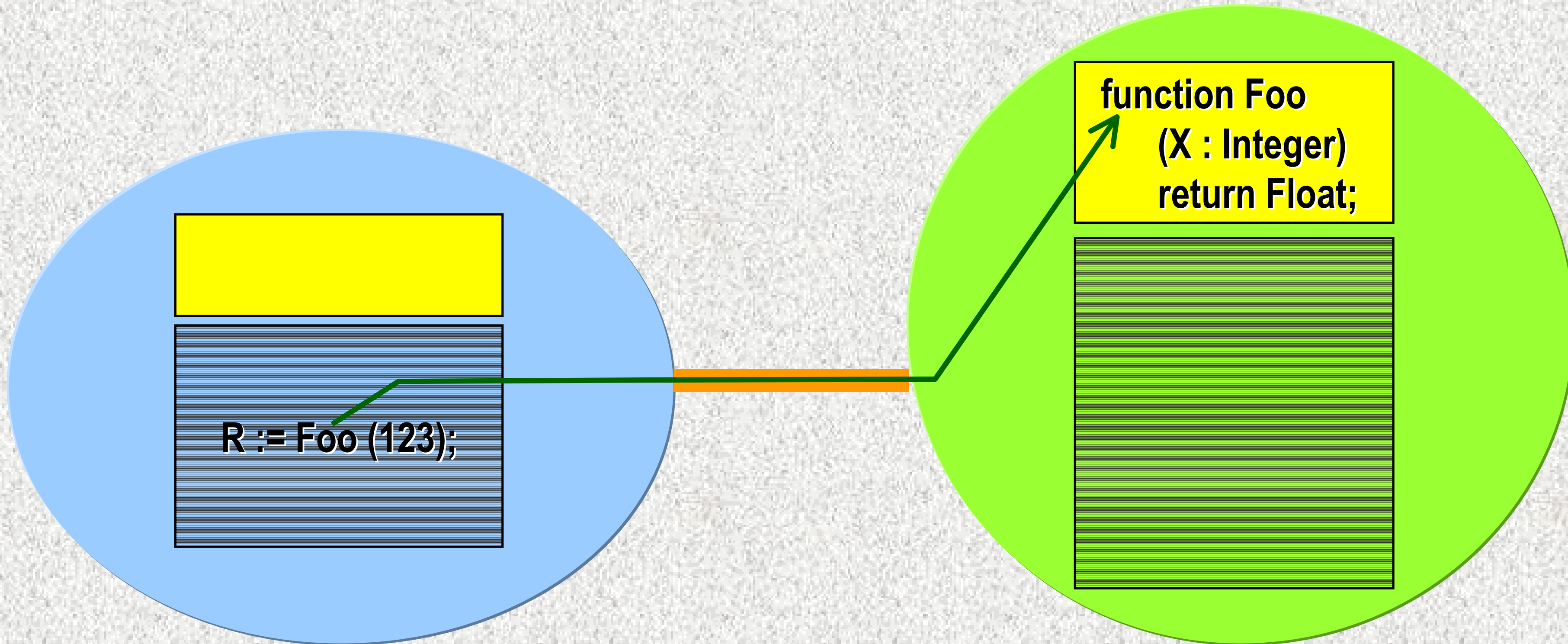
- No interfaces - very low-level programming
  - does not scale up
- Sockets exchange bytes
  - How do you exchange more complex data structures ?
  - How do you handle heterogeneous systems ?

© ACT Europe under the GNU Free Documentation License



© ACT Europe under the GNU Free Documentation License

Answer 2: Remote Procedure Calls



© ACT Europe under the GNU Free Documentation License

RPC

Client

Server

Compute

R := Foo (123);

send parameters

```
function Foo (X: Integer)
   return Float
is
 ...
begin
   ...
   return ...;
end Foo;
```

Wait for reply

Get result

send result or modified parameters

time

time

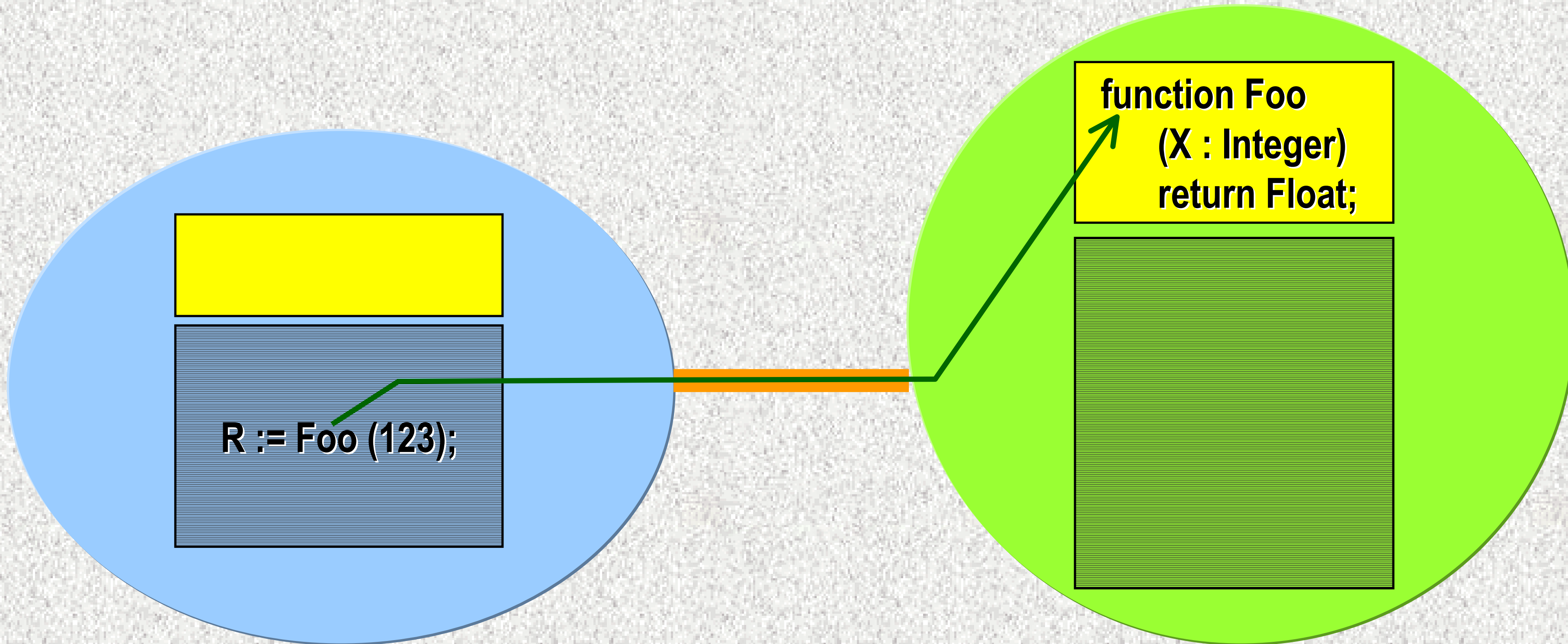
© ACT Europe under the GNU Free Documentation License

RPCs

- Remote procedure call completely handled by the system
- Parameters and results passed across the network without programmer intervention
- Heterogeneity handled transparently

© ACT Europe under the GNU Free Documentation License

Where is the Magic ?



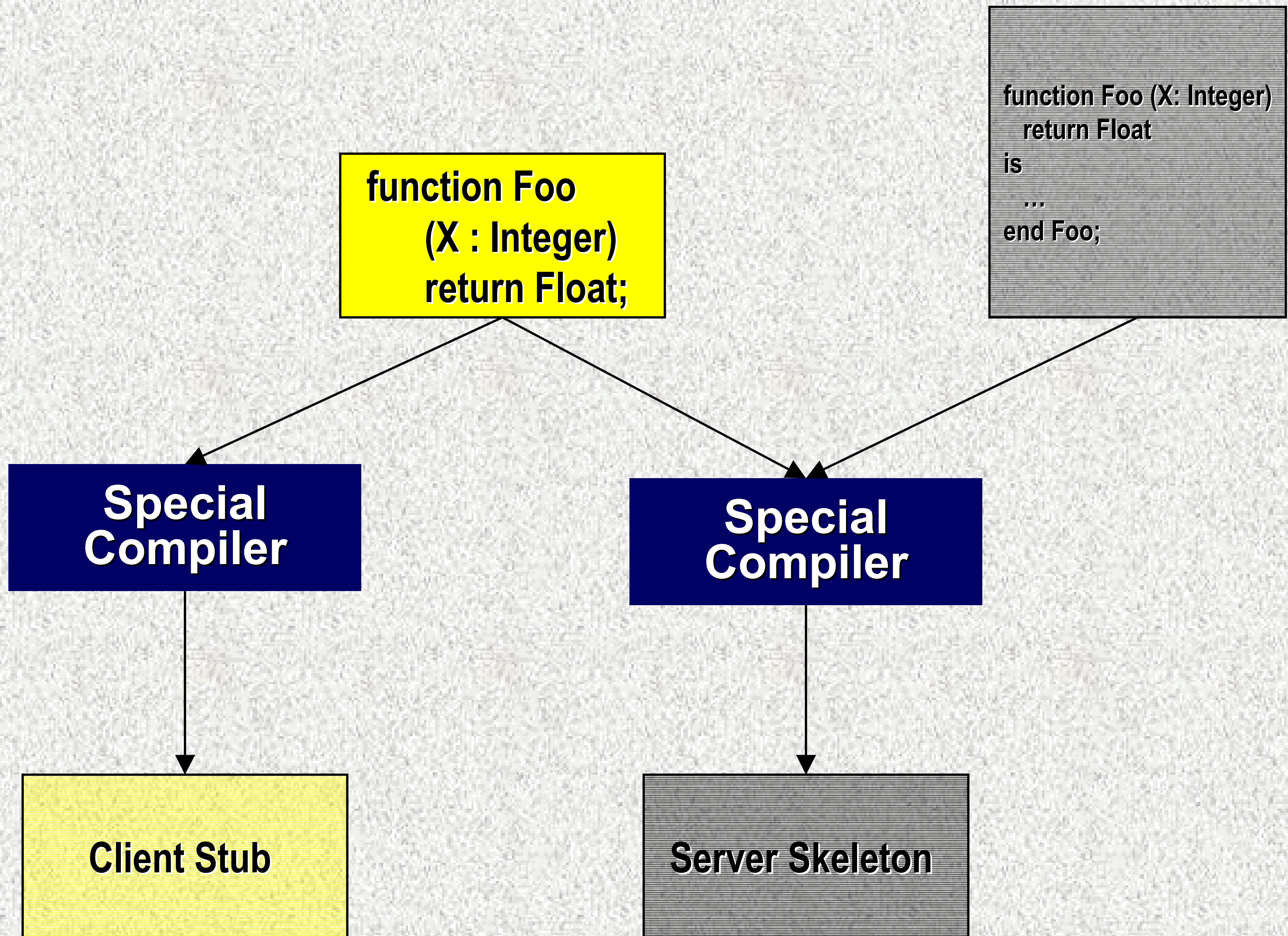
© ACT Europe under the GNU Free Documentation License

Client Stub & Server Skeleton

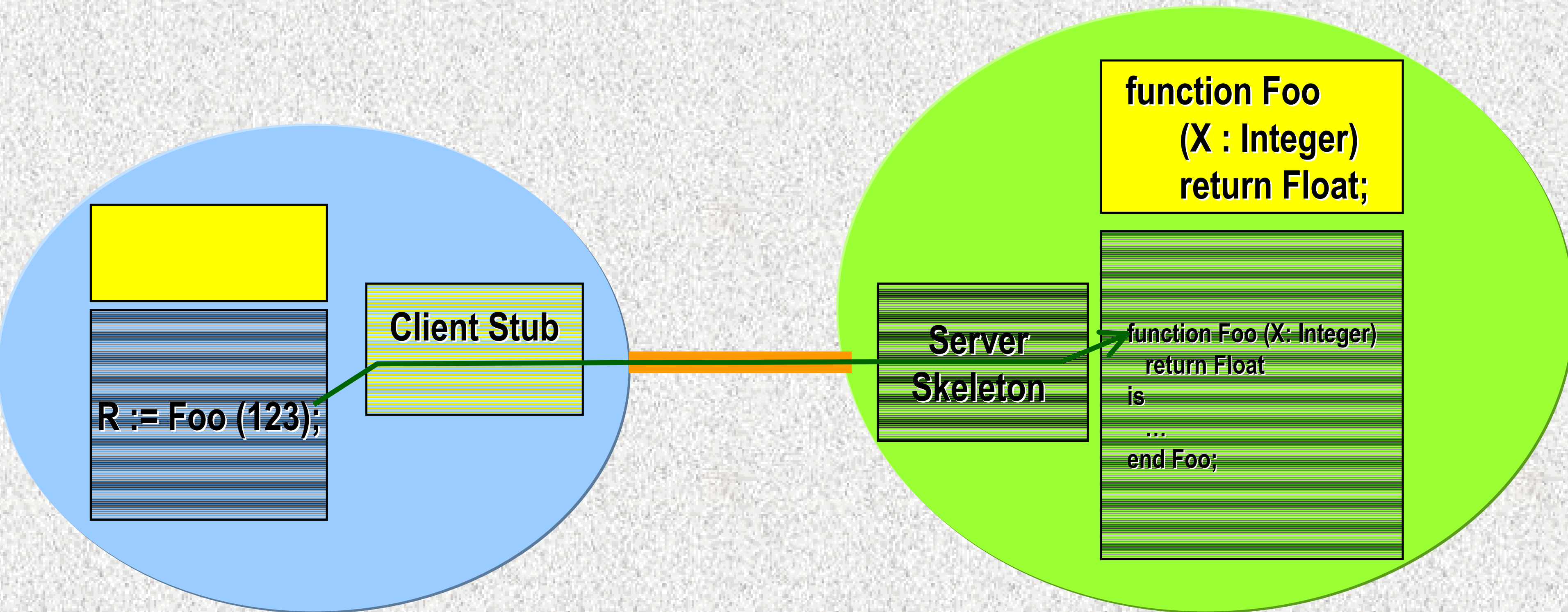
- From the server spec the system generates a client stub:
  - Marshals the parameters
  - Sends the request over the network
  - Waits for the response and unmarshals the result
- From the server spec (and server body) the system generates a server skeleton
  - Receives the RPC request
  - unmarshals the parameters
  - Selects and calls the appropriate subprogram
  - Marshals the result and sends the response

© ACT Europe under the GNU Free Documentation License

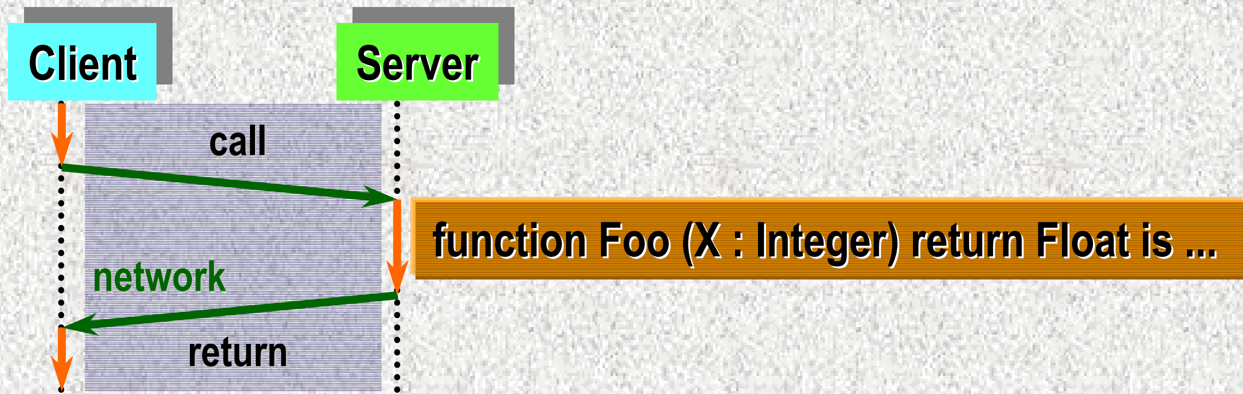
Stubs & Skeletons



© ACT Europe under the GNU Free Documentation License



© ACT Europe under the GNU Free Documentation License



Client
Client Stub
Server Skeleton
Server
call
parameter
marshalling
result
unmarshalling
call
function Foo
network
result
marshalling
return
result
unmarshalling
return

© ACT Europe under the GNU Free Documentation License

Enhancing RPCs

- **Exceptions**
  - exceptions raised in the callee canbe transmitted to the caller over the network
- **Asynchronous calls**
  - the caller does not need to wait for the result from the callee (one way procedure calls)
- **Pointers on remote procedures**
  - RPC through a pointer. At the point of call the spec of the callee is known but not its location or identity

© ACT Europe under the GNU Free Documentation License

Service Related to RPCs: Naming

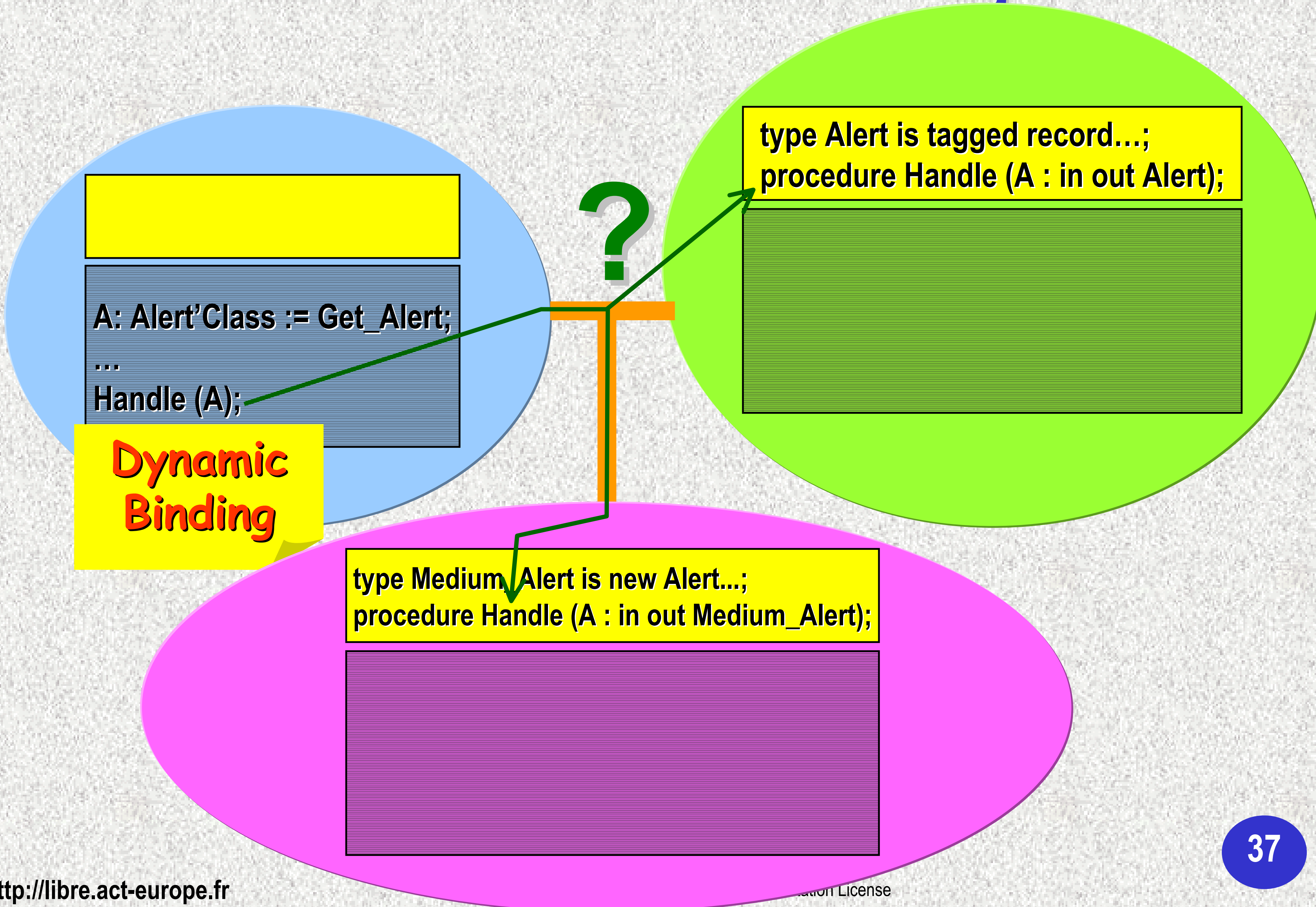
- Records the location of the various processes
  - location of client stubs and server skeletons

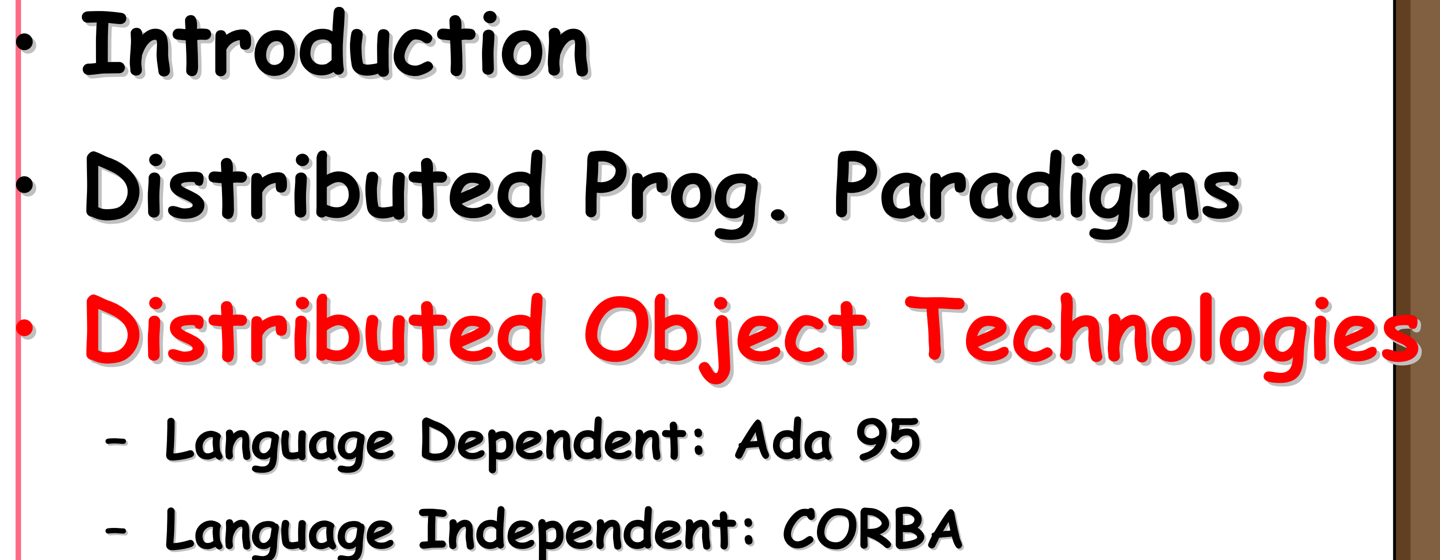
- This service is called via RPC

- To solve the circularity problem the naming service is at a known machine address

© ACT Europe under the GNU Free Documentation License

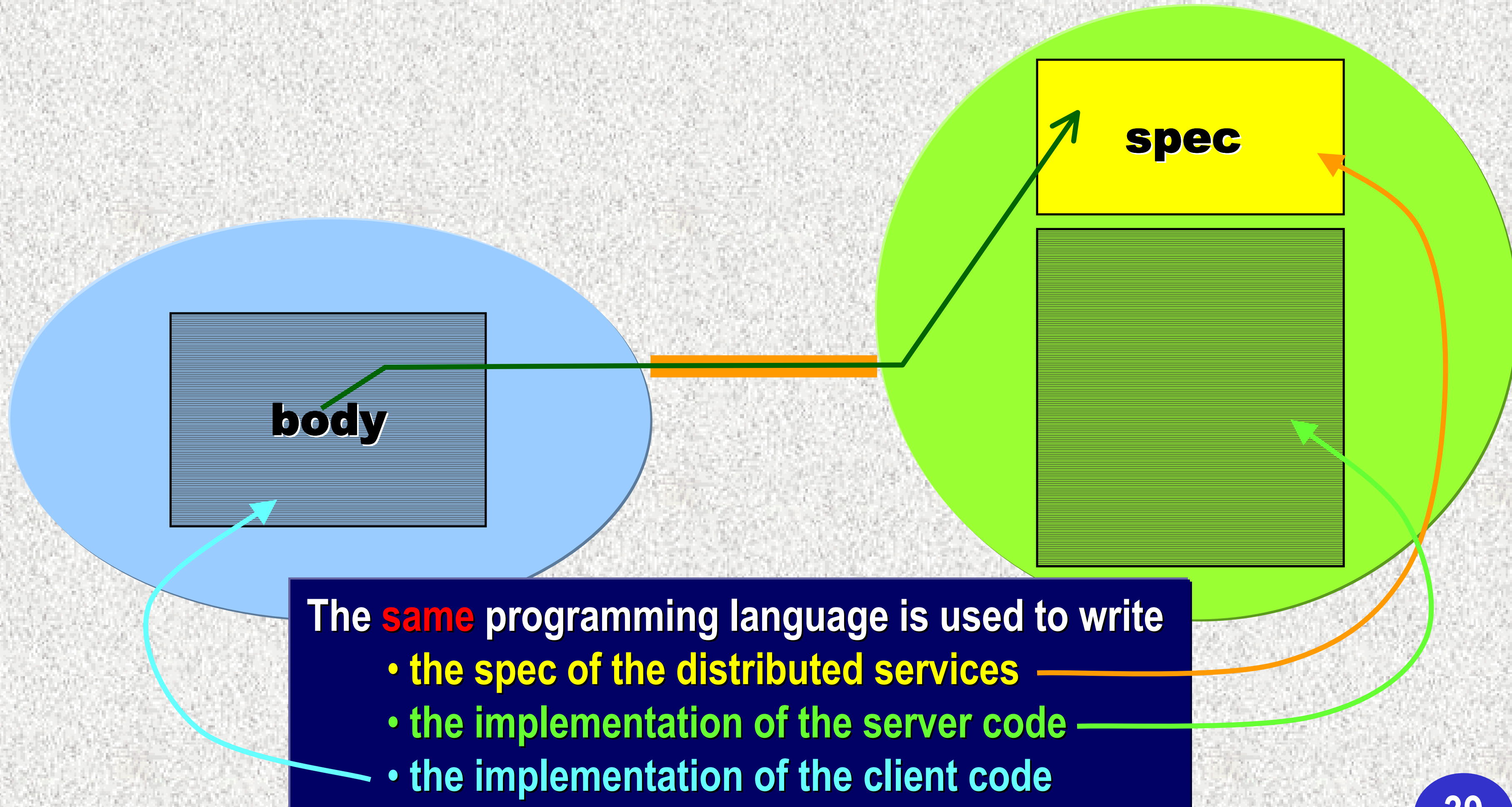
Answer 3: Distributed Objects





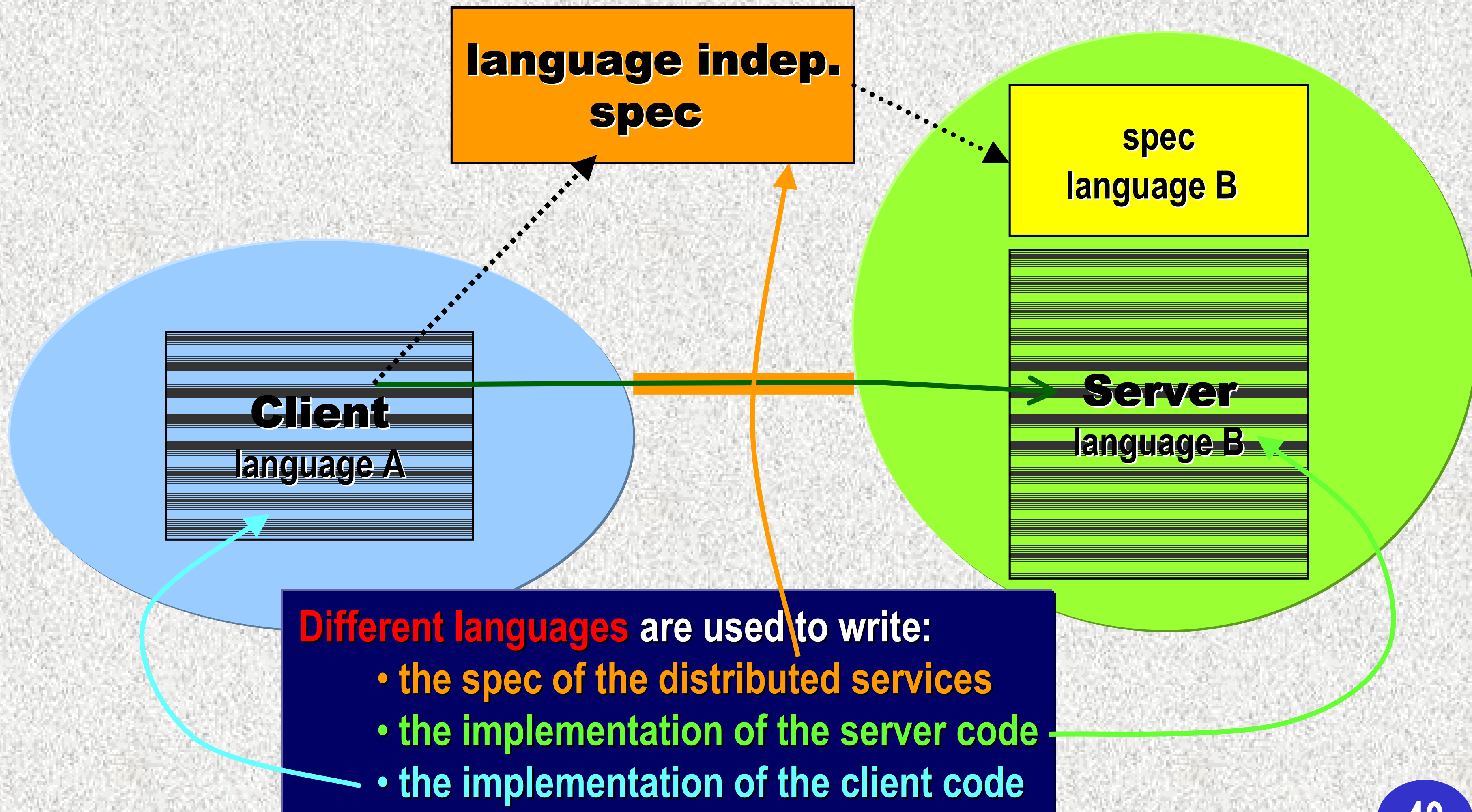
© ACT Europe under the GNU Free Documentation License

Language Dependent Distributed Objects Paradigm

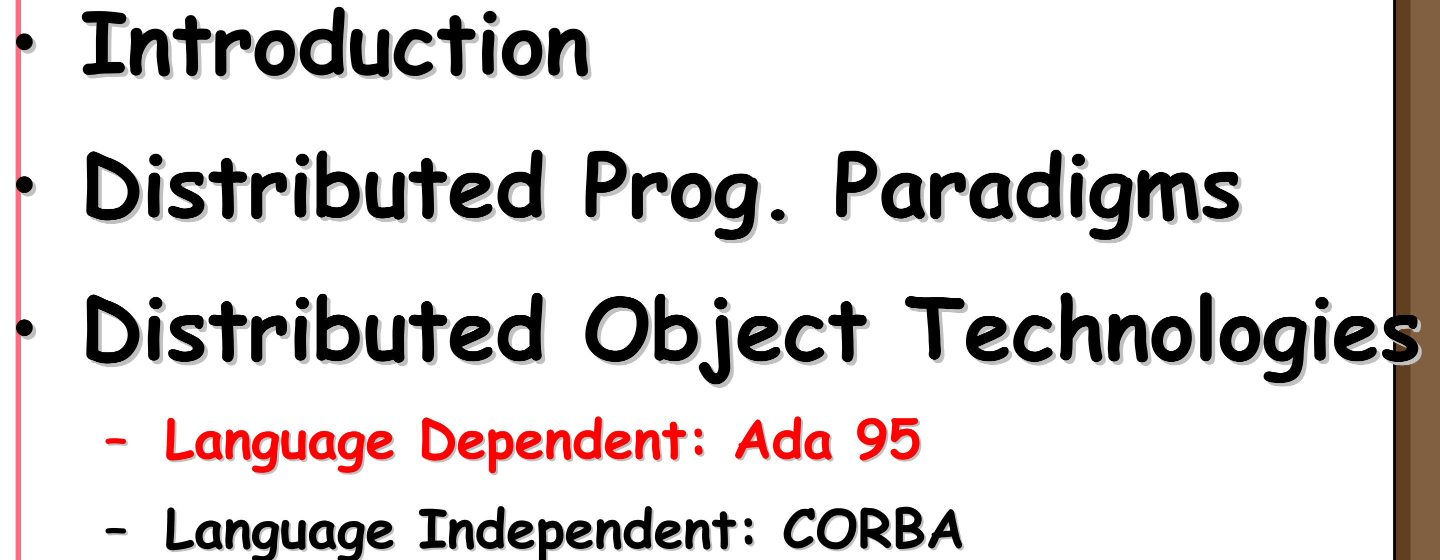


© ACT Europe under the GNU Free Documentation License

Language Independent Distributed Objects Paradigm



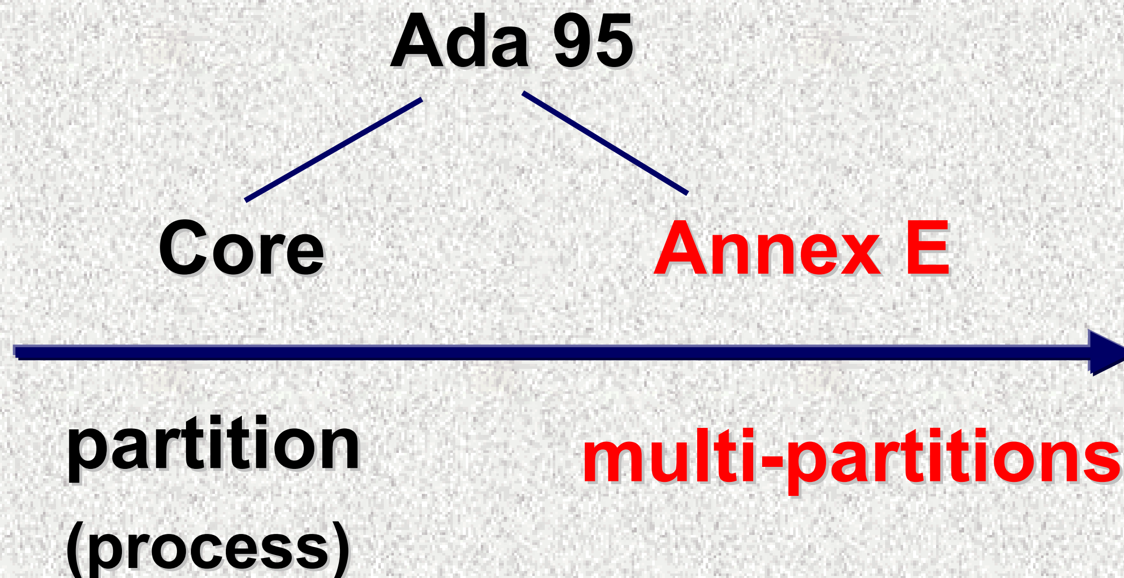
© ACT Europe under the GNU Free Documentation License



© ACT Europe under the GNU Free Documentation License

Ada 95
Distributed Systems Annex

Ada 95 Distributed Programming



A partition comprises one or more Ada packages

© ACT Europe under the GNU Free Documentation License

Supported Paradigms

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

© ACT Europe under the GNU Free Documentation License

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

© ACT Europe under the GNU Free Documentation License

Remote_Call_Interface (RCI)

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

© ACT Europe under the GNU Free Documentation License

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)

© ACT Europe under the GNU Free Documentation License

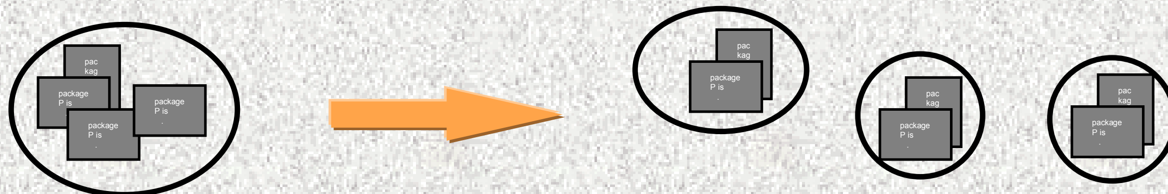
Shared_Passive

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

© ACT Europe under the GNU Free Documentation License

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.



© ACT Europe under the GNU Free Documentation License

Remote_Call_Interface

An Example

© ACT Europe under the GNU Free Documentation License

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);
```


© ACT Europe under the GNU Free Documentation License

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);
```


© ACT Europe under the GNU Free Documentation License

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);
```

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;
```

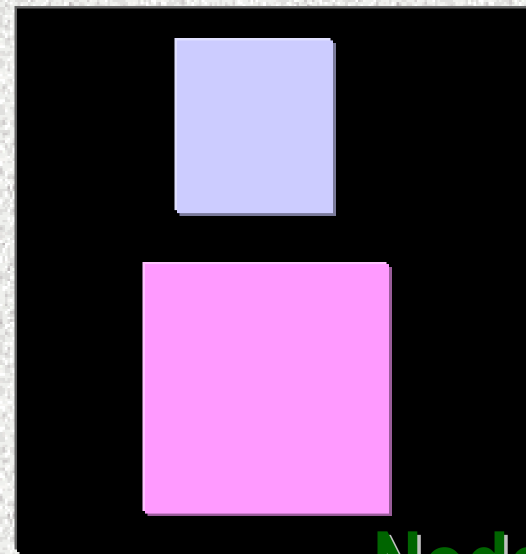
```
with Types; use Types;
with Sensors;
procedure Client_2 is
   T := Sensors.Get_T (Furnace);
```

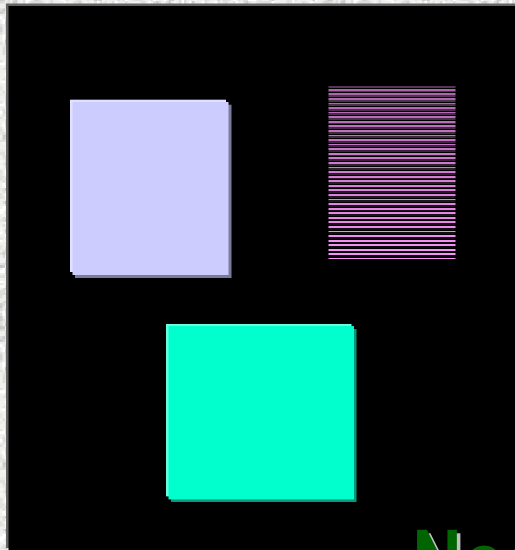
Partition

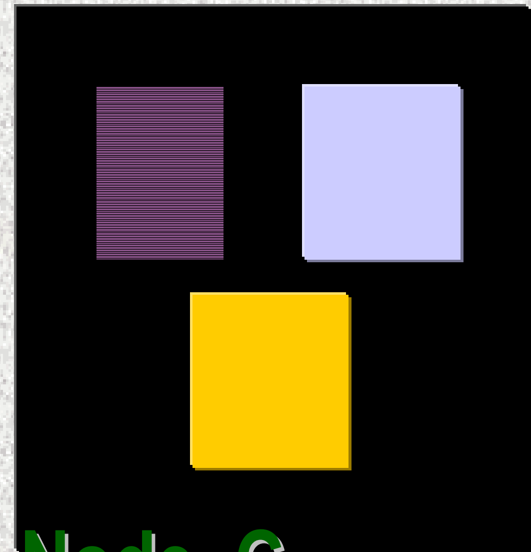
```
configuration Config_1 is
   Node_A : Partition := (Sensors);
   Node_B : Partition := (Client_1);
   Node_C : Partition := (Client_2);
end Config_1;
```


© ACT Europe under the GNU Free Documentation License

Partition



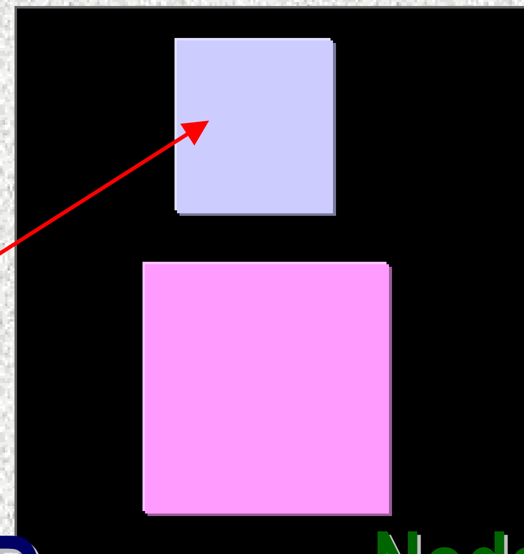


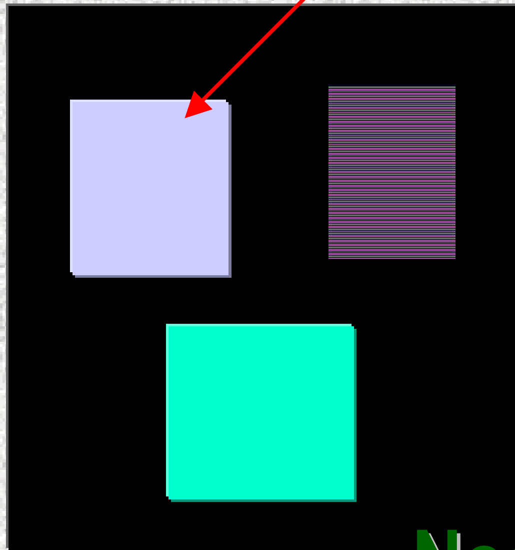


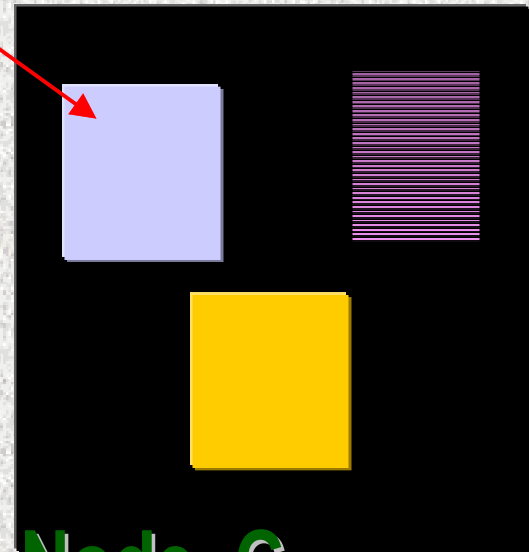
© ACT Europe under the GNU Free Documentation License

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

DUPLICATED

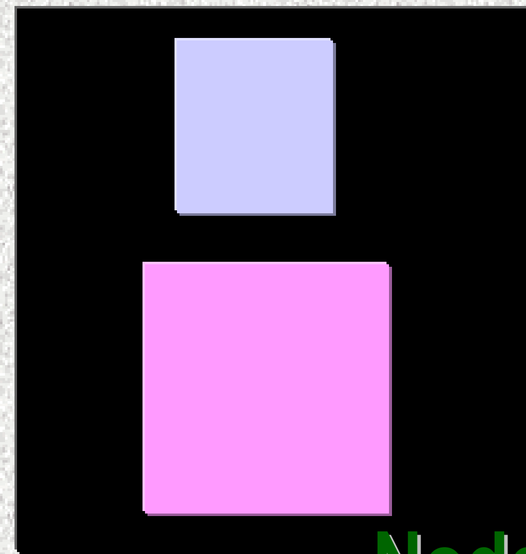




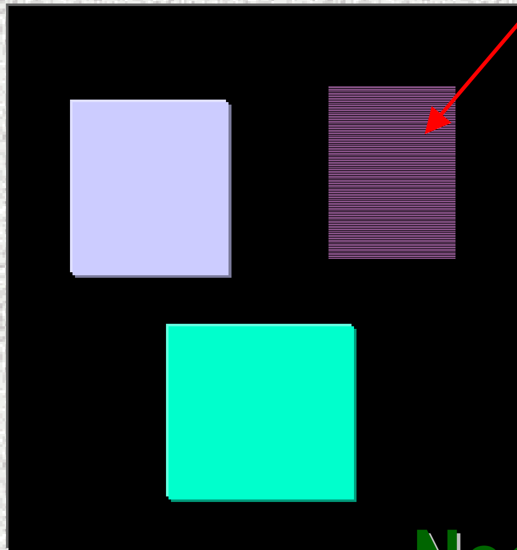


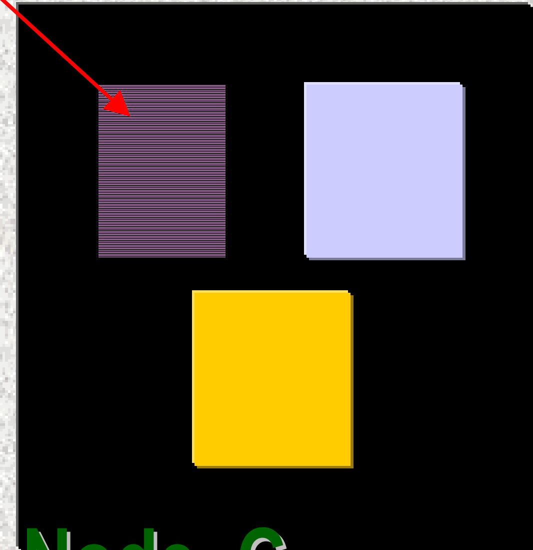
© ACT Europe under the GNU Free Documentation License

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



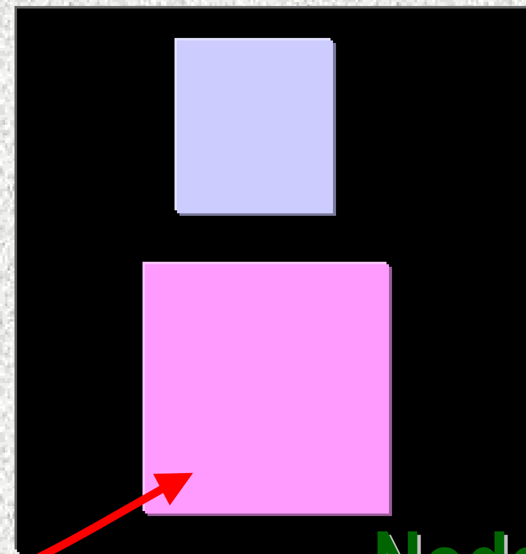
STUBS



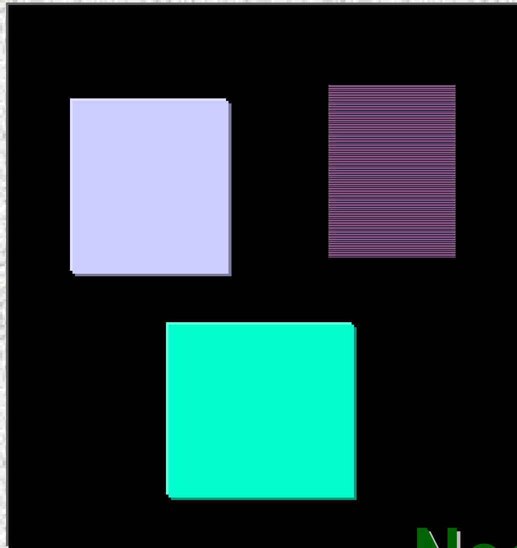


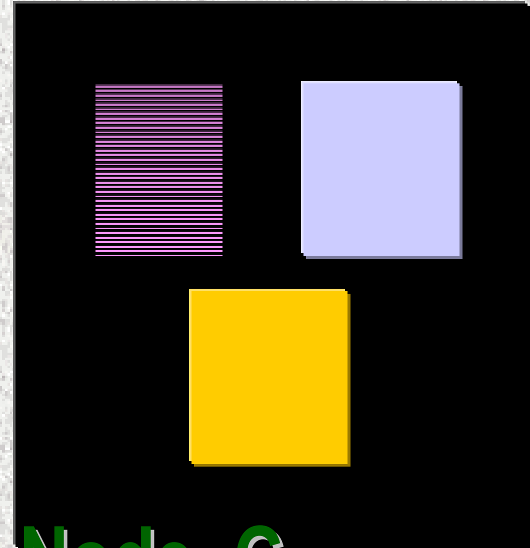
© ACT Europe under the GNU Free Documentation License

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



SKELETON
+ BODY





© ACT Europe under the GNU Free Documentation License

…..:= Sensors.Get_P (Boiler);

Sensors.Get_P **Stub**

Marshal Arguments

Send

Node_B

Sensors.Get_P **body**

Select body

Skeleton

Unmarshal Arguments

Receive

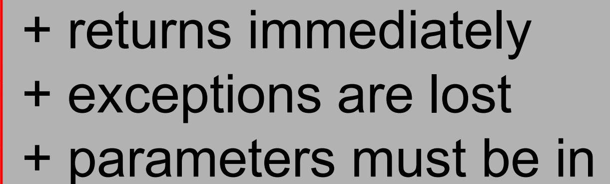
Node_A

© ACT Europe under the GNU Free Documentation License

Asynchronous Calls

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

end Bank;
```



© ACT Europe under the GNU Free Documentation License

Remote_Types

An Example

© ACT Europe under the GNU Free Documentation License

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;
```


©

```
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;
```


© ACT Europe under the GNU Free Documentation License

```
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;
```


© ACT Europe under the GNU Free Documentation License

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;
```


©

```
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;
```


© ACT Europe under the GNU Free Documentation License

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;
```

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;
```


© ACT Europe under the GNU Free Documentation License

What Happens When Executing the Distributed Program ?

© ACT Europe under the GNU Free Documentation License

```
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

© ACT Europe under the GNU Free Documentation License

```
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

http://libre.act-europe.fr
© ACT Europe under the GNU Free Documentation License

```
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 3: Process_Alerts in Node_C does an RPC to Get_Alert in 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

© ACT Europe under the GNU Free Documentation License

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

```
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

© ACT Europe under the GNU Free Documentation License

```
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 5: Node_C performs a dispatching RPC. It calls Handle in Node_AL or Node_AM

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

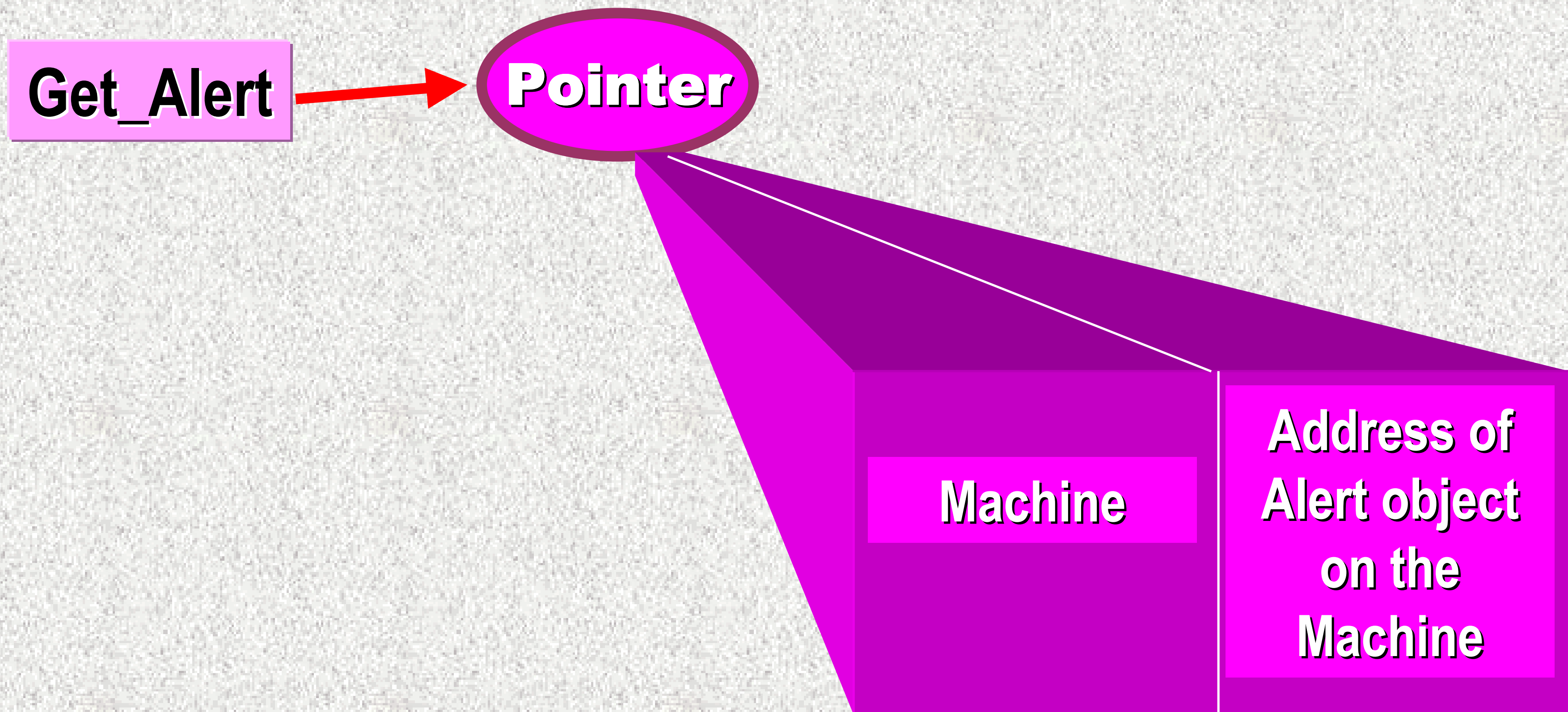
Node_B

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

Node_C

© ACT Europe under the GNU Free Documentation License

What Does Get_Alert Return ?

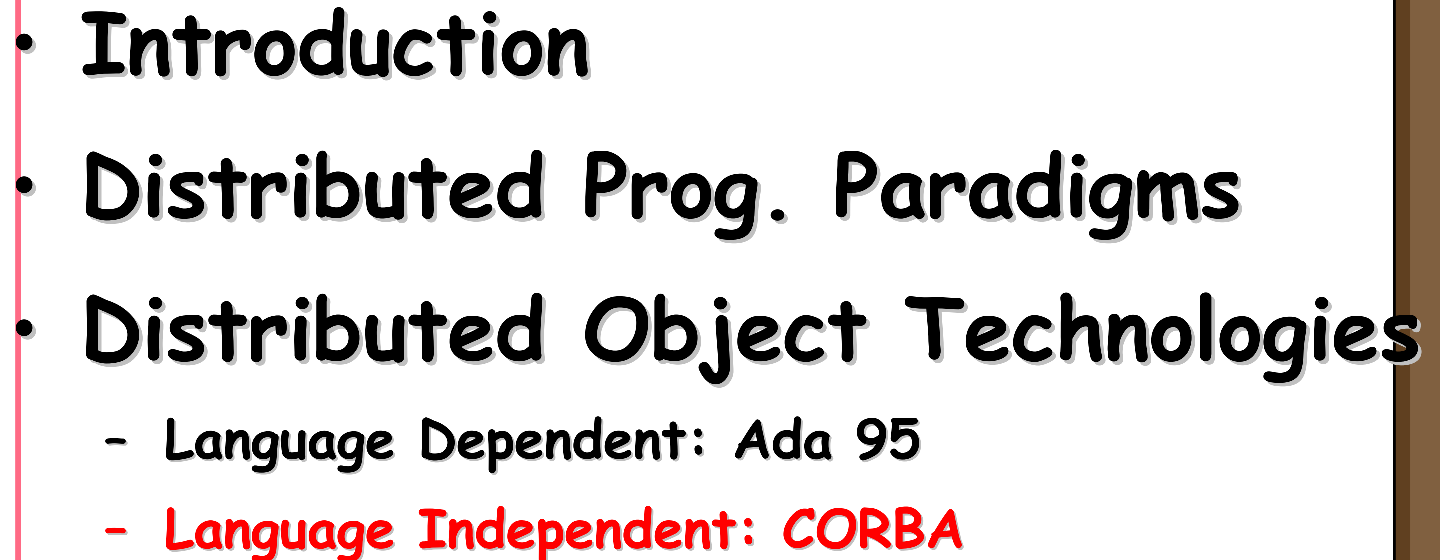


© ACT Europe under the GNU Free Documentation License

Remote Access to Class Wide Type

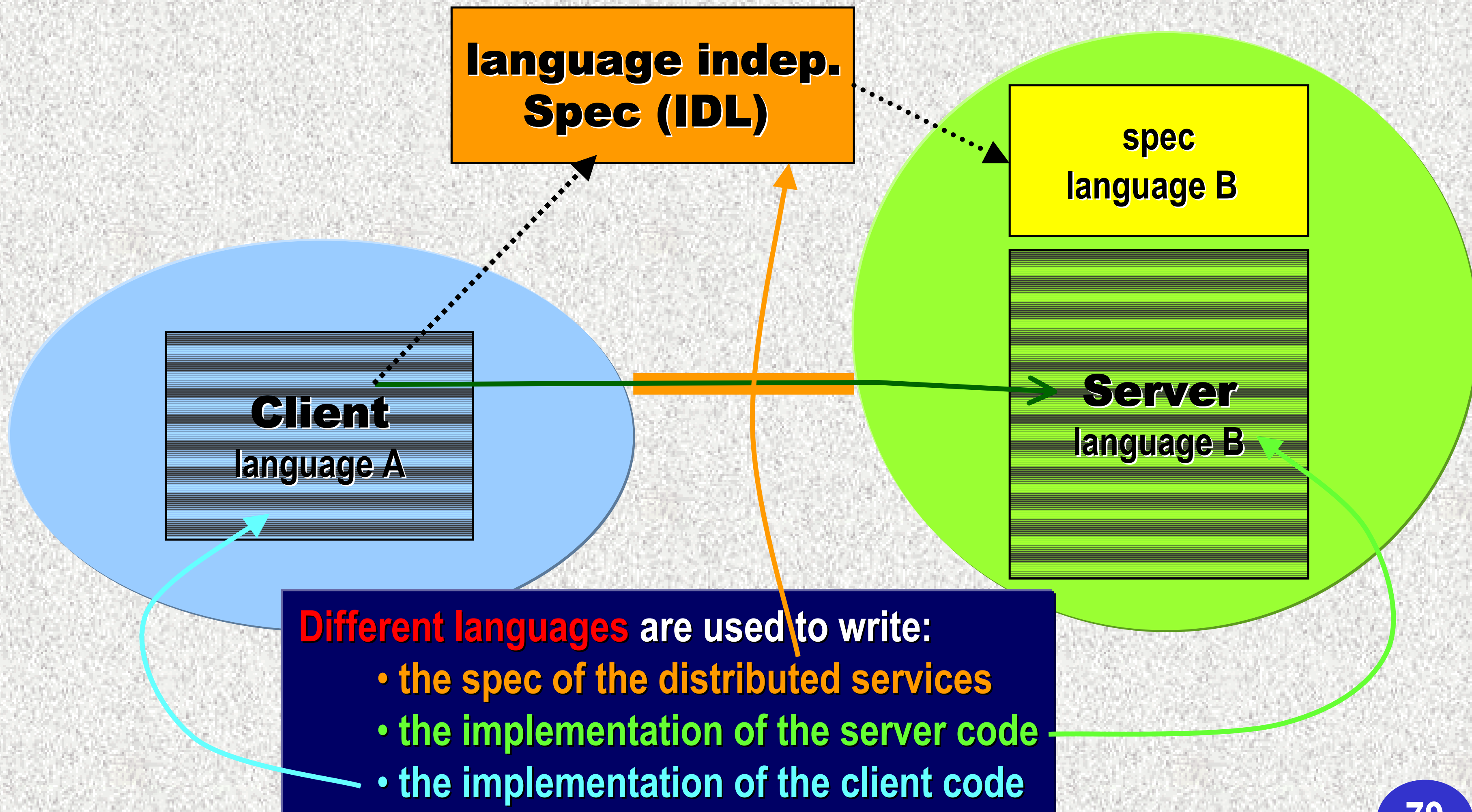
- At compile time:
  - You do not know what operation you'll dispatch to
  - On what node that operations will be executed on

© ACT Europe under the GNU Free Documentation License



© ACT Europe under the GNU Free Documentation License

Language Independent Distributed Objects Paradigm



© ACT Europe under the GNU Free Documentation License

CORBA Interfaces

- In Corba interfaces are described in IDL
  - (Interface Description Language)
- The IDL is independent of programming languages
- Each interface is translated in
  - Language A used for the client (client stub)
  - Language B used for the server (server skeleton)
- To implement the server the programmer completes the skeleton in language B
- To implement the client the programmer uses the services provided by th estub in language A

© ACT Europe under the GNU Free Documentation License

The CORBA Architecture

- RPC go through the ORB
  - (Object Request Broker)
- The ORB is a software bus
- ORBs communicate with a set of standardised protocols
  - IIOP, GIOP

© ACT Europe under the GNU Free Documentation License

The IDL

- Syntax similar to C++ with some Ada additions
- IDL must be translatable in various prog. Languages
  - Ada, C, C++, Java, …
- There are limitations in what you can write in the IDL
- Programmer must understand how the IDL is translated in the host language
  - to complete the server skeleton
  - to use the client stub

© ACT Europe under the GNU Free Documentation License

Example

```
module M {
   interface T {
      void P ();
   };
};
```

```
package M is
   pragma Remote_Types;
   type T is tagged ...;
   procedure P (O : in access T);
end M;
```


© ACT Europe under the GNU Free Documentation License

Exemple

```
module Echo {
    string echoString (in string mesg);
};


 Module foo {
    interface Buffer {
       exception Empty;
       void put (in string content);
       string get() raises (Empty);
    }
 };
```


© ACT Europe under the GNU Free Documentation License

Example of IDL translation in Ada

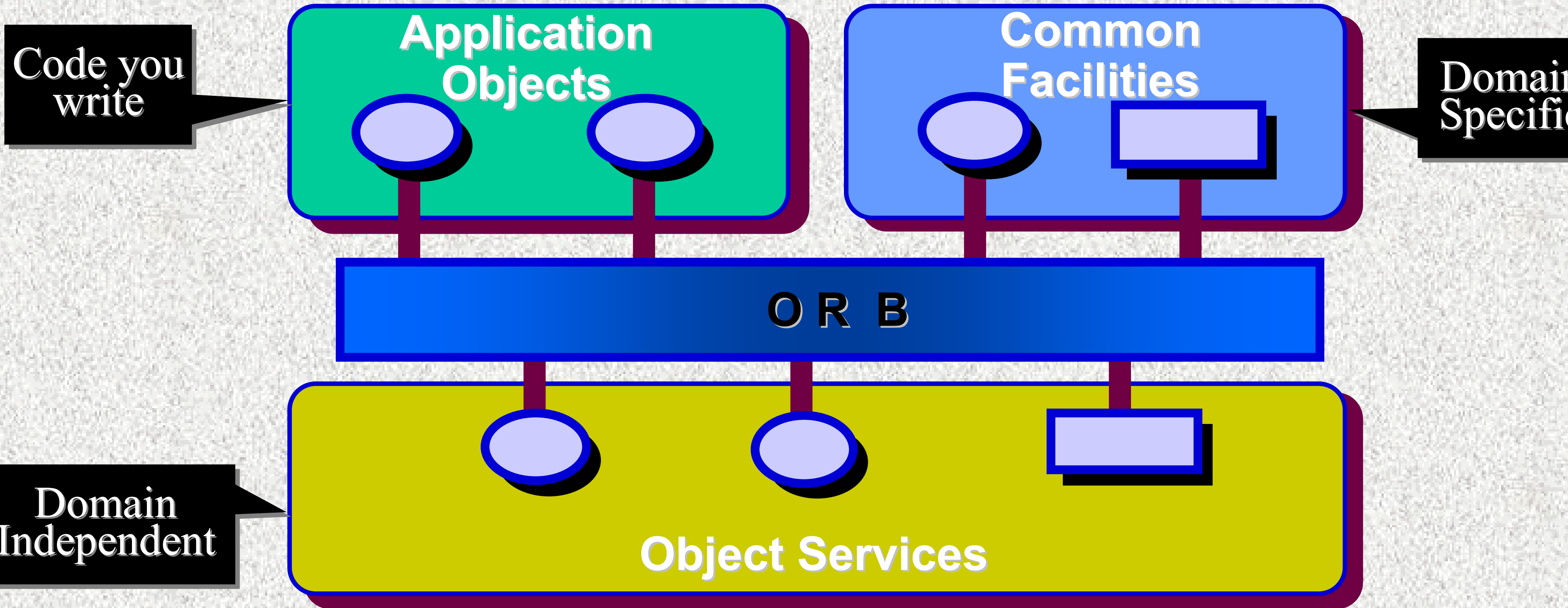
```
with Corba.Object;
package Echo is
   type Ref is new Corba.Object.Ref with null record;
   function To_Echo (Self : in Corba.Object.Ref'Class)
      return Ref'Class;
   function To_Ref (From : in Corba.Any) return Ref;
   function To_Any (From : in Ref) return Corba.Any;
   function echoString (Self : in Ref;
                        msg  : in Corba.String)
                     return Corba.String;
   Null_Ref : constant Ref := (Corba.Object.Null_Ref
                                  with null record);
   Echo_R_Id : constant Corba.RepositoryId :=
      Corba.To_Unbounded_String («IDL:Echo:1.0»);
end Echo;
```


© ACT Europe under the GNU Free Documentation License

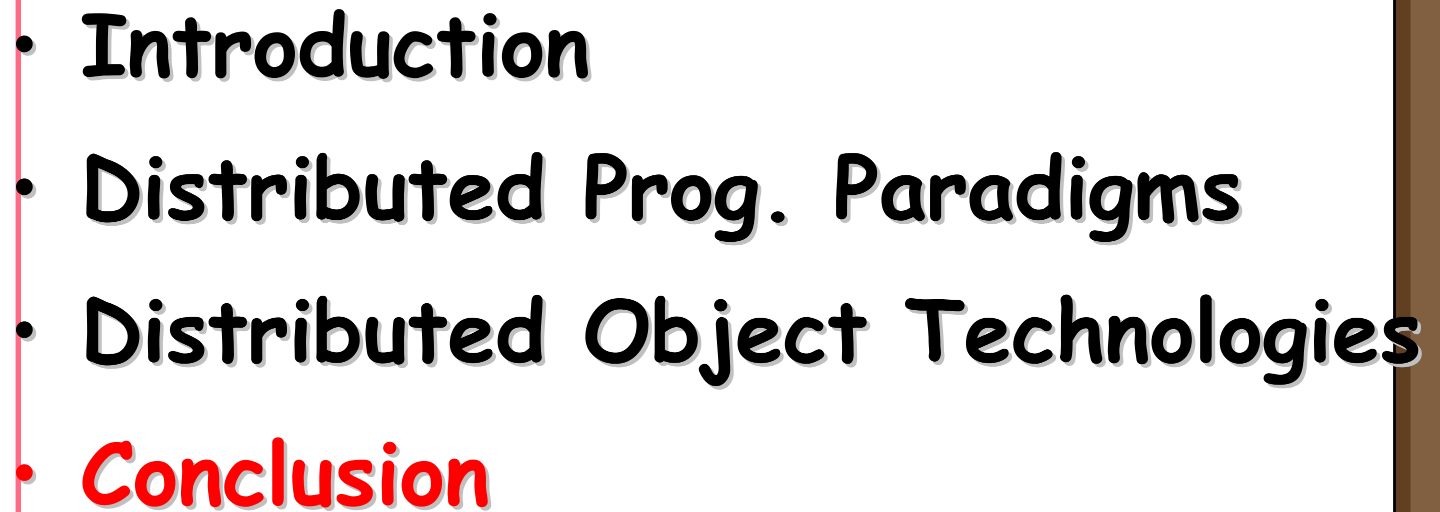
CORBA Services

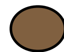
- The CORBA core services are very few
- Lot ’s of external services
  - Naming (distributed and hierarchical)
  - Persistance
  - Transaction
  - Security
  - ...

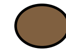
© ACT Europe under the GNU Free Documentation License



© ACT Europe under the GNU Free Documentation License







© ACT Europe under the GNU Free Documentation License

Developing a Distributed App

- Using network services directly
  - Sockets
- Using middleware
  - CORBA
  - COM/DCOM
- Using a distributed language
  - Ada 95 DSA
  - Java RMI

Similar issues with Tasking

© ACT Europe under the GNU Free Documentation License

Impact on Development Phases

General Design

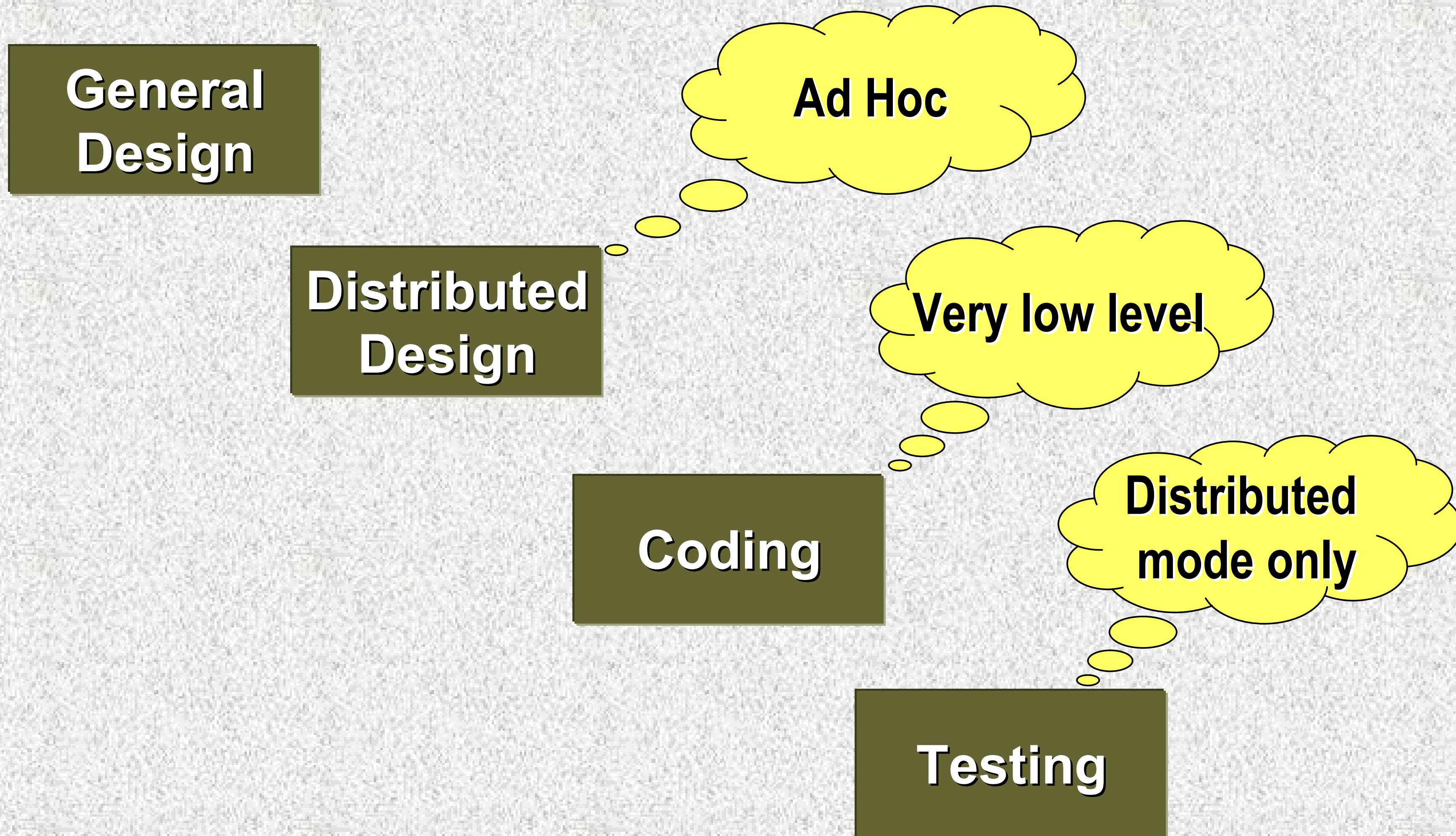
Distributed Design

Coding

Testing

© ACT Europe under the GNU Free Documentation License

Sockets



© ACT Europe under the GNU Free Documentation License

- Everything must be done with sockets
- Data marshaling/unmarshalling
- Handle heterogeneous systems directly

© ACT Europe under the GNU Free Documentation License

CORBA

General Design

Distributed Design

Coding

Testing

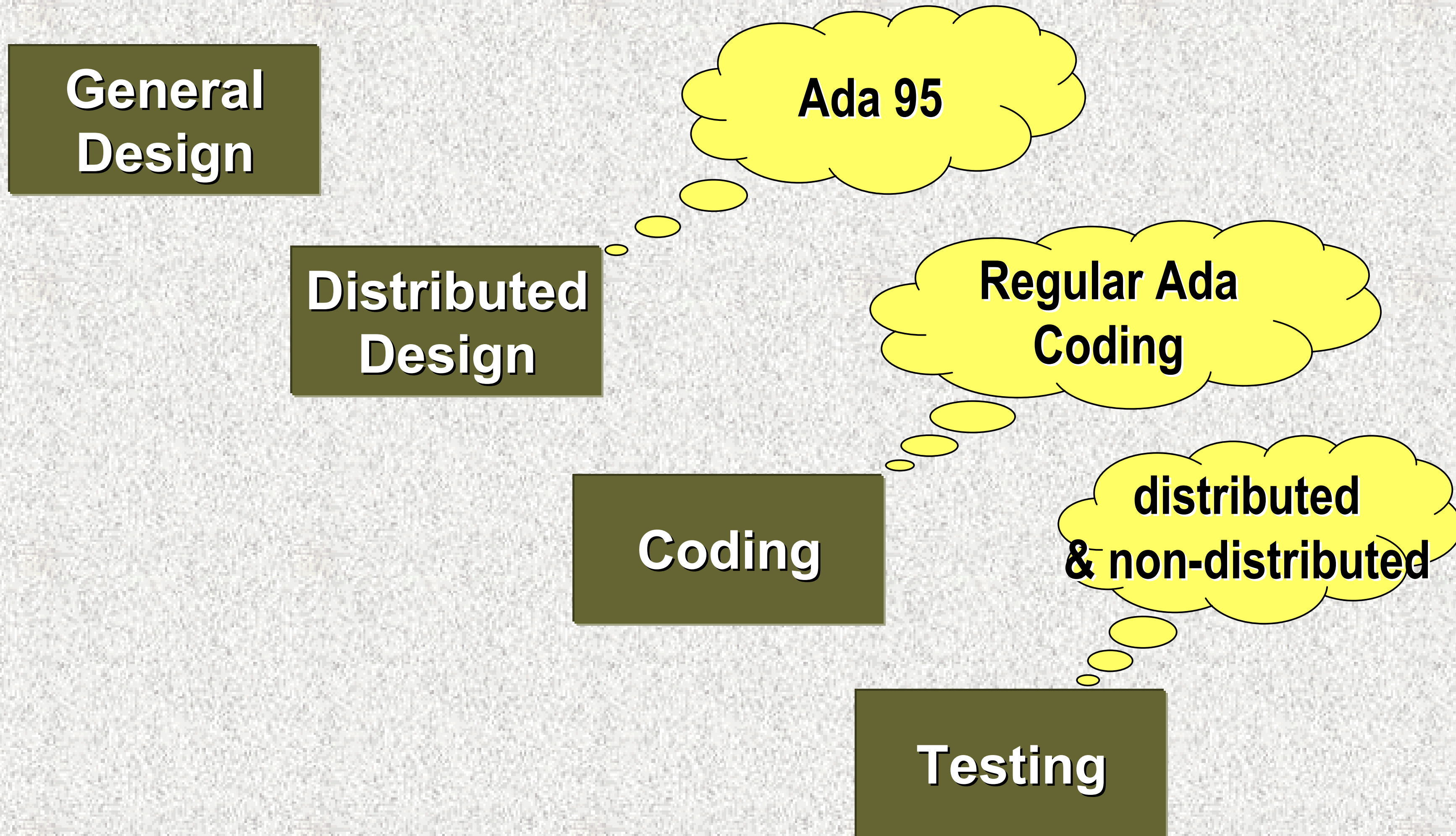
IDL

Must invoke high-level services directly

Distributed mode only

© ACT Europe under the GNU Free Documentation License

Ada 95 DSA



© ACT Europe under the GNU Free Documentation License

Ada 95 DSA & CORBA: Benefits

- Save developer's time, in socket programming:
  - Defining a client/server protocol
  - Defining a message format
  - Marshalling of data
  - Unmarshalling data
- Raise the level of abstraction

© ACT Europe under the GNU Free Documentation License