

# GNAT DISTRIBUTED SYSTEMS ANNEX

## GLADE USER MANUAL

### Introduction

- ☞ An Ada 95 distributed application comprises a number of partitions which can be executed concurrently on the same machine or, and this is the interesting part, can be distributed on a network of machines. The way in which partitions communicate is described in Annex E of the Ada 95 reference manual.
- ☞ A partition is a set of compilation units which are linked together to produce an executable binary. A distributed program comprises two or more communicating partitions.
- ☞ The distributed systems annex does not describe how a distributed application should be configured. It is up to the user to define what are the partitions in his program and on which machines they should be executed.
- ☞ The tool `gnatdist` and its configuration language have been purposely designed to allow you to partition your program and specify the machines where the individual partitions are to execute on.
- ☞ `gnatdist` reads a configuration file (whose syntax is described below) and builds several executables, one for each partition. It also takes care of launching the different partitions (default) and to pass arguments specific to each partition.

### Gnatdist Command Line Options

```
gnatdist [switches] config-file [list-of-partit]
```

The switches of `gnatdist` are, for the time being, exactly the same as for `gnatmake`. Read the `gnatinfo.txt` file from the GNAT distribution for info on these switches. By default `gnatdist` outputs a configuration report and the actions performed. The switch `-n` allows `gnatdist` to skip the first stage of recompilation of the non-distributed application.

All configuration files should end with the `.cfg` suffix. There may be several configuration files for the same distributed application, as you may want to use different distributed configurations according to your computing environment.

If a list of partitions is provided on the command line, only these partitions will be build. In the following configuration example, you can type :

```
gnatdist configuration partition_2 partition_3.
```

### How to Use Gnatdist to Configure a Distributed Application

1. Write a non-distributed Ada application. Use the categorization pragmas to specify the packages that can be called remotely. The `Shared_Passive` categorization pragma is not yet implemented. The `Remote_Call_Interface` and `Remote_Types` categorization pragmas are.
2. When this non-distributed application is working, write a configuration file that maps your categorized packages onto partitions. Don't forget to specify the main procedure of your distributed application (see below).
3. Type `gnatdist configuration-file`
4. Start your distributed application by invoking the start-up shell script or Ada program (depending on the "`pragma Starter`" option).

### Gnatdist Behind the Scenes

Here is what goes on behind the scenes in `gnatdist` when building a distributed application:

1. Each compilation unit in the program is compiled into an object module (as in non distributed applications). This is achieved by calling `gnatmake` on the sources of the various partitions. This step can be skipped by using the `-n` option.
2. Stubs are compiled into object modules (a stub is the software that allows a partition running on machine A to communicate with a partition running on machine B). Several timestamp checks are performed to avoid useless recompilation.
3. `gnatdist` performs a number of consistency checks, for instance it checks that all packages marked as remote call interfaces (RCI) are mapped onto partitions. It also checks that an RCI package is mapped onto only one partition.
4. Finally, the executables for each partition in the program are created. The code to launch partitions is embedded in the main partition except if another option has been specified (`pragma Starter`). In this case, a shell script (or nothing) is generated to start the partitions on the appropriate machines. This is specially useful when one wants to write client / server applications where the number of instances of the partition is unknown.

## The Configuration Language

The configuration language is "Ada-like". Because of its simplicity, it is described by means of an example. As the capabilities of GLADE will evolve, so will this configuration language.

Every keyword and construct defined in the configuration language have been used in the following sample configuration file.

Typically after having created the following configuration file you would type: `gnatdist my_config.cfg`

If you wish to build only certain partitions then list the partitions to build on the `gnatdist` command line as follows:

```
gnatdist my_config.cfg partition_2 partition_3
```

```
configuration My_Config is
```

```
-- The name of the file prefix must be the same as the name of
-- the configuration unit, in this example 'my_config'. The file
-- suffix must be 'cfg'. For a given distributed application
-- you can have as many configuration files as you wish.
```

```
Partition_1 : Partition := ();
procedure Master_Procedure is in Partition_1;
```

```
-- Partition 1 contains no RCI package.
-- However, it will contain the main procedure of the distributed
-- application, called 'Master_Procedure' in this example. If the
-- line 'procedure Master_Procedure is in Partition_1;' was missing
-- Partition 1 would be completely empty. This is forbidden, a
-- partition has to contain at least one library unit.
```

## The Configuration Language (2)

```
Partition_2, Partition_3 : Partition;
for Partition_2'Host use "foo.bar.com";
```

```
-- Specify the host on which to run partition 2.
```

```
function Best_Node (Partition_Name : String)
    return String;
pragma Import (Shell, Best_Node, "best-node");
for Partition_3'Host use Best_Node;
```

```
-- Use the value returned by an a program to figure out at execution
-- time the name of the host on which partition 3 should execute.
-- For instance, execute the shell script 'best-node' which takes the
-- partition name as parameter and returns a string giving the name of
-- the machine on which partition_3 should be launched.
```

```
Partition_4 : Partition := (RCI_B5);
```

```
-- Partition 4 contains one RCI package RCI_B5
-- No host is specified for this partition. The startup script
-- will ask for it interactively when it is executed.
```

```
for Partition_1'Storage_Dir use "/usr/bin";
```

```
-- Specify the directory in which the executables in each partition
-- will be stored.
```

```
for Partition'Storage_Dir use "bin";
```

```
-- Specify the directory in which all the partition executables
-- will be stored. Default is the current directory.
```

## The Configuration Language (3)

```
procedure Another_Main;
for Partition_3'Main use Another_Main;
```

```
-- Specify the partition main subprogram to use in a given
-- partition.
```

```
for Partition_3'Reconnection use
    Blocked_Until_Restart;
```

```
-- Specify a reconnection policy on Partition_3 crash. Any attempt
-- to reconnect to Partition_3 when this partition is dead will
-- be kept blocking until Partition_3 restart. As a default, any
-- restart is rejected (Rejected_ON_Restart). Another policy is to
-- raise Communication_Error on any reconnection attempt until
-- Partition_3 has been restarted.
```

```
for Partition_4'Command_Line use "-v";
```

```
-- Specify additional arguments to pass on the command line when a
-- given partition is launched.
```

```
for Partition_4'Termination use
    Local_Termination;
```

```
-- Specify a termination mechanism for partition_4. The default is to
-- compute a global distributed termination. When Local_Termination
-- is specified then a partition terminates when the local termination
-- is detected (standard ada termination).
```

## The Configuration Language (4)

```
pragma Starter (Method => Ada);
```

```
-- Specify the kind of startup method you would like. There are 3
-- possibilities: Shell, Ada and None. Specifying 'Shell' builds a shell
-- script. All the partitions will be launched from a shell script.
-- If 'Ada' is chosen, then the main Ada procedure itself is used to
-- launch the various partitions. If method 'None' is chosen, then
-- no launch method is used and you have to start each partition
-- manually.
```

```
-- If no starter is given, then an Ada starter will be used. In this
-- example, Partition_2, Partitions_3 and Partition_4 will be started
-- from Partition_1 (i.e. from the Ada procedure Master_Procedure).
```

```
pragma Boot_Server
    (Protocol_Name => "tcp",
     Protocol_Data => "`hostname`:`unused-port`");
```

```
-- Specify the use of a particular boot server. It is especially useful
-- when the default port 5555 used by GARLIC is already assigned.
```

```
Channel_1 : Channel := (Partition_1, Partition_4);
Channel_2 : Channel := (Partition_2, Partition_3);
```

```
-- Declare two channels. Other channels between partitions remain
-- unknown.
```

```
for Channel_1'Filter use "ZIP";
```

```
-- Use transparent compression/decompression for the arguments and
-- results of any remote calls on channel "Channel_1", i.e. between
-- "Partition_1" and "Partition_4".
```

## The Configuration Language (5)

```
    for Channel_2'Filter use "My_Own_Filter";

-- Use filter "My_Own_Filter" on "Channel_2". This filter must be
-- implemented in a package "System.Garlic.Filters.My_Own_Filter".

    for Partition'Filter use "ZIP";

-- For all data exchanged between partitions, use the filter "ZIP". (I.e.
-- for both arriving remote calls as well as for calls made by a parti
-- tion.)

begin

-- The configuration body is optional. You may have fully described
-- your configuration in the declaration part.

    Partition_2 := (RCI_B2, RCI_B4, Normal);

-- Partition 2 contains two RCI packages RCI_B2 and RCI_B4
-- and a normal package. A normal package is not categorized.

    Partition_3 := (RCI_B3);

-- Partition 3 contains one RCI package RCI_B3

end My_Config;
```

## Example

```
with Text_IO; use Text_IO;
with Server; use Server;
with Client; use Client;

procedure start is
--i : Integer;
begin
    Put_Line("Calling service");
    Server.F1(3,4, Call_Back'Access);
    Put_Line("Call done...");
    for i in Integer range 1..20 loop
        Put_Line("..."); Delay(0.5);
    end loop;
    Put_Line("Client tarminated...");
end start;
```

## Example

```
package Server is

    pragma Remote_Call_Interface;

    type Back_Reference is access procedure (
        ret : Positive);

    procedure F1 (a,b : Positive;
        bptr : Back_Reference);
    pragma asynchronous (F1);

end Server;
```

```
package Client is

    pragma Remote_Call_Interface;

    procedure Call_Back(val : Positive);

end Client;
```

```
with Text_IO; use Text_IO;

package body Client is

    procedure Call_Back(val : Positive) is
    begin
        Put_Line("Client: Got the result
        :"&Integer'image(val));
    end Call_Back;

end Client;
```