The Chance for Ada to Support Distribution and Real-Time in Embedded Systems

Juan López Campos, J. Javier Gutiérrez, and Michael González Harbour

Departamento de Electrónica y Computadores
Universidad de Cantabria, 39005 - Santander, SPAIN
{lopezju,gutierjj,mgh}@unican.es

Abstract. This paper presents a modification of GLADE —the current GNAT implementation of the Ada 95 Distributed Systems Annex (DSA)— to support the development of distributed applications with hard real-time requirements. This modified implementation, that we call RT-GLADE (Real-Time GLADE), is specially suitable for embedded applications composed of a small number of heterogeneous processors and communication networks, because it ensures predictable timing behaviour. A real-time model of the implementation allows the application developer to determine and optimize the overall timing behaviour by applying the corresponding schedulability analysis and priority assignment techniques. This realtime version of GLADE continues to conform to the DSA, so the entire real-time application can be built within the Ada 95 context. To implement RT-GLADE, we provide a priority-based communication network over standard Ethernet that is used to ensure predictable transmission times.

Keywords: Real-Time, Embedded Systems, Distributed Systems, Ada 95, Modelling, Schedulability.

1 Introduction[1]

In the past 20 years the concept and necessity of distribution in computer systems have received increasing attention, and the technology that allows us to make this distribution now offers a wide range of possibilities. We have seen how the distribution paradigms have been developing since the message passing services to the remote procedure calls, to the distributed objects, or more recently, to the distributed components. This need of distribution was included in the Ada 95 standard [16] in its Annex E, no doubt coming from the needs of Ada users. For instance, until the adoption of the Ada 95 standard, only in the ACM SIGADA Ada-Letters series, at least 38 papers appeared regarding distributed systems and Ada in different aspects such as real-time, fault-tolerance, communications, or modelling**.**

However, the truth is that this Annex has had a minimal impact on the different environments involving the development of distributed applications. We may think of multiple reasons, but perhaps, the most important one is that instead of using the implementation of the Distributed Systems Annex (DSA), Ada developers have

1. This work has been funded in part by the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) of the Spanish Government under grant number TIC2002-04123-C03-02 (TRECOM), and by the IST Programme of the European Commission under project IST-2001-34820 (FIRST).

always been able to find an alternative that could better adapt to the specific requirements of their application. For example, just for distribution, they could choose changing the programming language to Java, which supports a more modern distribution infrastructure. If they sought integration with software developed in other languages and for different platforms, they could choose the CORBA objects distribution [10]. As far as this last point concerns, there are studies in which some integration strategies between the Ada and CORBA worlds have been discussed and considered [12][15].

But perhaps the most important reason why the DSA has not been widely used is that in the application environment in which Ada is strongest, real-time systems, the DSA does not provide the required timing behaviour predictability and controllability. We think that if the DSA would support distribution of real-time applications, it could be used in those environments in which Ada is chosen as the superior real-time programming language.

The motivation for this paper is to obtain an implementation of the current Annex E that could be used for real-time applications, showing that the changes and additions to the DSA proposed in [6] are viable and complete. The implementation discussed in this paper is targeted at the application environment in which we usually work, which is embedded industrial control systems (robots or visual inspection systems) composed of different distributed processors connected by one or more communication networks, and in which it is necessary to guarantee temporal requirements.

There are a few implementations of the Distributed Systems Annex, that support partitioning and allocation of Ada applications on distributed systems. One of these implementations is GLADE, which was initially developed by Pautet and Tardieu [11][13] and is currently included in the GNAT project, developed by Ada Core Technologies (ACT) [1].

The implementation that we present in this paper is called RT-GLADE (Real-Time GLADE), because it was built by modifying GLADE to enhance its real-time capabilities. The modifications are based on our previous work towards the specification of a distributed real-time annex for Ada [5][6].

In order to develop real-time applications it is also necessary that RT-GLADE has access to communication networks with real-time features. In this paper we integrate into RT-GLADE the real-time network RT-EP [8], which implements a token-passing protocol over standard Ethernet hardware. Another important need in this kind of systems is a real-time operating system. We have implemented RT-GLADE with MaRTE OS [2], which is a real-time operating system that follows the POSIX Minimal Profile [14]. Because MaRTE OS is based on version 3.15p of the GNAT compiler, we have modified the 3.15p version of GLADE, the last public version available, which is compatible with that compiler. Having built all the parts of the system with real-time behaviour —application, Ada run-time system, underlying OS, communications middleware, and communications network—, it is possible to build a model that allows a correct characterization of the timing response [4][9][7].

The paper is organized as follows. First, in Section 2 we present an overview of GLADE to show its architecture and the main characteristics related to the real-time behaviour and the compliance with the DSA. Section 3 presents new capabilities introduced in the RT-GLADE implementation, as well as the subset of the configuration parameters usable for real-time applications. It also shows some details of the communication subsystem that supports real-time requirements. In Section 4 a simple example on how to use RT-GLADE to build an application is shown. Section 5 gives metrics that help in evaluating the benefits of the new implementation. Finally, in Section 6 we draw our conclusions.

2 Ada Distribution and GLADE Overview

Ada 95 [16] defines a distributed system as an interconnection of one or more processing nodes, and zero or more storage nodes, with some communication means among those nodes. It also defines a distributed program as one or more partitions that execute independently in the distributed system. The partitions communicate with each other by exchanging data, using remote subprogram calls and distributed objects. There are two kinds of Ada 95 partitions: active, which can execute in parallel with each other, possibly in a separate address space and possibly in a separate computer; and passive, which have no thread of control of their own, have all their library units preelaborated, and their data and subprograms are accessible to one or more active partitions. The communication between active partitions is made in a standard way using the facilities provided by the Partition Communication Subsystem (PCS). The PCS has a language-defined interface given by the package `System.RPC`, so an implementation of the PCS can be independent of the compiler and the run-time system.

The DSA leaves some important issues as implementation defined [5][6]. Some of them are very important to develop an implementation that has real-time capabilities. For example, the way in which RPCs are handled, or the priorities at which the task executing an RPC should execute are totally dependent on the implementation. Other implementation-defined issues like the configuration language to describe the distributed system, which may be important from a standardization perspective, are not so important from the real-time point of view.

GLADE is the first industrial-strength implementation of the distributed Ada 95 programming model. The work in [13] proposes GLADE as a framework for developing object-oriented real-time distributed systems. However, [5] and [6] show that there are issues that need to be addressed for a predictable and controllable implementation of the DSA, which are focused on the priority management in the RPC handlers and on the restrictions that have to be observed when configuring the system. Some but not all of these issues were implemented in version 3.15p of GLADE.

GLADE is divided in two major parts [11]:

- GARLIC: the Partition Communication Subsystem, which is primarily composed of the packages `System.RPC` and `System.Garlic` and its child packages, containing the heart of the PCS that takes care of network-related system calls, concurrent requests, partition localization and launching, error handling and recovery, etc.

- GNATDIST: the partitioning tool, which is responsible for checking the consistency of a distributed system before building it, calling GNAT with the appropriate parameters to build the needed stubs, configuring the filters that will be used between different partitions, linking the partitions with GARLIC and building the initialization sequence, and building the main program that will launch the whole distributed application on the specified hosts.

Inside GARLIC [3], a pool of *RPC handler* tasks of the type `RPC_Handler` is created at initialization time to take care of concurrently executing the RPCs on a given partition. This preallocation of tasks is done for the purpose of avoiding the overhead of task creation and destruction at each RPC. If an RPC arrives and all the tasks in the pool are being used by previously issued RPCs, then a new task will be created for the new RPC. The pool of tasks can be configured with the partition attribute `Task_Pool` of the configuration language, which allows expressing three parameters: the task pool minimum size (number of RPC handlers preallocated and always available), the task pool high size (when an RPC is completed, its RPC handler task is deallocated if the number of task in the pool is greater than this ceiling), and the task pool maximum size (it is a limit to the number of simultaneously active remote calls; if the number of active remote calls is greater than this number, then the request is kept pending until an RPC handler task becomes available).

GLADE also uses dynamic allocation of tasks when a message arrives at the receiver partition of an RPC. Every partition has one or more TCP/IP incoming ports (created from configuration parameters such as the `Boot_Location` pragma, or the `Self_Location` partition attribute), and for every incoming port a new task of the type `Accept_Handler` is created at initialization time, waiting for incoming messages. When a message arrives, this task hands over the processing of the message to another task of the type `Connect_Handler`, which is found in a second pool of tasks. At initialization time, this pool is empty, and the task is dynamically created in case it is necessary. The `Connect_Handler` task takes care of processing the message and calling the RPC handler task that executes the subprogram, meanwhile leaving the `Accept_Handler` free to wait for new incoming messages at the reception port.

Although it is possible to statically allocate the RPC handler tasks, allocation of the `Connect_Handler` tasks is dynamic and is not appropriate for real-time applications. There needs to be a mechanism for avoiding the dynamic allocation of internal tasks.

The DSA has no provision for expressing the priorities at which the task executing an RPC should execute, nor the priorities of the messages in a communication network. In GLADE 3.15p the concept of priority policy is introduced, with two possible values: `Client_Propagated` and `Server_Declared`. The first value provides a simple mechanism to express the execution priority of the RPC that is transparent to the user: the priority of the task invoking the RPC is encoded in the message sent to the receiver partition. Then, in that partition, the RPC handler task that will execute the call reads the priority of the original calling task, which is encoded in the received stream, and sets its own priority to that value. It is known, however, that this is not the optimum priority assignment [5]. The priority policy is set via a configuration pragma (`Priority`), which means that all the partitions will use the same policy. The `Server_Declared` value establishes a fixed priority for all the RPC handler tasks in a given partition using a partition attribute (`Priority`). The original priority of the RPC handler task is set to the maximum level by default, and also returns to this level when the task finishes its work. In [5] we showed that it was possible to achieve better results if the application could specify the initial priority of the RPC handler as well as the priority for each call. In addition, GLADE does not support specifying priorities for the messages in the communication networks.

In GLADE, calls between two partitions allocated in the same processing node are made using the network capabilities, while it would be more efficiently done by using some local protocol that would avoid going through the network.

There are also some aspects in GLADE introducing non uniform overheads that should be minimized in hard realtime systems. For example, once a task of the calling partition has sent a message over the network to make an RPC, it waits for the answer, the RPC reply, on a single entry of a protected object. Every task of the calling partition will be enqueued in the same entry, so, when an RPC reply is processed, it is necessary to determine which task is the 'owner' of the answer. To do this, every task enqueued in the entry is dequeued, and then all the tasks except the one accepting the RPC reply are enqueued again. This implies an unnecessary overhead that is proportional to the number of tasks waiting for an RPC reply.

The `Shared_Passive` pragma defined in the DSA is used for managing global data shared between active partitions. GLADE's implementation of this pragma is made using operating system files to hold the data corresponding to passive partitions, and it is assumed that the data will be shared using some kind of network file system. However, most implementations of these distributed file systems do not have real-time capabilities, and therefore we will not address support for this pragma in RT-GLADE.

3 RT-GLADE Characteristics

The modifications made to GLADE to support real-time behaviour are mainly focused on the following aspects, that will be described separately in more detail:

- Full application control of the priorities of the RPC handler tasks, and of the messages sent across the network, according to the recommendations given in [5].
- Incorporating a priority-based communication network based on standard Ethernet and a new local protocol to increase the efficiency of the calls into the same processing node.
- Removing the dynamic creation of the tasks at the receiving end of an RPC and improving the wait mechanism for the RPC reply in the calling partition.
- Adapting the configuration of the system to the new real-time capabilities, including the aspects related with the real-time communication networks.

The proposed modification makes the implementation still conform to the current DSA, although a few new interfaces are needed to support the real-time features.

3.1. Priority Management in the Overall System

In order to be able to control the timing behaviour of a real-time application we need some mechanism to specify the priorities for the RPCs that will be executed, as well as for the messages that will be sent across the network.

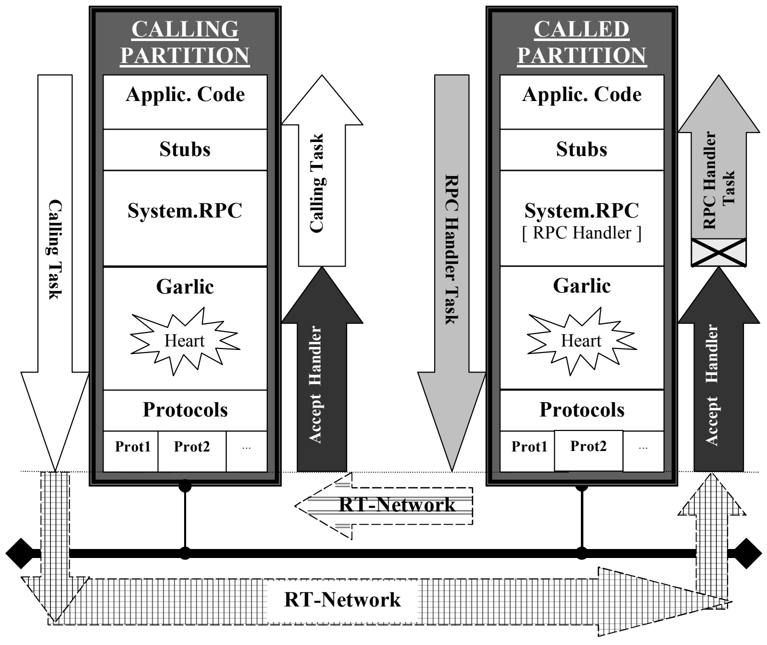
According to the discussion in [6], we use the type `Global_Priority` defined in package `System.Garlic.Priorities` and representing a value with a global meaning across the distributed system, and we modify that package to make it pure because we need to exchange priorities across the different partitions. We also use the mapping functions defined in `System.Garlic.Priorities.Mapping` that translate between values of this global priority type and values of `System.Priority`. The same naming scheme is used for adding the new mapping functions between the type `Global_Priority` and the priorities of the RT-EP network that we use in our implementation.

In addition, we create the `RPC_Priorities` package, which contains the operations to set the priorities of the outgoing message that is sent by the partition calling an RPC, of the RPC handler task for that particular call, and of the incoming message that is returned by the called partition when the execution of the RPC has finished:

```
with System.Garlic.Global_Priorities;
use System.Garlic.Global_Priorities;
package RPC_Priorities is
   procedure Set
      (RPC_Handler,
       Outgoing_Message,
       Incoming_Message: in Global_Priority);
   procedure Get
      (RPC_Handler,
       Outgoing_Message,
       Incoming_Message: out Global_Priority);
end RPC_Priorities;
```

Procedure `Set` is used to set the priorities used for future RPCs or APCs issued by the calling task, and can be invoked by the user before making the call. These priorities are in effect until `Set` is called again. In this way, the application can specify the priorities of its RPCs either on an individual basis, or by grouping several calls under the same priorities. Initial values for the priorities are set to an intermediate priority level. Procedure `Get` returns the current values of the RPC priorities. The implementation of this package stores the priorities by creating three task attributes using the facilities described in the optional but standard package `Ada.Task_Attributes` [5].

To avoid context switches, we have moved the work done by the `Connect_Handler` tasks into the `Accept_Handler` tasks. Since the underlying communication subsystem implements message queuing there is no message loss using this strategy.



	RT-GLADE	GLADE
▭	User Priority	User Priority
▦	Outgoing Priority	No Priority
■	Accept Handler Priority	Priority'Last
☒	RPC Handler Priority	Priority'Last
▩	RPC Handler Priority	Server Declared/Client Propagated Priority
▤	Incoming Priority	No Priority

Fig. 1 Priority Schemes in GLADE and RT-GLADE

To avoid the priority inversion that could be caused by the initial priority of the RPC handler tasks of the pool, we let the application configure it by specifying in the configuration file the partition attribute called `Priority`, of the type `Global_Priority`. In our implementation we use this attribute to also set the priority of the `Accept_Handler` tasks.

Fig. 1 shows a typical RPC call and how the priorities are managed in the original GLADE and in our implementation. For the latter, the calling task sets the priority values involved in the RPC: for the outgoing message, the incoming message, and the execution of the RPC in the called partition. Then it executes all the middleware code at its own priority until it sends the request through the real-time network. When the calling task reaches the `Do_RPC` or `Do_APC` functions of the `System.RPC` package, instead of writing the priority of the calling task in the message stream as in the original GLADE, we call `RPC_Priorities.Get` and we write the `RPC_Handler` and `Incoming_Message` priorities into that stream. The priority at which the stream is sent across the network is the `Outgoing_Message` priority (after mapping it to the appropriate type).

Once the message arrives at the called partition, an `Accept_Handler` task (see Subsection 3.3) running at the established priority processes the stream. As part of this processing, it reads the priority of the corresponding RPC handler from the message, selects a free handler from the pool, sets its priority to the desired value, and then awakens the handler passing the RPC stream to it. This strategy saves context switches compared to the original GLADE implementation in which it was the own RPC handler who set its own priority.

The RPC handler task reads the `Incoming_Message` priority and the parameters of the call from the message stream, and then invokes the subprogram associated with the RPC. When the call completes, the return parameters and error indications, if any, are sent to the calling partition at the `Incoming_Message` priority level. Then, the RPC handler task suspends itself by going back into the pool. It's priority is left unchanged because before it will start executing again an `Accept_Handler` task will set its priority to the value appropriate for the new RPC.

When the stream with the return parameters arrives at the calling partition, it is processed at the established priority by an `Accept_Handler` task (see Subsection 3.3) which in turn passes it to the calling task that was suspended waiting for this answer.

3.2. New Communication Features added to GARLIC

As RT-GLADE uses a network, it is obvious that it is necessary to use one with real-time characteristics. The TCP/IP network provided by GLADE is not suitable for real-time communication. In our RT-GLADE implementation we are using the RT-EP (Real-Time Ethernet Protocol) network [8], which is a software-based token-passing Ethernet protocol for multipoint communications in real-time applications using standard ethernet hardware. This protocol can manage messages up to 1492 bytes long in a single packet, but it is currently being extended to manage larger messages using packet division.

Within the implementation other real-time protocols may be available, so when an RPC is performed, it is necessary to determine the protocol that we are going to use for the communication. Once the protocol has been selected, it is necessary to pass to it the information required to contact the remote partition through the network. This information comprises the node identification where the partition is located, and any other possible information for the protocol chosen, e.g., the destination port as in the GLADE implementation.

RT-EP [8] uses a concept similar to the port, called the reception channel, which is a number used for the purpose of identifying communication endpoints in a given station. To be able to transmit through RT-EP it is necessary to know the MAC address of the destination node, and the RT-EP reception channel number inside that node. We code this information using the partition attribute `Self_Location` and the `Boot_Location` pragma of the configuration language (see Subsection 3.4).

We have defined a new virtual network protocol that we call the Local protocol, which is used to increase the efficiency when making a call to a subprogram belonging to an RCI library unit in a partition located in the same processing node than the calling partition. The use of the Local protocol also avoids a limitation of the RT-EP communication subsystem, which does not allow sending a message to the same node. This protocol is implemented with the same interface as RT-EP [8], also using a reception priority queue for each reception channel defined in the configuration process. A message sent through the Local protocol is enqueued with the priority of the called task, which is the RPC handler priority for the request and the priority of the calling task for the answer. When a message is sent through the Local protocol, neither the `Outgoing_Message` nor the `Incoming_Message` priorities are used, because there is no network scheduling involved. Internally, the priority of the calling task is written into the stream in order to enqueue the answer message in the right order.

Once a calling task executing an RPC has checked that the call does not belong to the same partition nor to a partition located in the same node, the implementation has to determine how to contact the remote partition, i.e., which protocol to use. For this purpose, the calling task checks a table with information on which protocol to use for accessing the desired partition. It is not necessary that if, for instance, partition P1 had to contact partition P2 using the RT-EP protocol, partition P2 had to use the same protocol to contact P1. In the configuration process, we use the partition links (see Subsection 3.4) to select the protocol to be used between a calling partition P1 and a called partition P2.

3.3. RPC Receiver Handling

RT-GLADE removes the dynamic task creation of the GLADE implementation for the execution of RPCs simply by configuring the number of RPC handler tasks in the pool to a sufficient number of static tasks, ensuring the execution of all the possible concurrent RPCs. For this purpose, we just set the minimum number of tasks in the pool using the partition attribute `Task_Pool`. If we want a static pool of tasks, the other two numbers in this attribute (high and maximum sizes) must be set to the same value as the minimum. This also follows the recommendations stated in [6].

The other place of the GLADE implementation with dynamic creation of tasks is the management of the `Accept_Handler` and `Connect_Handler` tasks. In RT-GLADE each partition has one `Accept_Handler` task (and no `Connect_Handler` task) per reception channel configured in the real-time protocol (currently the Local protocol and RT-EP). These tasks are created at initialization time and are immediately blocked waiting for the reception of messages at their respective reception channels.

In order to avoid the non uniform overhead associated with the use of a unique entry to wait for the RPC replies, we propose using a family of entries. Every task waiting for an RPC reply will be enqueued on a different entry. Before issuing the outgoing message, we determine the identifier of the entry at which the calling task will be enqueued. This identifier is sent with the message and then sent back with the reply to determine which entry is to be serviced. After usage, the entry is freed for subsequent use in some other call. A new configuration parameter is added to specify the number of entries of the protected object.

3.4. RT-GLADE Configuration Parameters

The GNATDIST tool has its own interface for implementing the configuration of a distributed application. This tool reads a "`.cfg`" file which follows the rules of its Ada-like configuration language. RT-GLADE uses a subset of GLADE's configuration language together with a few extensions. We will summarize which of the GLADE configuration attributes and pragmas are specially relevant to RT-GLADE, which are not used, and which are forbidden:

- The configuration and partition declarations, or setting the main procedure in a partition are the same as in the original tool.
- Pragma `Starter` has to be set to `None`, because we want to manually launch the different partitions generated. The reason is that RT-GLADE will be tested on a real-time OS (MaRTE) that does not have the conventional remote shell used by the GLADE launcher. Each MaRTE OS node can automatically upload the partition from a host where the partitions have been created.
- It is not necessary to set the partition attribute `Host` for every partition. This attribute is related with the remote launching facility that is not used in MaRTE.
- The partition attribute `Self_Location` is used to configure how a partition can be reached through a communication protocol. Each partition must specify this attribute at least once for every possible location in which the partition can be allocated. The information coded in this attribute is a list of: a protocol name, a processing node identifier, a reception channel identifier, and a list of partitions to which corresponding partition links will be set. A partition link defines the preferred protocol to send messages from the specified partition to the one to which the attribute `Self_Location` is applied.

  Pragma `Boot_Location` has to be specified also in order to determine the location of the boot server in which all the partitions have been registered. This boot server is located on the main partition. It contains the same information as `Self_Location`.

  In order to completely specify the network information an additional file must be written to associate the network addresses (MAC addresses for RT-EP) and the processing node identifiers. The reason for adding this new file is to not overload the configuration file with information that is only relevant to the communication protocols.
- The partition attribute `Task_Pool` must be configured as described in section 3.3.
- The filtering service is not necessary and, consequently, nor are the bidirectional channels that GLADE implements, which are mainly focused on this service. This part of the configuration is ignored.

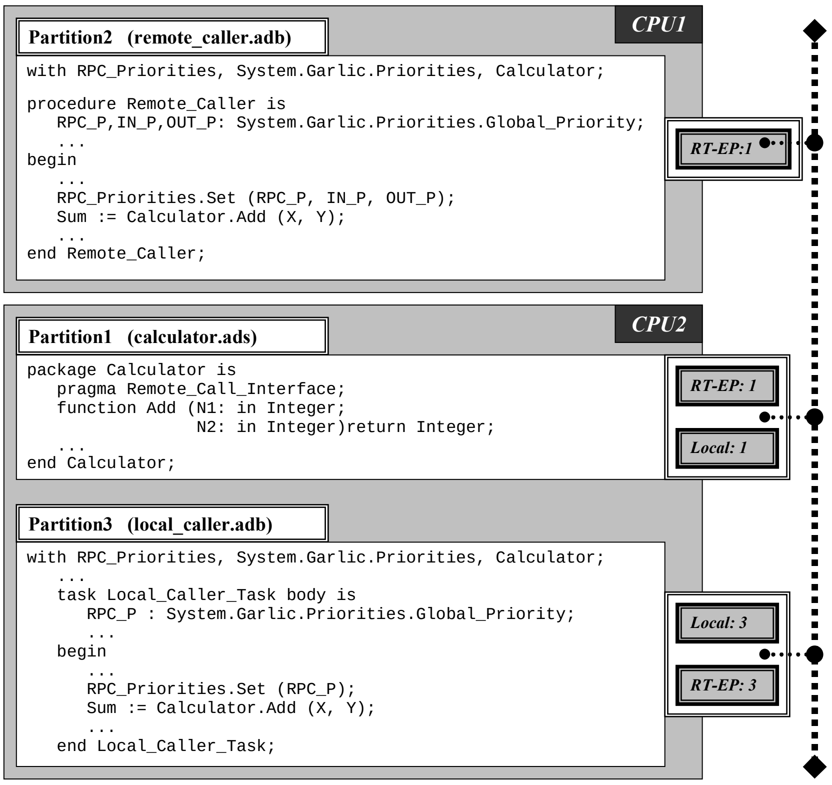
- Pragmas or attributes related to the configuration of passive partitions are forbidden. More work has to be done to arrive at a real-time solution for this issue.
- Dynamic aspects associated with the reconnection in GLADE are also forbidden for hard real-time applications due to the difficulties in achieving predictable timing behaviour; they could be used in a future implementation for soft real-time systems.
- The partition attribute `Priority` has been reused to configure the initial priorities of the `Accept_Handler` tasks.
- A new partition attribute called `Max_RPC_Replies` has been added to set the number of entries of the protected object at which the tasks doing an RPC wait for the reply.

4 Usage Example

Fig. 2 shows a simple example of a distributed application composed of two CPUs connected by a network using the RT-EP protocol. The example code is divided into three library units: the main procedure `Remote_Caller`, the RCI package `Calculator`, and a library unit containing the task `Local_Caller_Task`. Each library unit is allocated to a different partition and assigned to a CPU as shown. The main procedure makes an RPC to the `Calculator.Add` function, through the RT-EP protocol. This call must specify the three priorities needed to make an RPC. The `Local_Caller_Task` calls the same function, but because it is located in the same CPU it will use the Local protocol, and therefore only the `RPC_Handler` priority must be specified.

The RT-GLADE configuration file that corresponds both to the distribution and to the way in which the protocols are used is as follows:

```
configuration Configuration_File is
   pragma Starter (None);
   Partition1: partition := (Calculator);
   Partition2: partition := (Remote_Caller);
   Partition3: partition := (Local_Caller);
   procedure Remote_Caller is in Partition2;
   pragma Boot_Location (("RT_EP","CPU1:1"));
   for Partition1'Self_Location use
      ((("RT_EP","CPU2:1"), ("Local","CPU2:1:Partition3")));
   for Partition2'Self_Location use
      (("RT_EP","CPU1:1"));
   for Partition3'Self_Location use
      ((("RT_EP","CPU2:3"), ("Local","CPU2:3:Partition1")));
   for Partition1’Priority use 27;
   for Partition1’Task_Pool use (8, 8, 8);
   for Partition1’Max_RPC_Replies use 4;
end Configuration_File;
```


CPU1
Partition2 (remote_caller.adb)
RT-EP:1
CPU2
Partition1 (calculator.ads)
RT-EP: 1
Local: 1
Partition3 (local_caller.adb)
Local: 3
RT-EP: 3

```
with RPC_Priorities, System.Garlic.Priorities, Calculator;

procedure Remote_Caller is
   RPC_P,IN_P,OUT_P: System.Garlic.Priorities.Global_Priority;
   ...
begin
   ...
   RPC_Priorities.Set (RPC_P, IN_P, OUT_P);
   Sum := Calculator.Add (X, Y);
   ...
end Remote_Caller;
```

```
package Calculator is
   pragma Remote_Call_Interface;
   function Add (N1: in Integer;
                 N2: in Integer)return Integer;
   ...
end Calculator;
```

```
with RPC_Priorities, System.Garlic.Priorities, Calculator;
   ...
   task Local_Caller_Task body is
      RPC_P : System.Garlic.Priorities.Global_Priority;
      ...
   begin
      ...
      RPC_Priorities.Set (RPC_P);
      Sum := Calculator.Add (X, Y);
      ...
   end Local_Caller_Task;
```

Fig. 2 Simple example using RT-GLADE

The partition attribute `Self_Location` applied to `Partition2` determines that it has two communication protocols (RT-EP and Local) and the channels used. Although in this case it is not necessary to specify a partition link because there is only one network protocol available, `Partition2` explicitly requests from `Partition1`: "if you send a message to me, you have to use the RT-EP protocol and reception channel 2". In this example, it can be seen also that the number of RPC handler tasks in `Partition1` is statically established to a value of 8, the initial priority of these tasks and of the accept handlers is set to 27, and the maximum number of simultaneously pending replies is set to four.

5 Evaluation

The real-time modelling of an application using RT-GLADE may be done by using the models described in [8] for RT-EP and [9] for the RPCs. The measurement of the time parameters specified in the model depend on the particular platform and thus their absolute values are not representative. In this section we will show the response times of a simple example that shows how the full control of the individual RPC and message priorities lets us achieve better results than with the two priority models used

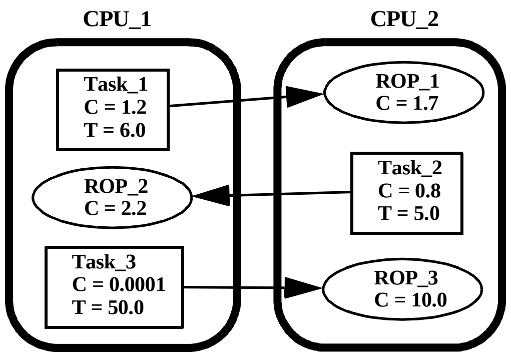


Fig. 3 Architecture of an example application

in the original GLADE. Fig. 3 shows the architecture of a simple application with two processors and three tasks (numbered 1 to 3) executing remote operations (named ROP 1 to 3) in the opposite processor. The worst-case execution times (C) and periods (T) are shown in the figure. The application is divided in two partitions, one for each processor. `Task_1` and `Task_2` have hard real time requirements with deadlines equal to their respective periods; `Task_3` has no real-time requirements, and thus is executed at a low priority level. Time units are in seconds. Since the original GLADE does not consider priorities in the network, we have used a single priority level for all the messages in the network, to make a fairer comparison. The time measurements have been made on the same platform, so that the differences are only due to the priority management scheme.

Table 1 shows the worst-case response times observed for `Task_1` and `Task_2` with the following priority schemes: RT-GLADE with some specific assignment that is not possible in GLADE; and the two possibilities in GLADE: the Client Propagated model and the Server Declared model, each with different possible priority combinations.

Table 1. Comparison of the different priority schemes

Priority Schemes	Task_1 Prio.	ROP_1 Prio.	Task_2 Prio.	ROP_2 Prio.	Task_1 WCRT (s)	Task_2 WCRT (s)
RT-GLADE	High	Medium	High	Medium	4.26	4.81
Client Propagated	High	High	Medium	Medium	3.34	Unbounded
	Medium	Medium	High	High	8.17	3.44
Server Declared	High	Medium	High	Medium	16.81	6.18
	High	High	Medium	Medium	10.85	Unbounded
	Medium	Medium	High	High	Unbounded	3.44

We can see how the timing requirements of this particular example are only met under RT-GLADE. The results for the Server Declared scheme are especially bad because the remote operation `ROP_3` invoked by the lower priority task is forced to execute at the same priority as `ROP_1`. The unbounded response times are obtained in some cases because the task is unable to complete before its period and work continues to accumulate with increasing response times.

6 Conclusions and Further Work

We have presented RT-GLADE, which is a new version of GLADE, the GNAT implementation of the Ada 95 Distributed Systems Annex, with better real-time capabilities. The new implementation continues to conform to the DSA and is also offered as free software under the GNU licence. To achieve predictable and controllable timing behaviour we have modified the priority scheme used by the implementation to allow the application to fully specify the priorities of the communication messages and of the RPC handlers. We have also eliminated the need for dynamic task creation and some sources of non uniform overheads.

In addition to changes to the middleware, we had to add network services capable of providing real-time behaviour. We have used RT-EP, a real-time protocol based on standard Ethernet hardware. In addition, because some calls to procedures declared as remote may result in calls to the same processing node, a Local protocol has been included for efficiency and generality purposes.

More work needs to be done in the current implementation to remove the unused features of GLADE that are still coded in RT-GLADE. It is also important to change the way in which GLADE starts up the system, in order to make it simpler and more controllable, although the current implementation does not affect the real-time behaviour once the system has been initialized. Finally, as a very specific issue, a mechanism to catch the exceptions raised by an APC would be useful for the applications using RT-GLADE.

In summary, by using RT-GLADE we can build applications that can guarantee meeting their real-time requirements. As a conclusion, we have shown how the changes and additions that we proposed in [6] for the DSA are viable and complete, and can lead to a Real-Time Distributed Systems Annex in Ada with moderate effort. It would be extremely useful for the Ada Language to incorporate these changes and additions to the standard DSA. This extension could facilitate using Ada's distribution services in real-time systems, and would help in keeping Ada as the reference language for real-time systems.

References

[1] Ada-Core Technologies, Ada 95 GNAT Pro, `http://www.gnat.com/`

[2] M. Aldea and M. González. “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe 2001, Leuven, Belgium, in Lecture Notes in Computer Science, LNCS 2043, May 2001.

[3] Y. Kermarrec, L. Pautet, and S. Tardieu. “GARLIC: Generic Ada Reusable Library for Interpartition Communication”. Proceedings of Tri-Ada’95, Anaheim, California, USA, ACM, 1995.

[4] M. González Harbour, J.J. Gutiérrez, J.C. Palencia and J.M. Drake. “MAST: Modeling and Analysis Suite for Real-Time Applications”. Proceedings of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[5] J.J. Gutiérrez García, and M. González Harbour. “Prioritizing Remote Procedure Calls in Ada Distributed Systems”. Proceedings of the 9th International Real-Time Ada Workshop, ACM Ada Letters, XIX, 2, pp. 67–72, June 1999.

[6] J.J. Gutiérrez García, and M. González Harbour. “Towards a Real-Time Distributed Systems Annex in Ada”. Proceedings of the 10th International Real-Time Ada Workshop, ACM Ada Letters, XXI, 1, pp. 62–66, March 2001.

[7] Jane W. S. Liu. “Real-Time Systems”. Prentice Hall, 2000.

[8] J.M. Martínez, M. González Harbour, and J.J. Gutiérrez. “RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel”. Proceedings of the 2nd International Workshop on Real-Time LANs in the Internet Age, RTLIA 2003, Porto (Portugal), July 2003.

[9] J. Javier Gutiérrez, José M. Drake, Michael González Harbour, and Julio L. Medina. “Modeling and Schedulability Analysis in the Development of Real-Time Distributed Ada Systems”. Proceedings of the 11th International Real-Time Ada Workshop, ACM Ada Letters, Vol. XXII, No. 4, pp. 58–65, December 2002.

[10] Object Management Group, “Realtime CORBA Joint Revised Submission”. OMG Document orbos/99-02-12 ed., March 1999.

[11] L. Pautet and S. Tardieu, “Inside the Distributed Systems Annex”. Proceeding of the Intl. Conf. on Reliable Software Technologies, Ada-Europe’98, Uppsala, Sweden, in LNCS 1411, Springer, pp. 65–77, June 1998.

[12] L. Pautet, T. Quinot, and S. Tardieu. “CORBA & DSA: Divorce or Marriage”. Proc. of the International Conference on Reliable Software Technologies, Ada-Europe’99, Santander, Spain, in Lecture Notes in Computer Science No. 1622, pp. 211–225, June 1999.

[13] L. Pautet and S. Tardieu. “GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems”. Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC'00), Newport Beach, USA, March 2000.

[14] IEEE Std. 1003.13-2003. Information Technology -Standardized Application Environment Profile- POSIX Realtime and Embedded Application Support (AEP). The Institute of Electrical and Electronics Engineers.

[15] Scott Moody. “Object-Oriented Real-Time Systems Using a Hybrid Distributed Model of Ada 95's Built-in DSA Capability (Distributed Systems Annex-E) and CORBA”. Proceedings of the 8th International Real-Time Ada Workshop, ACM Ada-Letters, XVII, 5, pp. 71–76, September-October 1997.

[16] S. Tucker Taft, and R.A. Duff (Eds.). “Ada 95 Reference Manual. Language and Standard Libraries”. International Standard ISO/IEC 8652:1995(E), in LNCS 1246, Springer, 1997.