Towards a Real-Time Distributed Systems Annex in Ada

By: José Javier Gutiérrez García and Michael González Harbour
Departamento de Electrónica y Computadores
Universidad de Cantabria
39005- Santander, SPAIN
{gutierjj, mgh}@unican.es

Abstract[1]

In this paper we address the problem of combining two issues that are standardized separately in two Ada 95 annexes: Real-Time Systems (Annex D) and Distributed Systems (Annex E). With these annexes it is possible to build applications with real-time requirements, or alternatively distributed applications; but real-time distributed applications are not directly supported. In this paper we propose extensions to the Distributed Systems Annex that would provide support for developing distributed real-time applications in Ada. The paper proposes an interface for assigning priorities to the messages in the interconnection networks, and also to the tasks executing remote operations, in a way that avoids priority inversions. It also addresses the issues of configuration of the RPC handler tasks and non blocking asynchronous remote procedure calls.

Keywords: *Ada, Real-Time, Distributed Systems, Priority Optimization, Ada Distributed Systems Annex*

1. Introduction

Developers of distributed systems can use Ada 95 to support the functional aspects of their systems. However, if these systems have to meet real-time requirements then the timeliness is as important as functionality. The Distributed Systems Annex (DSA) of Ada 95 [5] provides a flexible way for distributing Ada programs in a multiple-processor platform. This distribution is based on the concepts of program partitions, and remote procedure calls (RPCs). But the language does not provide interfaces nor semantics that would enable using the DSA for distributing applications with real-time requirements. At the time Ada 95 was standardized the mixture of distributed and real-time systems was considered to be insufficiently known, and thus the standard did not attempt its specification. In any case, the standard is a good starting point for further investigation into these issues.

Previous works explored aspects of the distribution of real-time applications, like the prioritization of remote procedure calls [1], and pointed out new issues that should be covered by a new DSA with real-time characteristics [2]. This paper elaborates on these issues and provides proposals regarding other important issues that relate real-time and the DSA. The Ravenscar profile is not addressed in this paper because the mechanisms required to implement the proposed real-time distributed systems annex are far more complex than the simple tasking model provided in that profile.

One of the issues pointed out in [2] was the use of CORBA and specially the new real-time CORBA standard [3] as an alternative to a new DSA extended with real-time capabilities. However, we think that there is large value in having solutions to support real-time distributed applications within the Ada standard, which would provide a much simpler and potentially more efficient mechanism for writing distributed applications, given the complexity of CORBA [4]. In any case, the issues addressed in real-time CORBA can be a good reference to study the requirements and features that a new real-time DSA should support. CORBA will always be available for developing distributed applications in a multi-language environment.

The paper is organized as follows. In Section 2 we provide a quick review of the requirements that were used for the development of the real-time CORBA standard. Sections 3 to 5 describe the issues identified in Ada real-time distributed systems and the proposed solutions to them; in particular, the execution model of RPC-receivers, the priorities of RPC handler tasks and of messages in the networks (including the mapping of priorities), the

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC99-1043-C03-03

asynchronous RPC blocking, and additional documentation for the DSA. Finally, in Section 6 we give our conclusions.

2. RT-CORBA Requirements

Although the client-server model of CORBA differs from the Ada model based on RPC's, many of the issues that are encountered when trying to define a real-time DSA have been faced also during the standardization of Real-Time CORBA. Consequently, we will make a quick review of the main requirements for that standardization process:

- *Define a Schedulable Entity*: an activity is defined as a design concept that uses threads provided by an underlying OS to implement the concept. Optionally, there is a fixed priority scheduling service to help the application programmers schedule activities.
- *Interfaces for priority control of Schedulable Entities*: CORBA priority is defined as a universal, platform-independent scheme. A mapping interface is defined to map CORBA priority to or from the native priority scheme of a given scheduler.
- *Mechanism for propagating client priority to the server*: a Priority Model Policy is defined to determine the priority at which a server handles requests from clients. It has two models: *Client_Propagated* (the server honours the priority of the request set by the client), and *Server_Declared* (the server handles requests at a priority declared at the time of the server object creation).
- *Mechanism for avoiding or bounding priority inversion*: mutex interface, policies for specifying and configuring communication protocols, a threadpool abstraction used to manage threads of execution on the server side, policies (that let the client set up multiple transport connections and specify use of non-multiplexed connections), and a server-side Priority Transform mechanism to implement priority protocols.
- *Define "resources" for the purpose of resource management*: threads, threadpools, transport connections and request buffers.
- *Mechanism for management of resource allocation*: a mutex interface to coordinate contention, management of thread priorities, and an API for threadpool management, protocol policies and the *explicit_bind* operation to manage transport connections.

In addition, there are some optional requirements that are also important for achieving real-time behavior: timeouts, interface for installation of user-provided transport protocols, interaction protocol between real-time and non real-time objects, and runtime interfaces for a Schedulable Entity.

3. RPC Handler and Message Priorities

The Ada 95 DSA requires that "the implementation of the RPC-receiver shall be reentrant, thereby allowing concurrent calls on it from the PCS (Partition Communication Subsystem) to service concurrent remote subprogram calls into the partition." It also requires the implementation to document "whether the RPC-receiver is invoked from concurrent tasks." In addition, it provides the following implementation advice: "Whenever possible, the PCS on the called partition should allow for multiple tasks to call the RPC-receiver with different messages and should allow them to block until the corresponding subprogram body returns". We will use the term *RPC handler* for each of these tasks that execute an RPC-receiver.

For real-time applications it is very important that this implementation advice is followed, because there will be RPC execution requests with different degrees of urgency, and thus will need to be served by tasks of different priorities. In order to be able to predict and bound the response times to RPC execution requests it is necessary to be able to specify the priorities at which the different RPC handlers execute, as well as the priorities at which the messages are transmitted in the network, if supported. This section discusses the different aspects that influence the priority scheme of RPCs, by elaborating on the results of [1].

3.1. Priority Types

The type `System.Priority` is not pure because it represents relative priorities that are meaningful only to one processor in the distributed system. In a heterogeneous system, other processors may have different ranges for their `System.Priority` type. In addition, the priorities on the communications network may be wholly different. It is necessary to have a global priority type that can represent priorities that are meaningful across the whole distributed system, and mapping functions to convert values of the global priority type to the native priority types in each processor or network.

For this purpose, we can create the package `Global_Priorities` with a type `Global_Priority` representing a value with a global meaning across the distributed system. For each CPU and network, a mapping function would translate a value of this global priority type to a value of `System.Priority` or of the network priority appropriate for that resource.

```
package Ada.Global_Priorities is
   pragma Pure (Global_Priorities);
   type Global_Priority is
      range implementation-defined;
end Ada.Global_Priorities;
```

The body of the mapping functions should be provided by each implementation of the PCS. A different implementation should be available if needed for each processor and each network in the system. The specification of the mapping functions should be standard, though, for portability. The native priority types for the processors have a standard name (`System.Priority`), but the native priority types for the networks don't. Therefore, it is necessary to consider two groups of mapping functions, one for the processors, and one for the networks.

Consequently, the Real-Time Distributed Systems Annex should specify the following mandatory package:

```
with System;
package Ada.Global_Priorities.Mapping is
   function To_Global_Priority (
      The_Priority : System.Priority)
      return Global_Priority;
   function To_Native_Priority (
      The_Priority : Global_Priority)
      return System.Priority;
end Ada.Global_Priorities.Mapping;
```

And it should give an implementation advice recommending that the rest of the mapping functions between the type `Global_Priority` and the network priority types should be included in package `Global_Priorities.Mapping` and should use the same naming scheme of that package.

3.2. Initial Priority of the RPC Handlers

If the initial priority of the RPC handler is left unspecified, it could easily cause priority inversion. For example, if the RPC handler's priority is set initially to a medium level [6], or if it is set to a low value because of keeping the priority of a previous execution, a high priority request that is about to being serviced by that RPC handler may have to wait for a long time, blocked by some other tasks that may in fact have less priority than the request.

To solve the problem of the initial priority of the RPC handler, we can just set it to the maximum priority (`System.Priority'Last`). In this way, the RPC handler will start at a very high priority, and will immediately read the desired priority from the stream and set its priority to the appropriate value. Then it will call the RPC-receiver as specified in the DSA. Later, when the RPC-receiver returns, the RPC handler will send back the appropriate message (in case of a synchronous RPC), and then it will set its own priority back to the highest level just before blocking itself waiting for the next RPC request. No priority inversion occurs using this scheme.

A better approach would be to let the application set the value of this initial priority. In this way, it could choose an optimum value that would not cause priority inversions for RPC's, while keeping a lower overall blocking for very high priority tasks in the server's CPU.

One way to set the initial priority of each RPC handler would be to have a configuration pragma:

```
pragma Initial_RPC_Handler_Priority
       (Global_Priority);
```

This pragma specifies the initial priority of the RPC handlers. Once the handler is servicing an RPC, its priority is set to the priority specified for that RPC, according to the discussion in Subsection 3.3. When the RPC is completed, the priority of the handler is set back to the initial priority.

The default value for the initial priority for RPC handlers should be `System.Priority'Last`.

3.3. Priorities of RPC Handlers

In order to achieve optimum response times, the application should have the ability to set the priorities of the calling tasks and of the RPCs independently [1]. In addition, if a priority-based communication system is used, the priorities of the outgoing and the incoming messages should be specifiable.

One problem with RPCs is that they may be invoked many times, from many different tasks with different timing requirements, and thus each invocation may need its own priority. Therefore a priority pragma attached to the specification of the remote procedure is not appropriate for setting the different priorities involved in the RPC. We need an operation that the user can invoke before making an RPC. This operation should be in a package that is visible from the application. Package `System.RPC` does not seem appropriate, since it is not intended for direct use from the application. Therefore a package like the following could be created:

```
with Ada.Global_Priorities;
use Ada.Global_Priorities;
package Ada.RPC_Priorities is
  procedure Set
   (RPC_Handler : in Global_Priority);
  procedure Set
   (RPC_Handler,
    Outgoing_Message: in Global_Priority);
```

```
  procedure Set
   (RPC_Handler,
    Outgoing_Message,
    Incoming_Message: in Global_Priority);
  procedure Get
   (RPC_Handler,
    Outgoing_Message,
    Incoming_Message: out Global_Priority);
end Ada.RPC_Priorities;
```

Procedure `Set` in the above package would set the priority or priorities used for future RPCs or APCs issued by the calling task. These priorities would be 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 could be implementation defined. Procedure `Get` would return the current values of the RPC priorities.

3.4. Configuration of the Pool of RPC Handlers

In order to manage the pool of RPC handlers in a portable way, a configuration mechanism of the pool should be available to create a minimum static number of tasks at initialization time. It could also be interesting to be able to add RPCs dynamically, to allow more flexibility for those applications that could tolerate dynamic creation of more tasks. Therefore, the following issues should be considered for the configuration of the pool:

- *Static or dynamic size of the pool*: some systems need the flexibility of dynamic size, while others need the predictability of static size. A maximum number of tasks is also interesting for systems with a dynamic pool size.
- *Initial number of RPC handlers*: This number should be configurable. If the size is static, this would be the final number. If dynamic, the size could grow dynamically up to the maximum.
- *Explicit or implicit creation of new RPC handlers*: Increasing the number of RPC handlers in the case of a dynamic size pool could be done explicitly, via a procedure, or implicitly, when an RPC arrives and there are no RPC handlers available. Given that it is difficult for a server to anticipate when new tasks will be necessary, the implicit creation looks more useful. Therefore, an implementation advice could be specified to this effect.
- *Queuing of pending RPC requests*: Given that the number of tasks in the pool may always be limited (even in the dynamic case, the system´s maximum number of tasks may be reached), it is necessary to specify how the pending RPC requests are handled. The best way is to store them in a priority queue, ordered according to the RPC priority. This would be an implementation requirement.

Consequently, one solution for this RPC handler pool configuration would be to create the following configuration pragmas:

```
pragma Maximum_RPC_Pool_Size
       (Maximum Number of RPC Handlers);

pragma Minimum_RPC_Pool_Size
       (Minimum Number of RPC Handlers);
```

If both pragmas specify the same value, this implies that the size of the pool is static. Default values for these pragmas should be implementation defined.

4. Non-Blocking Asynchronous RPC

The LRM specifies in its DSA: “The task executing a remote subprogram call blocks until the subprogram in the called partition returns, unless the call is asynchronous. For an asynchronous remote procedure call, the calling task can become ready before the procedure in the called partition returns”. In addition it says that “All forms of remote subprogram calls are potentially blocking operations”.

For real-time applications, it would be desirable to have a form of asynchronous RPC that would be non blocking. A blocking operation has the potential of introducing new context switches, possibilities for new deadlock situations, and other effects that an application may want to avoid. An asynchronous operation may need to block if the resources in the local communications driver are temporarily unavailable (for example, a queue for storing the message). For this case, a new exception could be declared, that would be raised to the application requiring a non blocking APC.

Therefore, one solution to this problem would be to declare a new pragma:

```
pragma Non_Blocking (local_name);
```

with the same effects than pragma `Asynchronous`, except that the call would be required to be non blocking, and could raise `Temporarily_Unavailable`, which would be a new exception declared in `System.RPC`.

5. Additional Documentation for the DSA

To be able to carry out the schedulability analysis of an application using the real-time DSA it is necessary to know if the implementation uses additional tasks, together with their priorities, execution activation mechanisms, and

worst-case execution times. This implies that there are new documentation requirements for the real-time DSA.

The documentation requirements are split into two parts: one part for the implementation of the compiler and run-time part of the DSA, and the other one for the implementation of the PCS, including the communication network drivers.

The documentation requirements for the PCS should also include the additional messages required to implement the protocols, and the overhead that the protocols themselves introduce for each message.

6. Conclusions

Although the real-time CORBA standard can be used to accomplish the development of real-time distributed applications, defining real-time capabilities for the Ada 95 Distributed Systems would allow the development of this kind of application in a simpler and potentially more efficient way.

This paper describes a set of extensions required to add real-time capabilities to the current DSA. The number of extensions is not very large, and it does not seem difficult to implement. Therefore, it seems that instead of creating a new annex for real-time distributed systems, the proposed extensions could be incorporated into the current DSA without much effort. The extensions would be mandatory if both the Real-Time Systems Annex and The Distributed Systems Annex were supported by an implementation.

References

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

[2] Scott Arthur Moody (Rapporteur): “Session Summary: Distributed Ada and Real-Time”. 9th International Real-Time Ada Workshop, ACM Ada Letters, XIX, 2, pp. 15-18, June 1999.

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

[4] L. Pautet, T. Quinot, and S. Tardieu. “CORBA & DSA: Divorce or Marriage?”. Intl. Conf. on Reliable Software Technologies, Ada-Europe’99, Santander, Spain, in LNCS 1622, Springer, pp. 211-225, June 1999

[5] S. Tucker Taft, and R.A. Duff (Eds.) “*Ada 95 Reference Manual. Language and Standard Libraries*”. International Standard ISO/IEC 8652:1995(E), in Lecture Notes on Computer Science, Vol. 1246, Springer, 1997.

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