Support for a real-time transactional model in distributed Ada

Héctor Pérez Tijero, J. Javier Gutiérrez, and Michael González Harbour

Grupo de Computadores y Tiempo Real, Universidad de Cantabria
39005-Santander, SPAIN
{perezh, gutierjj, mgh}@unican.es

Abstract[1]

This paper presents a proposal to integrate a generic technique to express complex scheduling and timing parameters of distributed transactions as a part of the Distributed Systems Annex of Ada (DSA). The technique allows real-time middleware implementations to change their scheduling policies for both the processing nodes and the networks with a minimal interference in the application code. The proposed mechanisms are managed by the middleware in a transparent way. The only requirement is an initial configuration operation, which can be generated automatically, and a single operation call to set an event identifier in each of the tasks that initiates a distributed transaction. The automatic process used to obtain the initial configuration of the distributed application, based on a real-time model, is also proposed. The implementation of this proposal is currently being developed.

1. Introduction

The Distributed Systems Annex of Ada (DSA) [10] enables developers to build flexible distributed systems. The distribution is based on the concept of program partitions and Remote Procedure Calls (RPCs). Inherent aspects of distributed systems such as concurrency and shared variables are directly supported by the language itself leading to a consistent environment for distributed applications. As a separate feature, real-time issues are supported in Ada by the Real-Time Systems Annex [10]. However, a clear interaction between both annexes has not been defined. This implicit detachment makes it harder for Ada users to build distributed systems with real-time requirements.

The Ada language includes a variety of scheduling policies as part of its Real-Time Systems Annex such as Fixed Priorities, EDF, or Round Robin. However, the evolving complexity of real-time systems has lead to the need for using more sophisticated scheduling techniques, capable of simultaneously satisfying multiple types of requirements such as hard real-time guarantees and quality of service requirements in the same system. To better handle the complexity of these systems, scheduling services of a higher level of abstraction are being designed, usually based on the concept of resource reservations or service contracts [1]. Any of these heterogeneous scheduling policies should be supported by the middleware in a flexible and standard approach as is proposed in [5].

One of the main issues in real-time applications is the ability to guarantee whether deadlines will be met or not. The schedulability analysis techniques applied to distributed systems have evolved a lot both for fixed priorities and EDF leading to decreasing pessimism in the calculation of response times [3][8][9]. Our research group has developed MAST [2], a tool suite to perform the schedulability analysis and the assignment of scheduling parameters to a real-time application. MAST is based on an event-driven model of a distributed real-time system in which execution and communications flows are specified together with the timing

1. This work has been funded in part by the Spanish Ministry of Science and Technology under grant number TIN2008-06766-C03-03 (RT-MODEL), and by the IST Programme of the European Commission under project FP6/2005/IST/5-034026 (FRESCOR). This work reflects only the author's views; the EU is not liable for any use that may be made of the information contained herein.

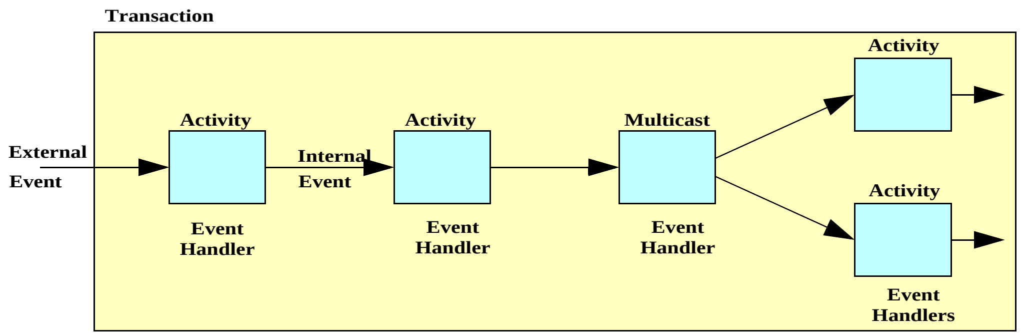

Figure 1: Transactional model

requirements through an element called transaction. The transactional model [3] has traditionally been used within analysis techniques to compute response times in distributed systems. Under this model, events arriving at the system trigger the execution of activities (in processors or networks) that may generate additional events and thus activities, making up a chain of events and activities with end-to-end timing requirements. We will discuss how this model can be integrated into DSA in order to support real-time distributed systems.

This paper aims to define a standard interface to allow the distribution middleware to support real-time capabilities embedded in the transactional model. Furthermore, this interface should be independent of the scheduling policy and target non intrusive programming. This would require:

- Hiding the details related to scheduling from the software engineers developing the functional parts of the application by separating the logic of the application and the real-time aspects.
- Allowing the use of interchangeable scheduling policies and the free (possibly automatic) assignment of their scheduling parameters.
- Providing an infrastructure for CASE tools (e.g., MAST) to enable a direct mapping between the model and the real-time configuration of the application.
- Trying to facilitate the building of distributed applications with real-time requirements making the process as similar as possible to a non-distributed application (Ada philosophy [10]).

This document is organized as follows. Section 2 introduces the previous work done to support the transactional model in distributed Ada according to the DSA. Section 3 analyses in detail the new approach, describing aspects of scheduling and management of the remote calls through the proposed new interfaces. Section 4 explains how to automatise the configuration of a distributed application through the proposed APIs, using a real-time model as a reference. An example of use is included in Section 5. Finally, Section 6 draws the conclusions.

2. Related work

Traditional standards used in the development of distributed applications [6][10] follow different architectures (e.g., client-server or distributed objects) to build such systems. It is easy to show how those architectures could be easily modelled and analysed using the transactional model. Additionally, real-time distribution middleware is aimed to support different scheduling policies with heterogeneous parameters [7].

A distributed transaction is defined to relate parts of an application consisting of multiple threads executing code in multiple processing nodes, and exchanging messages through one or more communication networks. This behaviour is illustrated in Figure 1. In a transaction, events arriving at the system trigger the execution of activities, which can be either task jobs in the processors or messages in the networks. These activities, in turn, may generate additional events that trigger other activities, and this gives rise to a chain of events and activities, possibly with end-to-end timing requirements.

In [5] the optimisation of the response times of a distributed transaction was discussed and a solution to support the event-driven transactional model according to the DSA was proposed. Under this approach, each activity involved in a transaction should be configured independently and the scheduling parameters must be defined not only for processing tasks but also for the message streams in the communication networks. Two kinds of schedulable entities are defined:

- For the processing nodes, the handler tasks intended to execute remote calls. These handlers were created explicitly with the appropriate scheduling information.
- For the network, the endpoints or communication points are used to transport messages through the network with specific scheduling parameters associated with the endpoint. The endpoints are also created explicitly with static scheduling parameters.

Complete support for transactional model was provided by a single parameter, called *Event_Id*, which represents the event that triggers the execution of an activity within its transaction. Only this parameter was sent through the network avoiding the transmission of more complex data structures that could lead to inefficient usage of the network [5].

An initial interface was described to customize the communication layer and handler tasks. This API considered four primitives:

- *Create_Query_Send_Endpoint* creates a communication endpoint to send messages with specific scheduling parameters.
- *Create_RPC_Handler* creates a receive endpoint and an RPC handler task that will listen for incoming requests at this endpoint.
- *Create_Reply_Send_Endpoint* creates a send endpoint for sending the reply message of a synchronous remote call.
- *Create_Reply_Receive_Endpoint* creates a receive endpoint where the application task can wait for the reply message resulting from a remote call.

Each endpoint created by this interface is identified by an *Event_Id* parameter that the application should specify explicitly at the beginning of the transaction. This *Event_Id* may be modified by the application code at any point of the transaction.

3. Transactional model support

Compared to the approach shown in Section 2, this paper proposes an extended solution with the following new features:

- Separate interfaces for RPC handler tasks and communication endpoints to ease the integration of new, possibly different, scheduling policies both in processing nodes and networks.
- The receive endpoint where the RPC Handler waits is not associated to a particular *Event_Id*, to allow sharing RPC handlers by several transactions.
- An adapted transactional model support with event transformations to avoid the management of real-time aspects within the application code. This management is done at runtime using configuration information defined at initialization time.
- A complete interface to facilitate the configuration of a real-time distributed application using CASE tools.

Our new proposal defines an interface to allow the middleware and the applications to manage event associations. The idea is similar to that presented in [5] and the *Event_Id* is the only scheduling parameter sent through the network. However, the new proposal allows a more general approach with a more flexible support of the transactional model. The general approach is shown in Figure 2. In the transactional model, external events trigger the transactions, and the setting of an identifier of those events is the only operation that our model requires the application code to perform. The rest of the transaction elements, including the communication endpoints, the handler tasks, and all the scheduling parameters, can be described as a part of a

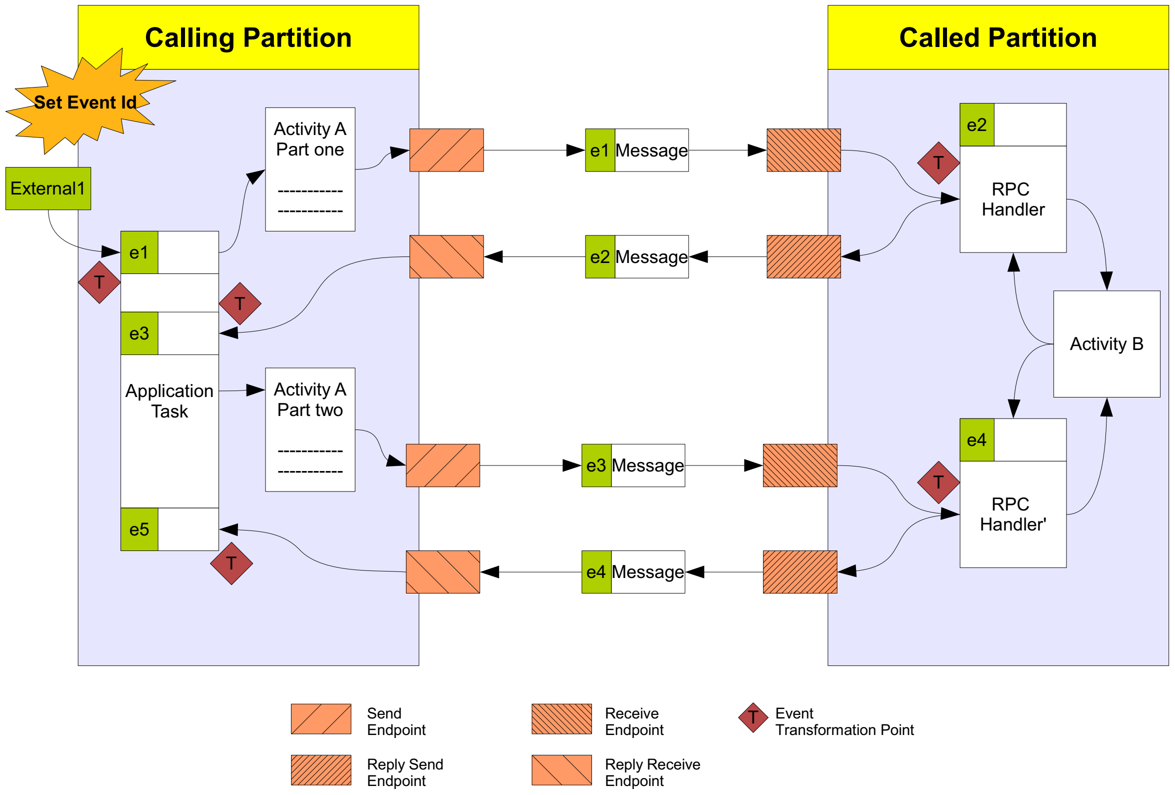

Figure 2: Event flow in synchronous remote calls

configuration operation. Once the task at the beginning of the transaction has set the initial event, all the subsequent activities (including the task's activity itself) are scheduled according to the associated event at each moment. This event is set internally by the middleware at the transformation points defined at the following actions (see Figure 2):

- Setting the initial event (i.e., external event).
- The start of the execution of an RPC Handler.
- The reception of the reply by a task waiting for a synchronous RPC.

Appropriate scheduling parameters will be used according to the *Event_Id* for each application task or RPC Handler. This new event management is particularly useful in transactions that invoke the same remote operation two or more times, and which may require different scheduling parameters at each call. This case is illustrated in Figure 2 where the application task makes several requests to the same remote service. Setting the initial *Event_Id* parameter is sufficient to enable the middleware to identify the particular point in the transaction and therefore the scheduling parameters required in each case. This model also allows an RPC Handler to be shared among different transactions executing in turn with different scheduling parameters, which is interesting to reduce the total number of tasks in the system.

The work in [5] stated that all resources involved in a transaction had to be specified through a single generic interface. However, our current proposal defines a set of Ada packages in order to be integrated into the DSA.

Firstly, we have defined a package to include identifiers for the basic elements to be set up in a distributed system: communication networks, send/receive ports and nodes.

```
package Ada.Distributed is
   pragma Pure;
   type Port_Id is range implementation-defined;
   type Node_Id is range implementation-defined;
   type Network_Id is range implementation-defined;
end Ada.Distributed;
```

A second package contains the identifiers of common elements in the proposed transactional model for real-time environments:

```
package Ada.Distributed.Real_Time is
   pragma Pure;
   type Send_Endpoint_Id is private;
   type Receive_Endpoint_Id is private;
   type Event_Id is range implementation-defined;

private
   type Send_Endpoint_Id is ...;
   type Receive_Endpoint_Id is ...;
end Ada.Distributed.Real_Time;
```

The *Send_Endpoint_Id* and *Receive_Endpoint_Id* are the types used to identify endpoints, which are used as links to enable the communication among different nodes. The type *Event_Id* is used to attach an identifier which associates scheduling parameters to the schedulable entities within the transactions (send endpoints or handler tasks) and to identify the endpoint where a task performing an RPC should await the reply.

Finally, three APIs have been designed, all of them children of *Ada.Distributed.Real_Time*:

- *Network_Scheduling*: an API to contain the endpoint configuration and assign the scheduling parameters for the messages in the communication networks.
- *Processing_Nodes*: an API to contain the operations to configure the RPC handler tasks and assign their scheduling parameters.
- *Event_Management*: an API to support the description of the transactional architecture via event transformations.

3.1. Network Scheduling Interface

The DSA does not explicitly consider scheduling policies in communication networks although they have a strong influence in the response time of a remote subprogram call. Meeting real-time requirements requires the networks to be scheduled with appropriate techniques. Furthermore, there are no mechanisms defined in the DSA to enable the transmission of scheduling parameters among nodes, which is needed to establish reasonable bounds on the end-to-end response times. As shown in [4], the integration of these aspects is necessary to achieve real-time behaviour.

The approach described in Section 2 tries to solve these limitations by making the communication endpoints visible, and by associating scheduling parameters to the messages sent through them. Here we follow a similar approach, but a new API has been designed to allow for more flexibility:

```
package Ada.Distributed.Real_Time.Network_Scheduling is

   type Message_Scheduling_Parameters is abstract tagged private;
   type Message_Scheduling_Parameters_Ref is access all
          Message_Scheduling_Parameters'Class;

   procedure Create_Receive_Endpoint
     (Net         : Network_Id;
     Port         : Port_Id;
     Endpoint     : out Receive_Endpoint_Id) is abstract;
```

```
      procedure Create_Send_Endpoint
         (Param       : Message_Scheduling_Parameters_Ref;
         Dest_Node    : Node_Id;
         Event        : Event_Id;
         Net          : Network_Id;
         Dest_Port    : Port_Id;
         Endpoint     : out Send_Endpoint_Id) is abstract;

      procedure Create_Reply_Receive_Endpoint
         (Net         : Network_Id;
         Event_Sent   : Event_Id;
         Port         : Port_Id;
         Endpoint     : out Receive_Endpoint_Id) is abstract;

      procedure Create_Reply_Send_Endpoint
         (Param       : Message_Scheduling_Parameters_Ref;
         Dest_Node    : Node_Id;
         Event        : Event_Id;
         Net          : Network_Id;
         Dest_Port    : Port_Id;
         Endpoint     : out Send_Endpoint_Id) is abstract;

      procedure Destroy_Receive_Endpoint
         (Endpoint    : Receive_Endpoint_Id) is abstract;

      procedure Destroy_Send_Endpoint
         (Endpoint    : Send_Endpoint_Id) is abstract;

   private
      type Message_Scheduling_Parameters is abstract tagged ...;
   end Ada.Distributed.Real_Time.Network_Scheduling;
```

Extensions of the *Message_Scheduling_Parameters* tagged type will contain the specific network scheduling parameters that must be associated to a specific endpoint. Furthermore, each scheduling policy must implement operations to map its own scheduling parameters (e.g., priorities, deadlines, or contract-based parameters) onto extensions of this private type. This API will be used at the configuration phase of an application. A description of the primitive operations for the network configuration follows:

- *Create_Receive_Endpoint* creates an endpoint in a particular network to listen for incoming requests. Unlike the interface provided in [5], this endpoint is not directly associated to an event identifier. This model is more flexible because a receive endpoint could process different transactions depending on the incoming events. This behaviour will be illustrated in Section 5. Another important difference is that the current interface separates the definition of a handler task from the receive endpoint, leading to a more flexible approach that allows different scheduling policies for processing nodes and networks.
- *Create_Send_Endpoint* creates a new endpoint to send requests through a particular network with specific scheduling information associated to the provided event identifier.
- *Create_Reply_Receive_Endpoint* creates an endpoint in a particular network to listen to replies to a synchronous remote call. The *Event_Id* is required by the processing task to wait for a reply once the request is performed. Therefore, this parameter links a send endpoint with its associated receive reply endpoint.
- *Create_Reply_Send_Endpoint* creates a communication endpoint to send replies through a particular network in a synchronous request. This endpoint will hold specific scheduling information associated to the event identifier provided.

Finally, corresponding operations are provided to destroy the created endpoints.

3.2. Processing Node Scheduling Interface

The DSA requires servicing remote subprograms calls concurrently within a partition. This rule is essential for real-time distributed systems where the arrival of remote requests with different degrees of urgency is frequent. Handler tasks are responsible for awaiting arriving requests and processing them. Appropriate scheduling parameters can be associated to the handler tasks by means of the following API:

```
package Ada.Distributed.Real_Time.Processing_Node_Scheduling is

   type Task_Scheduling_Parameters is abstract tagged private;
   type Task_Scheduling_Parameters_Ref is access all
          Task_Scheduling_Parameters'Class;
   type Handler_Id is range implementation-defined;

   procedure Create_RPC_Handler
     (Default_Params    : Task_Scheduling_Parameters_Ref;
     Endpoint           : Receive_Endpoint_Id;
     RPC_Handler        : out Handler_Id) is abstract;

   procedure Destroy_RPC_Handler
     (RPC_Handler       : Handler_Id) is abstract;

   procedure Set_Event_And_Sched_Params_Association
     (Params            : Task_Scheduling_Parameters_Ref;
     Event              : Event_Id) is abstract;

   function Current_Handler_Id return Handler_Id;

private
   type Task_Scheduling_Parameters is abstract tagged ...;
end Ada.Distributed.Real_Time.Processing_Node_Scheduling;
```

For the processing nodes, tagged type *Task_Scheduling_Parameters* should be extended with appropriate data for the scheduling policy supported (e.g., deadlines for EDF or more complex parameters for contract-based scheduling). Similarly to the network interface, supported policies must implement operations to map their own scheduling parameters onto this private type.

As we stated before, handler tasks and communication endpoints should be configured separately to make it possible to use different scheduling policies in the processors and in the networks, in contrast to the previous proposal [5] which required using the same policy in both kinds of schedulable resources. Package *Processing_Node_Scheduling* defines the following three primitives that are used to configure RPC handler tasks:

- *Create_RPC_Handler* creates a handler task for proccessing incoming requests arriving at a particular endpoint. Note that a single RPC handler task could process several requests matching different transactions, since it is just bound to a particular endpoint and not to a single event.
- *Destroy_RPC_Handler* destroys the RPC handler task.
- *Set_Event_And_Sched_Params_Association* links the specified event to the scheduling parameters provided.

Finally, the middleware will require a primitive to internally manage the scheduling parameters associated with an RPC handler. This primitive is only used by the middleware, so a new package is provided to separate the application user interface from the middleware's one:

```
package Ada.Distributed.Real_Time.Processing_Node_Scheduling.Internals
is
   procedure Update_Scheduling_Parameters
     (Params      : Task_Scheduling_Parameters_Ref;
      RPC_Handler : Handler_Id := Current_Handler_Id)
      is abstract;
end Ada.Distributed.Real_Time.Processing_Node_Scheduling.Internals;
```

Update_Scheduling_Parameters will update scheduling parameters for a selected RPC Handler, therefore allowing different architectures to process remote requests (e.g., the middleware could use the Leader/ Followers [11] or Half-Sync / Half-Async [12] patterns).

3.3. Event Management Interface

This API supports the connection among the different parts of a transaction providing operations to set associations between pairs of events.

```
package Ada.Distributed.Real_Time.Event_Management is

   procedure Set_Event_Association
      (Input_Event   : Event_Id;
      Output_Event   : Event_Id);

   function Get_Event_Association
      (Input_Event  : Event_Id) return Event_Id;

   procedure Set_Event_Id (New_Event : Event_Id);

   function Get_Event_Id return Event_Id;

end Ada.Distributed.Real_Time.Event_Management;
```

Final users should configure the event flow within a transaction, and the middleware will be in charge of automatically setting the appropriate event at the transformation points of the remote call. For this purpose the proposed API defines the following operations:

- *Set_Event_Association* links the input event of an activity to the output event. Middleware will use this link to select an appropriate output event at the specified transformation points.
- *Get_Event_Association* returns the output event associated to the specified input event.

Finally, additional operations are included to allow the application and middleware to handle event identifiers (*Set_Event_Id* and *Get_Event_Id* operations).

4. Automatic generation of the configuration operations

The creation and configuration of communication endpoints and handler tasks, and the assignment of scheduling parameters to them, may be seen as a complex task that requires the application developer to add a lot of code to an already complex application. However, most of the work can be done at initialization time, and all the information needed can be automatically obtained from a real-time model of the transactions involved. From this model it is possible to make a transformation to automatically generate the configuration code that needs to be run at initialization time. Following this approach, the application developer would only need to include in its code a simple *Set_Event_Id* call to set the initial event that triggers the transaction.

Any application with real-time requirements requires a model of its real-time behaviour in which the operations, tasks, messages, triggering events and their interactions and deployment on a particular hardware platform are explicitly described. This model allows the application developer to perform a real-time analysis in which the timing requirements of the applications can be validated. This model can be generated simply for the purpose of analysis, or it could be obtained from design information of the application developed, for instance, with the MARTE UML profile for embedded real-time systems [13].

In our research group we have developed MAST [2] as a model to represent the real-time behaviour of an application. The MAST model contains descriptions of the execution and communications platform, the concurrent architecture of the application, its operations including any synchronization, and the transactions that describe the flow of events in the system. The MAST model of an application may be generated from design information written in the MARTE UML profile. From the model it is possible to perform schedulability analysis as well as automatic assignment of scheduling parameters, or sensitivity analysis.

The MAST model contains most of the information that is required to automatically generate the operations required to configure the architectural elements used by the middleware, such as communication endpoints and handler tasks. The current MAST model would need to be augmented to describe a few additional properties that are required for this automatic configuration:

- Nature of each task: whether it is dynamically created by the middleware or not (i.e., if it is a handler task, or an application task).
- Identify those message transmission operations that are coupled between them because they belong to a single RPC

The rules that would be used to generate the configuration operations are:

- For each message transmission operation create a send endpoint in the sending node (with the scheduling parameters associated with the message stream), and a receive endpoint in the receiving node.
- For each handler task create the corresponding handler
- For each input event to an activity, generate the mapping between the event and the task scheduling parameters. This includes the association between input and output events as well as between events related to the user or RPC Handler tasks and their scheduling parameters.

The mapping between the MAST model and the configuration operations uses all the MAST model events except those defined as an output of activities holding messages. In the following section we show this translation process through an example.

5. Example of use

This section describes a simple example of an application with two linear transactions performing asynchronous remote procedure calls through three processors and using one communication network. The MAST model of this system is shown in Figure 3. This example can represent a system that has to take an image of the environment, send it to another processor to be analyzed, and finally cause an action to occur in another place. The application has two transactions performing the same operations over different elements. Each transaction is composed of five activities:

- Taking an image.
- Sending the image to be analyzed.
- Analyzing the image to perform an actuation.
- Sending the command for actuation.
- Performing an action depending on the command received.

Figure 3 shows the details of the events, activities and processing resources (the processors or the network). Each processor contains only a single partition, so for our purposes, partitions and processors are equivalent. Note that in Partition 2 the processing image activity is performed by the same task for both transactions.

As can be seen in Figure 4, our approach simplifies this model considering the external events *External1* and *External2* which trigger the execution of the activities in *Partition 1* which, in turn, generate output events *e11* and *e21*. Those events cause messages to be sent to perform the remote calls. *Partition* 2 identifies the incoming events in the reception endpoint and transforms them to the corresponding events *e13* or *e23*. Then, the middleware sets the appropriate scheduling parameters for the handler tasks accordingly to the event received. After executing the activities in *Partition 2*, messages are sent to make the new remote call to *Partition 3* with the current events. Finally, the middleware in *Partition 3* will transform the incoming events

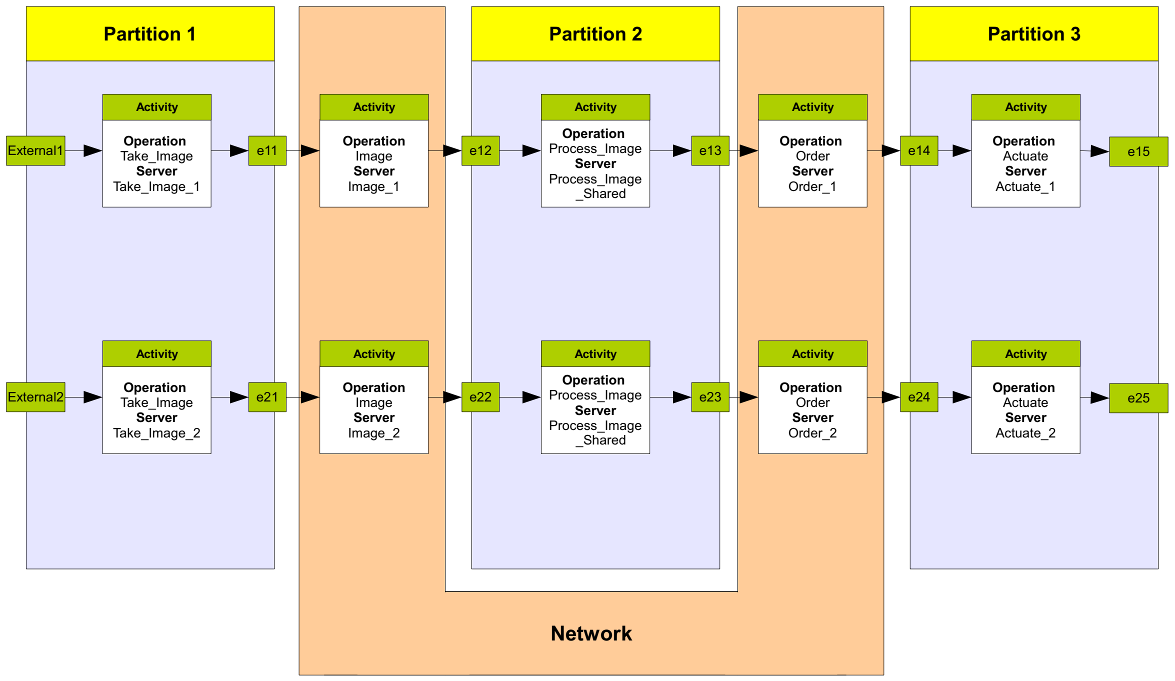
Partition 1
Partition 2
Partition 3
Network
Activity
Operation Take_Image Server Take_Image_1
Operation Image Server Image_1
Operation Process_Image Server Process_Image _Shared
Operation Order Server Order_1
Operation Actuate Server Actuate_1
Operation Take_Image Server Take_Image_2
Operation Image Server Image_2
Operation Order Server Order_2
Operation Actuate Server Actuate_2
External1
External2
e11
e12
e13
e14
e15
e21
e22
e23
e24
e25

Figure 3: Mast model

into events *e15* and *e25* which will be used to set the corresponding scheduling parameters for the activities. This ends the execution of transactions.

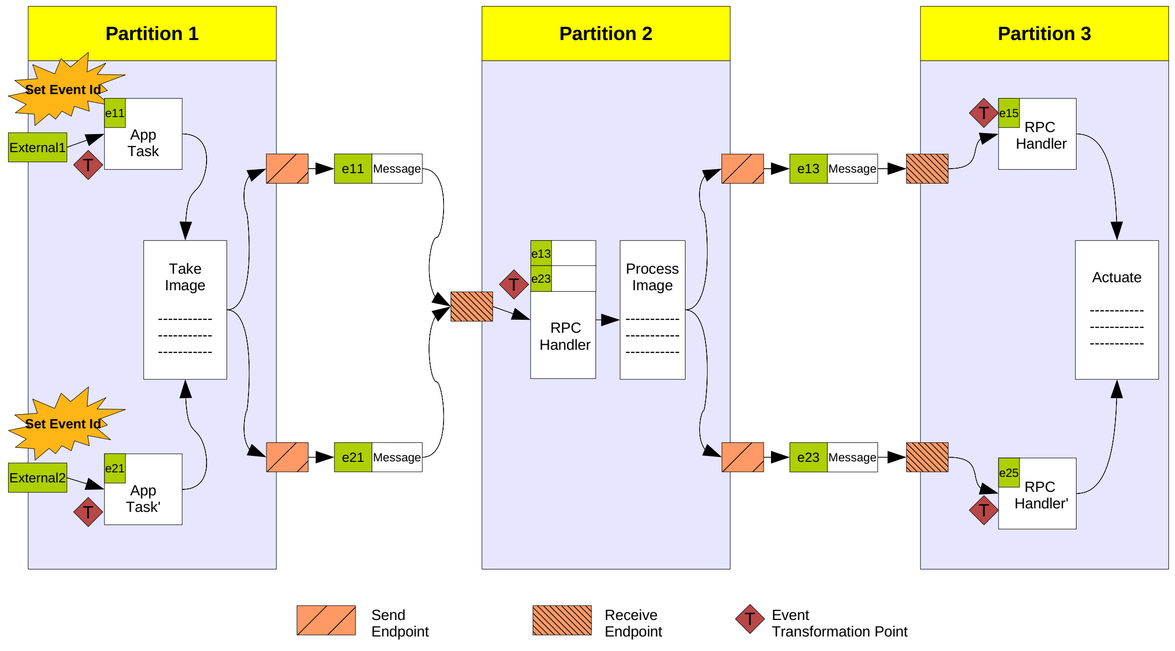

Figure 4: Simplified model

The configuration file for each partition would be generated automatically from the MAST model shown in Figure 3. As an example, we will show how to generate the configuration code for *Partition 2* and a similar procedure could be followed for *Partition 1* and *Partition 3*. The configuration code for *Partition 2* should include the following procedure calls:

- *Set_Event_Association:* As can be seen in Figure 3, *Partition 2* receives *e11* or *e21* as incoming events and transforms them to events *e13* or *e23* respectively. Such mapping between events remains registered in the middleware.
- *Create_Receive_Endpoint* and *Create_Send_Endpoint:* This partition considers four network transmission activities: two of them are for sending messages (scheduled by *Order_1* and *Order_2*) and thereby associated with two send endpoints with their specific scheduling information. The other two network activities (scheduled by *Image_1* and *Image_2*) correspond to incoming messages and therefore they must be mapped to receive endpoints. In this case only one receive endpoint is required because both input events (*e11* and *e21*) are processed by the same server task (*Process_Image_Shared*).
- *Create_RPC_Handler:* The *Process_Image_Shared* server acts as an RPC handler task and therefore must be created explicitly by the middleware.
- *Set_Event_And_Sched_Params_Association:* This call enables the shared RPC handler task to execute the *Process_Image* operation with different scheduling parameters depending on the triggering event after the corresponding transformation (*e13* or *e23*).

Using the procedures mentioned above, the Ada configuration code for *Partition 2* is shown below:

```
procedure Partition_2_Configuration_File is

-- The definition of variables is omitted

begin

   --  Set event associations
   Set_Event_Association
     (Input_Event  => e11,
      Output_Event => e13);

   Set_Event_Association
     (Input_Event  => e21
      Output_Event => e23);

   --  Create receive endpoint for shared RPC handler
   Create_Receive_Endpoint
     (Net      => Default_Network,
      Port     => Receive_Port,
      Endpoint => Rcv_Endpoint_Id);

   --  Create RPC Handler and scheduling params associated
   Set_Event_And_Sched_Params_Association
     (Params => Process_Image_Shared_Sched_Params_e13,
      Event  => e13);

   Set_Event_And_Sched_Params_Association
     (Params   => Process_Image_Shared_Sched_Params_e23,
      Event    => e23);

   Create_RPC_Handler
     (Default_Params => Process_Image_Shared_Sched_Params,
      Endpoint       => Rcv_Endpoint_Id);
```

```
    --  Create two send endpoints
    Create_Send_Endpoint
       (Param      => Msg_Scheduling_Parameters_e13,
        Dest_Node  => Partition_3,
        Event      => e13,
        Net        => Default_Network,
        Dest_Port  => Rcv_Port_Partition_e13,
        Endpoint   => Snd_Endpoint_Id_e13);

    Create_Send_Endpoint
       (Param      => Msg_Scheduling_Parameters_e23,
        Dest_Node  => Partition_3,
        Event      => e23,
        Net        => Default_Network,
        Dest_Port  => Rcv_Port_Partition_e23,
        Endpoint   => Snd_Endpoint_Id_e23);

end Partition_2_Configuration_File;
```

6. Conclusions

In this work, we have presented a proposal for the integration of real-time capabilities into the Distributed Systems Annex of Ada. The solution is based on the transactional model taken from scheduling theory, and is presented as a set of APIs to be included in a future version of the Ada standard.

The set of APIs proposed enables the logic of the application to be separated from the real-time aspects. The only requirement is simply to establish in the application software the initial event of each transaction. Furthermore, the configuration process of the transactions can be performed automatically taking advantage of the model obtained from CASE tools used in the design of the application. Using this set of APIs the middleware can manage the scheduling parameters in a transparent way. An example has been included to illustrate how this is done. Implementation of this framework is currently underway.

References

1. FRESCOR project web page: http://frescor.org
2. M. González Harbour, J. Javier Gutiérrez, J.C. Palencia, and J.M. Drake. “MAST: Modeling and Analysis Suite for Real Time Applications”. 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, IEEE Computer Society Press, pp. 125-134, 2001.
3. J. Liu. “Real-Time Systems”. Prentice Hall, 2000.
4. J. López, J. Javier Gutiérrez, and M. González Harbour. “The Chance for Ada to Support Distribution and Real Time in Embedded Systems”. 9th International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, LNCS, Vol. 3063, Springer, 2004.
5. J. López, J.J. Gutiérrez, and M. González Harbour. “Interchangeable Scheduling Policies in Real-Time Middleware for Distribution”. 11th International Conference on Reliable Software Technologies, Porto (Portugal), LNCS, Vol. 4006, Springer, 2006.
6. Object Management Group. “Realtime CORBA Specification”. OMG Document, v1.2 formal/05-01-04, January 2005.
7. Héctor Pérez, J. Javier Gutiérrez, Daniel Sangorrín, and Michael González Harbour. “Real-Time Distribution Middleware from the Ada Perspective”. 13th International Conference on Reliable Software Technologies, Ada-Europe, Venice (Italy), LNCS 5026, 2008.

8. J.C. Palencia, and M. González Harbour, “Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems”. 20th IEEE Real-Time Systems Symposium, 1999.

9. J.C. Palencia and M. González Harbour. “Response Time Analysis of EDF Distributed Real-Time Systems”. Journal of Embedded Computing (JEC), IOS Press, Vol. 1, Issue 2, 2005.

10. S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy (Eds.). “Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1”. LNCS 4348, Springer, 2006.

11. Douglas C. Schmidt, Carlos O'Ryan, Michael Kircher, and Irfan Pyarali. “Leader/Followers - A Design Pattern for Efficient Multi-threaded I/O Demulitplexing and Dispatching”. PLoP 2000 conference, Allerton Park, Illinois, USA, 2000.

12. Douglas C. Schmidt and Charles D. Cranor. “Half-Sync/Half-Async: A Architectural Pattern for Efficient and Well-structured Concurrent I/O”. Pattern Languages of Program Design, (Coplien, Vlissides, and Kerth, eds.), Addison-Wesley, Reading, MA, 1996.

13. Object Management Group. “A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2” OMG Document Number: ptc/2008-06-09, June 2008.