Interchangeable Scheduling Policies in Real-Time Middleware for Distribution

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. When a middleware layer is designed for providing semi-transparent distribution facilities to real-time applications, a trade-off must be made between the expressiveness and control capabilities of the real-time parameters used, and the simplicity of usage. Middleware specifications such as RT-CORBA or Ada's Distributed Systems Annex (DSA) rely on the use of priorities to map the timing requirements of the application, thus restricting the possible scheduling policies. This paper presents a generic technique to express complex scheduling and timing parameters of distributed transactions, allowing real-time middleware implementations to change their scheduling policies for both the processing nodes and the networks. The technique has been tested in an implementation of Ada's DSA, providing two interchangeable policies: a fixed-priority scheduler, and a complex contract-based flexible scheduler.

1 Introduction[1]

As real-time embedded systems grow in complexity and cover an increasing number of application areas, the need for real-time distribution grows accordingly. Developing software that can be migrated in a semi-transparent way from a single node platform to different distributed platforms with different interconnection architectures requires the use of a distribution middleware that takes care of all the communication implied, without explicit intervention of the application. There are different distribution middleware technologies, such as CORBA [8], which supports the distributed object paradigm, or the Distributed Systems Annex (DSA) in Ada 95 [12], which is mainly based on remote procedure calls (RPCs) and also supports distributed objects.

The advantages of these distribution middleware technologies are that they provide a level of abstraction that allows the application developer to concentrate on the problem being solved, independently of the platform used to execute it, and without having to program explicit message passing. Later, at configuration time, the particular mapping of software elements to processing nodes and communication networks is established, allowing the flexibility of migrating to different platforms, and the ability to explore different configurations. Explicit message passing makes the application not well structured and difficult to analyse.

1. This work has been funded in part by the Spanish *Ministry of Science and Technology* under grant number TIC2002-04123-C03-02 (TRECOM), and by the IST Programme of the European Commission under project IST-2001-34140 (FIRST).

If distribution middleware is to be used in real-time applications, it is necessary that the application developer has some way of mapping the timing requirements of the application into system parameters that can be used by the system to guarantee the required timing properties. In addition, the services provided and used by the middleware, such as dynamic task creation, the scheduling policies, or the networks and communication protocols must be designed so that the system is capable of guaranteeing predictable response times. For these reasons distribution middleware specially adapted to real-time systems has been specified and implemented. One of these specifications is real-time CORBA [9]. Another one is the DSA in the Ada language [12]; although it is not specifically designed to support real-time applications, it is possible to develop implementations that provide hard real-time guarantees [10][11][5].

Both of these middleware technologies, RT-CORBA and Ada's DSA, are based on fixed priority scheduling. The timing requirements of the application must be mapped on to priorities assigned to the user tasks and also to the tasks implementing the remote procedure handlers or the servers. In addition, if a fixed priority communication network is used, the application must also specify the priorities of all the messages involved [5]. This requirement somehow limits the "transparency" of distribution, but it is well known that in real-time applications a model of all the activities being performed must be known, and there must be ways to influence their timing behaviour. This applies to both hard and soft real-time applications, despite the fact that the timing models are required to be more precise for hard real-time.

The fixed priority approach is simple in the sense that the application just has to specify a number that can be dynamically assigned to tasks and messages, and fixed priority scheduling is widely available and simple to implement. However, many realtime applications being built today have a mixture of complex timing requirements that require the use of advanced scheduling policies capable of flexibly managing the available resources [2]. Moving towards more complex scheduling policies means that a single number such as a priority or a deadline is not enough to express the application requirements. Distribution middleware must be adapted to support these complex scheduling parameters that cannot be changed or transmitted dynamically. It must also be adapted to support changeable scheduling policies that can fit specific application requirements.

In this paper we present some ideas that allow a distribution middleware to manage complex scheduling parameters specified by an application in a way that minimizes overhead. We also show how these ideas can be used to adapt an implementation of Ada's DSA. For this particular implementation we show the API that the application has to use to set the scheduling parameters, and the way in which a new scheduling policy can be added to the system.

The paper is organized as follows. First, in Section 2 we present the model used to describe the event flow of a distributed real-time system, and its influence on the new approach for distribution middleware. In Section 3 we discuss the particular aspects of the communication layer in an implementation of Ada's DSA, and in Section 4 we

show the corresponding API. In Section 5 we show how to introduce a new scheduling policy under the new approach. Section 6 contains a simple example that illustrates the usage of the API, while Section 7 provides a case study and evaluates the ease of usage. Finally, Section 8 contains our conclusions.

2 The Transactional Model Applied to Distribution Middleware

Traditional distributed architectures built with RT-CORBA or Ada's DSA are based on the client-server architecture or the distributed objects model [10]. However, for analysing the response time of a real-time distributed application it is common to use the event-driven transactional model [7] (not to be confused with the transactional model used in database applications), in which 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 way to a chain of events and activities, possibly with end-to-end timing requirements, called the transaction (see Fig. 1). It is easy to show how an application designed with distributed objects or under the client-server architecture can be modelled and analysed using the transactional model.

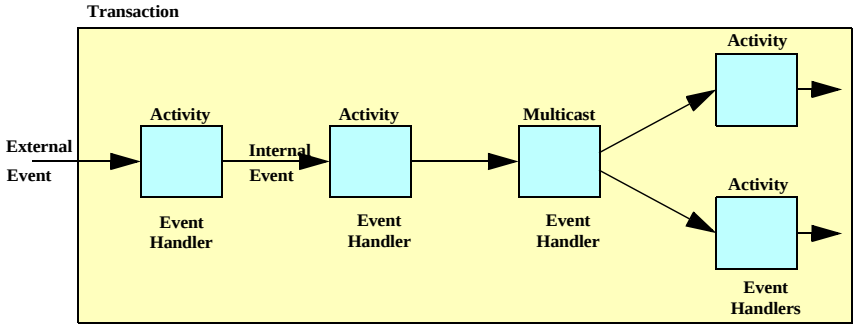

Fig. 1. Model of a transaction

To allow the highest degree of flexibility and the best timing results, it is useful to allow the application developer to assign the scheduling parameters of each activity involved in a transaction in an independent way [4]. This contradicts the traditional way of assigning scheduling parameters used in common distributed middleware specifications and implementations. For instance, in RT-CORBA we can assign a priority to a server, or it can inherit the priority of the calling client. None of these alternatives is completely satisfactory, because scheduling analysis might show that in a particular system configuration the optimum solution is to execute the server at a low priority when called from a high priority server, and use a high priority when called from particular a low priority client [4]. It is the transaction in which the server is being executed that should be used to determine the priority, but not with the rigidity that inheriting the client's priority imposes.

To allow a flexible use of the transactional model we proposed in [5] a modification of the GLADE [11] implementation of Ada's DSA, called RT-GLADE, in which the application developer was able to assign all the priorities involved in an RPC: the priority of the client task, the priorities of the query and reply messages, and the priority of the RPC handler in the server side. As it can be seen in Fig. 2, the underlying implementation automatically sets the priority of the query message, and encodes into it the priorities of the RPC handler and of the reply message. The system then chooses a task from the pool of RPC handlers to execute the remote procedure call, and dynamically changes its priority to the one specified in the message. Once the call is completed, the reply message, if any, is sent at the priority specified by the application. The remote call is transparent to the application code with the exception that the priorities must be set.

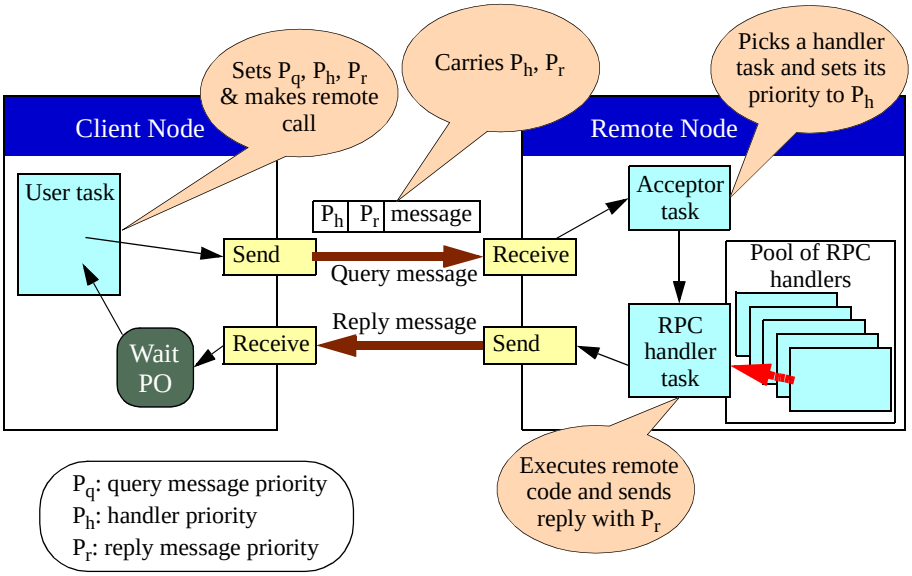

Fig. 2. Handling priorities in RT-GLADE

However this solution is tied to the use of fixed priorities, and does not work if more complex scheduling policies are used. Sending the scheduling parameters of the RPC handler and the reply message through the network is inefficient if these parameters are large in size. Dynamically changing the scheduling parameters of the RPC handler may also be inefficient. Both problems appear in the contract-based scheduling framework described later in Section 5, where the scheduling parameters represent a contract with tens of parameters, and for which dynamically changing a contract requires an expensive renegotiation process.

The solution proposed in this paper consists of explicitly creating the network and processor schedulable entities required to establish the communication and execute the remote calls, and identifying the created schedulable entities with a short identifier that can be easily encoded in the messages transmitted (see Fig. 3):

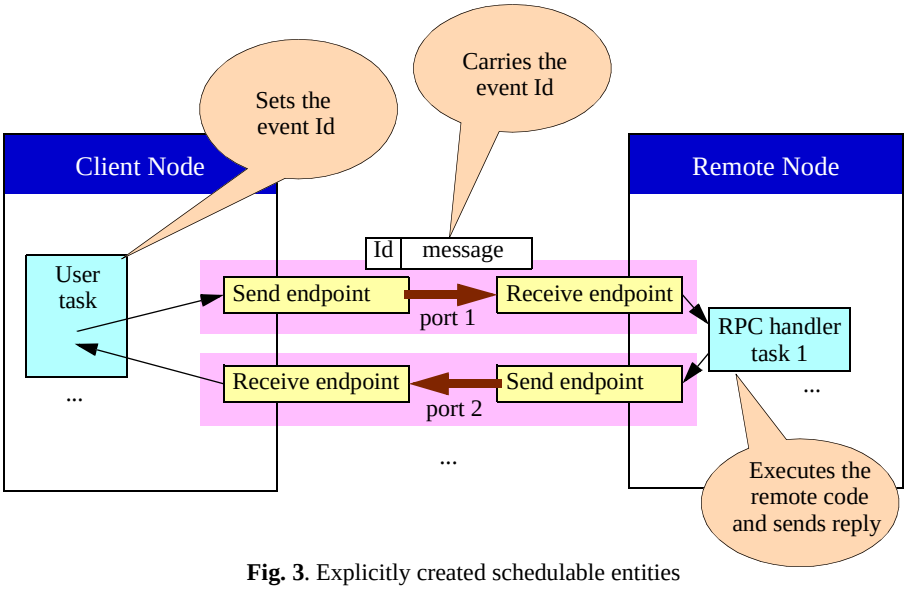

Fig. 3. Explicitly created schedulable entities

- For the processing nodes, the schedulable entities in the server side are the RPC handlers. Instead of having a pool of RPC handler tasks, we will create these tasks explicitly, each with their own appropriate scheduling parameters.
- For the network, the schedulable entities are communications ports that are used to establish the scheduling parameters that will be used for messages sent through that particular port. We will identify each of these ports through the two endpoints used to send and receive messages at either node. We will assume that the scheduling parameters are assigned to the port through its send endpoint. The endpoints are created and assigned their scheduling parameters explicitly.

For identifying these schedulable entities we will relate to the transactional model and use an identifier that represents the event that triggers the activity executed by the schedulable entity. We call it an `Event_Id`.

To achieve distributed communication in an application developed with the proposed approach it is necessary to create the following elements, as shown in Fig. 3:

- A send endpoint must be created in the client's node to send the message with the RPC query, containing information about the destination node and port.
- A receive endpoint must be created in the remote node for an RPC handler task to wait for the RPC reply message; it must specify the same port used in the caller's send endpoint.
- An RPC handler task must be created in the remote node to execute the remote call; it will directly wait for messages arriving at the corresponding receive endpoint in the remote node.

- A send endpoint must be created at the remote node to send back the RPC reply to the client; it must contain information on the destination node (where the client's partition is) and a port.
- A receive endpoint must be created in the client's node for the calling task to await the reply message; it must use the same port specified in the remote node's send endpoint.

It is also necessary to establish the corresponding scheduling parameters for each of the above elements, both in the processors and in the networks. This is usually done in the configuration or initialization part of each of the software components used. This configuration could be automatically generated by a tool that would obtain the information from the model of the transaction. Once configured, the usage of the new scheme is almost transparent, and the only requirement is that the application sets the desired `Event_Id`.

With this approach we can also eliminate the restriction in RT-GLADE that a distributed transaction with servers making nested calls to other remote servers could not fully specify the desired priorities for the nested calls, because only one set of priorities is sent in the calling message. With the new approach it is possible to fully specify all of the scheduling parameters, even in the presence of nested remote calls.

The proposed approach makes it possible to use different scheduling policies in different partitions, as each RPC handler task and communication endpoint can be created with the scheduling parameters that are appropriate for their underlying node or network.

It should be noted that there is no need to create more RPC handler tasks than those required by the transaction's architecture. The same RPC handler task can be used to execute many different remote procedures belonging to the same partition, as long as they can share the same scheduling parameters and provided they don't need to be executed concurrently among themselves.

3 The Communication Layer in RT-GLADE

In this section we will explain the modifications made to the original RT-GLADE communication layer in order to implement the proposed approach for interchangeable scheduling policies. The main architectural changes are:

- Removal of the `Acceptor_Task`: the purpose of these tasks was to receive messages directly from the network and process part of their information to then dynamically set the priority of the selected RPC handler tasks and awaken them. This is not necessary any more, so these tasks are removed in the new implementation, making it more efficient.
- Removal of the pool of RPC handlers: they were in charge of executing the calls in the remote node. In the new implementation the RPC handler tasks are created explicitly, and they wait for messages from the network directly, using the communication endpoints also created explicitly.

- Removal of the wait mechanism for a remote call reply: now the same user task that made the RPC waits directly for the reply because the communication endpoint is already created and known.

In order to identify the schedulable entities we create a special identifier called `Event_Id`. This identifier is specified by the application and used when creating the schedulable entities, which are the communication endpoints and the RPC handlers. Before an RPC is invoked, the application task must set the `Event_Id` associated with the current transaction, thus identifying the send and receive endpoints at its partition. This `Event_Id` is added to the RPC query message, so that the RPC handler task can read it and determine the send endpoint to which the reply should be sent in the remote node.

Internally, there is a mapping established between this `Event_Id` and the corresponding information relative to the communication layer; in particular, the identifiers of the required communication endpoints. It is possible to reduce the number of RPC handler tasks by grouping the execution of remote code placed in the same remote node through the same RPC handler.

4 The Application Interface for Ada's DSA

The application program interface appears in three packages: `Rt_Glade_Types`, containing basic data types; `Rt_Glade_Event_Id_Handling`, containing the only operations that are required to be invoked from the user code, to set or get the `Event_Id` for a remote call; and `Rt_Glade_Scheduling`, which contains the abstract types used to explicitly create and set the scheduling parameters of the communication endpoints and RPC handlers.

For representing the network elements, we will assume that there could be several networks, each identified with a value of the type `Network_Id`. A node is identified with a value of the type `Node_Id`. Reception queues at the destination node and network are identified by a port Id of the type `Port_Id`. These three types appear in package `Rt_Glade_Types`.

The `Event_Id` is stored as a task attribute in order to be easily used inside the RT-GLADE communication layer, as we did in the original RT-GLADE with the priorities involved in each RPC. The call used to set the `Event_Id` for a remote call is:

```
procedure Set_Event_Id (Id : Rt_Glade_Types.Event_Id);
```

There is also a function that may be called by an RPC handler task to get the `Event_Id` carried inside the query message that activated the current execution.

Package `Rt_Glade_Scheduling` contains two abstract tagged types and their corresponding classwide access types. The first of these types, `Task_Scheduling_Parameters`, represents the scheduling parameters of an RPC handler task. The second type, `Message_Scheduling_Parameters`, represents the scheduling parameters used for the messages sent through a specific endpoint. Both

types must be extended to represent actual parameters required by the underlying scheduling policies.

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

The primitives used to customize the communication layer by creating the RPC handler tasks and the communication endpoints are:

- Procedure `Create_Query_Send_Endpoint` creates a send endpoint for query messages. The arguments are the `Event_Id` that will be associated with this endpoint, the network that will be used to send the messages, the destination node and port in that network, and the scheduling parameters used for sending the messages.
- Procedure `Create_RPC_Handler` creates a receive endpoint for the query message and an RPC handler that waits at that receive endpoint. The arguments passed are the network and port from where the receive endpoint must receive the messages, the associated `Event_Id`, and the scheduling parameters used for the RPC handler.
- Procedure `Create_Reply_Send_Endpoint` creates a send endpoint for sending the reply messages to the originating partition. The necessary arguments are the `Event_Id` that will be associated with this endpoint, the network that will be used to send the messages, the destination node and port in that network, and the scheduling parameters for sending the messages.
- Procedure `Create_Reply_Receive_Endpoint` creates a receive endpoint from where the application task can read the reply message. Its arguments are the associated `Event_Id` and the network and port from where the receive endpoint must receive the messages.

Corresponding operations are provided to destroy the created endpoints or handlers, but their specification is omitted here for saving space.

For creating all the elements shown in Fig. 3 the application has to take the following actions when initializing software components involved in a remote call:

- Choose an unused `Event_Id` (`My_Event_Id`), an unused port in the remote node (`Remote_Port`), and an unused port in the client's partition node (`Calling_Port`).
- In the client's node, create the send endpoint by invoking `Create_Query_Send_Endpoint` using the desired scheduling parameters and network, specifying the node where the remote partition is located, and using `Remote_Port` and `My_Event_Id`. And create the receive endpoint by invoking `Create_Reply_Receive_Endpoint` using the desired network, and using `Calling_Port` and `My_Event_Id`.

- In the remote node, create the RPC handler and the receive endpoint by invoking `Create_RPC_Handler` using the desired scheduling parameters and network, and using `Remote_Port` and `My_Event_Id`. And also create the send endpoint by invoking `Create_Reply_Send_Endpoint` using the desired scheduling parameters and network, specifying the node where the calling node is located, and using `Calling_Port` and `My_Event_Id`.

After this initialization, RPCs can be made and they will be automatically directed through the appropriate endpoints and RPC handler by just specifying `My_Event_Id`, with procedure `Set_Event_Id` described above.

5 Implementation of Specific Scheduling Policies

Once a system supports a new scheduling policy, adapting the RT-GLADE implementation to use it requires extending the abstract types declared in `Rt_Glade_Scheduling`. The same applies to a new real-time communication protocol using a new scheduling policy for the messages.

The extension for the task and message scheduling parameters types requires that at least the scheduling parameters for the new policy are included among the new attributes of both types. In addition, the `Rt_Glade_Scheduling` specifies in its private part two abstract primitive operations of these types that must be defined. Their specification is:

```
procedure Create_Task
  (Params   : in Task_Scheduling_Parameters;
   Endpoint : in Rt_Glade_Types.Receive_Endpoint_Id;
   Tid      : out Ada.Task_Identification.Task_Id);
procedure Create_Send_Endpoint
  (Params   : in Message_Scheduling_Parameters'Class;
   Node     : in Rt_Glade_Types.Node_Id;
   Net      : in Rt_Glade_Types.Network_Id;
   Port     : in Rt_Glade_Types.Port_Id;
   Endpoint : out Rt_Glade_Types.Send_Endpoint_Id);
```

Procedure `Create_Task` creates an RPC handler task and associates the parameters to it in a manner appropriate to the scheduling policy being used. The handler waits for messages (from any source) arriving at the specified endpoint. Each message carries the `Event_Id` of the sender. If the call requires a reply, the handler sends the reply message through the send endpoint associated with the `Event_Id` carried in the message, unless the `Event_Id` is changed by the handler code.

Procedure `Create_Send_Endpoint` creates a send endpoint for the specified node, net and port and associates the parameters to it in a manner appropriate to the scheduling policy being used.

For the purpose of demonstrating the ability to change the scheduling policy, we have implemented two extensions of the `Rt_Glade_Scheduling` package, one for the traditional fixed priority scheduling, but under the new approach, and the other one for a complex contract-based flexible scheduling policy [3].

For the fixed priorities, we have implemented the extension in a child package called `Rt_Glade_Scheduling.Priorites`. The particularization of the two tagged types for this case are:

```
type Task_Priority is new Task_Scheduling_Parameters with record
   RPC_Handler_Priority:System.Garlic.Priorities.Global_Priority;
end record;

type Message_Priority is new Message_Scheduling_Parameters with
record
   Endpoint_Priority : System.Garlic.Priorities.Global_Priority;
end record;
```

The `Create_Task` operation creates an RPC handler task and sets its priority to the value specified in the `Task_Scheduling_Parameters` object. We do the same in `Create_Send_Endpoint` for the communication endpoints.

The First Scheduling Framework (FSF) [3] is a framework for a scheduling architecture that provides the ability to compose several applications or components to build a real-time system, and to flexibly schedule the available resources while guaranteeing hard real-time requirements. It is is based on establishing service contracts that represent the complex and flexible requirements of the application, and which are managed by the underlying system to provide the required level of service. FSF is applied both to processor and networks. From the application's perspective, the requirements of an application component are written as a set of contracts, which are negotiated with the underlying implementation. To accept a set of contracts, the system has to check if it has enough resources to guarantee all the minimum requirements specified, while keeping guarantees on all the previously accepted contracts negotiated by other application components. If as a result of this negotiation the set of contracts is accepted, the system will reserve enough capacity to guarantee the minimum requested resources, i.e., processor capacity and network bandwidth, and will adapt any spare capacity available to share it among the different contracts that have specified their desire or ability for using additional capacity.

In summary, the FSF contracts are the scheduling parameters of the tasks and messages under this framework, and they are very different in nature from the plain fixed priorities.

To use this scheduling framework, we have created a child package called `Rt_Glade_Scheduling.Fsf` in which we have extended the types to hold an FSF contract as a scheduling parameters. In addition, the operations `Create_Task` and `Create_Send_Endpoint` perform the contract negotiation. It is important to set in the contract, at least, the minimum budget, and the maximum period to ensure a minimum amount of system resources.

6 Example Using the Proposed Approach

In this subsection we will show how to build a simple application using the approach presented in this paper. For simplicity, we will use the fixed priorities scheduling scheme. In this application, shown in Fig. 4, we are going to perform a remote add

operation. The application is composed of two partitions, each placed in a different node. Partition `p1` contains the code of the main program, `menu`, and the initialization code for that partition, `P1_Init`, while partition `p2` contains the code implementing the calculator, `Calculator`, and the initialitation code, `P2_Init`. The Event Ids and the network ports are chosen arbitrarily from those still available.

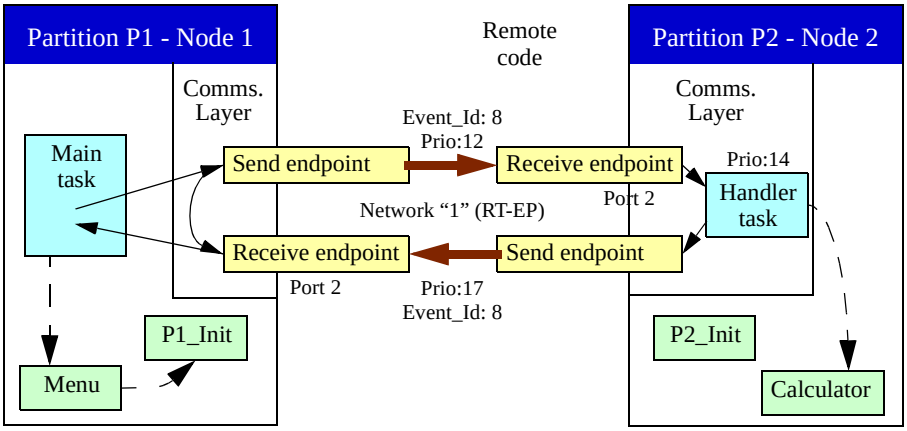

Fig. 4 Example using fixed priorities in the new version of RT-GLADE

The `menu` procedure code is as follows:

```
with Calculator, RT_Glade_Event_Id_Handling, P1_Init;
procedure Menu is
   A,B,Sum:Integer;
begin
   -- Initialize the communications endpoints
   P1_Init;
   -- Set the event id for subsequent RPCs
   Rt_Glade_Event_Id_Handling.Set_Event_Id(8);
   loop
      ...
      Sum := Calculator.Add (A, B); -- remote call
      ...
   end loop;
end Menu;
```

The `P1_Init` procedure is used to initialize all the neccesary elements to establish the communications in partition `p1`:

```
with Ada.Real_Time, Rt_Glade_Scheduling;
with Rt_Glade_Scheduling.Priorities;
procedure P1_Init is
   R_Message_Params :
      Rt_Glade_Scheduling.Message_Scheduling_Parameters_Ref;
begin
   -- Create the send endpoint for the query messages
   R_Message_Params:= new Rt_Glade_Scheduling.Priorities.
      Message_Priority'
        (Rt_Glade_Scheduling.Message_Scheduling_Parameters
         with Endpoint_Priority=>12);
```

```
   Rt_Glade_Scheduling.Create_Query_Send_Endpoint
      (Params=>R_Message_Params.all, Node=>2,
       Net=>1, Port=>2, Event=>8);

   -- Create the receive endpoint for the reply messages
   Rt_Glade_Scheduling.Create_Reply_Receive_Endpoint
      (Net=>1, Port=>2, Event=>8);
end P1_Init;
```

The `Calculator` package is a normal ada implementation with the only difference that the package has to be categorized with the `pragma Remote_Call_Interface` which makes all the procedures callable remotely.

```
package Calculator is
   pragma Remote_Call_Interface;

   function Add (A : in Integer; B : in Integer) return Integer;
end Calculator;
```

The `P2_Init` procedure is a main partition program to initialize all the neccesary elements to establish the communications in `p2`:

```
with Rt_Glade_Scheduling, Rt_Glade_Scheduling.Priorities;
with Ada.Real_Time;
procedure P2_Init is
   R_Task_Params :
      Rt_Glade_Scheduling.Task_Scheduling_Parameters_Ref;
   R_Message_Params :
      Rt_Glade_Scheduling.Message_Scheduling_Parameters_Ref;
begin
   -- Create the RPC handler
   R_Task_Params:=new Rt_Glade_Scheduling.Priorities.
      Task_Priority'
         (Rt_Glade_Scheduling.Task_Scheduling_Parameters
          with RPC_Handler_Priority=>14);
   Rt_Glade_Scheduling.Create_RPC_Handler
      (Params=>R_Task_Params.all, Net=>1, Port=>2, Event=>8);

   -- Create the send endpoint for the reply message
   R_Message_Params:= new Rt_Glade_Scheduling.Priorities.
      Message_Priority'
         (Rt_Glade_Scheduling.Message_Scheduling_Parameters
          with Endpoint_Priority=>17);
   Rt_Glade_Scheduling.Create_Reply_Send_Endpoint
      (Params=>R_Message_Params.all, Node=>1,
       Net=>1, Port=>2, Event=>8);
end P2_Init;
```

As we can see, once the initialization code for creating the communication endpoints and the RPC handler is written, the only change to the application code is the setting of the Event Ids.

7 Case Study and Evaluation of Application Complexity

We have evaluated the impact of migrating a real application that had been built using the fixed-priority version of RT-GLADE to the new proposed approach for scheduling, in particular using the FSF contracts mentioned in Section 5. The application is a simulated tile inspection plant that uses a real industrial robot arm and

a video acquisition system, running on a MaRTE OS operating system [1] and using the RT-EP real-time communication protocol [6]. To minimize the number of RPC handler tasks, we have grouped all the calls for a particular transaction and to a given partition into a single RPC handler, calculating its budget as the sum of the worst case execution times of all the RPCs involved. With this approach, every transaction in the application code has been assigned a single `Event_Id` that identifies its communication endpoints and the associated RPC handler task.

The number of source code lines that were necessary to make the networks contracts for seven distributed transactions was 14*7 = 98 lines to create 14 send endpoints, 7*8 = 56 lines to create 7 RPC handlers and associated receive endpoints in the remote nodes, and 7 lines to create the 7 receive endpoints in the calling tasks. Besides we added 13 “with” lines, and 7 lines to specify the `Event_Ids` of each distributed transaction. So in total we have added just 181 lines of code, to a program of more than 13.000 lines.

In summary, we did not require any changes to the architecture when migrating from the original version of RT-GLADE to the new version. Changes were only required for the creation of the network contracts, the communication endpoints, and the RPC handlers, and for the specification of `Event_Ids`. The amount of new lines of code is very small, compared to the size of the application. We can conclude that having a well-documented architecture and real-time model of the application makes it very easy for a designer to use the FSF contracts, or any other special-purpose scheduling policy, for a distributed application under the new RT-GLADE implementation.

8 Conclusions

We have proposed an approach to support generic scheduling parameters and policies in real-time distributed middleware. This approach follows the transactional model in which the scheduling parameters are determined by the sequence of events that are activated inside a real-time transaction, allowing the highest degree of flexibility and resource usage. Complex scheduling policies, such as those used to achieve flexible scheduling using contracts or reservations can be handled with the proposed approach, even if the size of the scheduling parameters is large, or if dynamic changes of these parameters are expensive. With the new approach we can define the scheduling parameters of whole distributed transactions with complete freedom, even in the presence of nested remote procedure calls.

The approach has been tested in an implementation of Ada’s DSA, by providing two interchangeable policies: a fixed-priority scheduler, and a complex contract-based flexible scheduler. The new implementation continues to conform to Ada’s DSA and is independent of the kind of scheduling parameters used. The architecture of this implementation is simpler than before, although it may require more space as the pool of RPC handlers is replaced by a possibly larger set of explicitly created handlers. In summary the new implementation is more flexible, and has more control on the scheduling of the different elements involved in the distributed transactions, at the

price of requiring more intervention from the application and using more tasks and network access points.

References

1. 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, Springer LNCS 2043, May 2001.

2. Bruno Bouyssounouse and Joseph Sifakis (Eds.) “Embedded Systems Design. The ARTIST Roadmap for Research and Development”, Springer LNCS Vol. 3436, 2005.

3. M. Aldea, et al. “FSF: A Real-Time Scheduling Architecture Framework”. Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, San Jose, CA, USA, April 2006.

4. 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.

5. Juan López Campos, J. Javier Gutiérrez, Michael González Harbour, "The Chance for Ada to Support Distribution and Real-Time in Embedded Systems". Proceedings of the 8th International Conference on Reliable Software Technologies, Ada-Europe, Springer LNCS 3063, June, 2004, ISBN:3-540-22011-9, pp. 91,105.

6. José María Martínez and Michael González Harbour, "RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet". Proceedings of the 9th International Conference on Reliable Software Technologies, Ada-Europe, York, Springer, LNCS-3555, June, 2005.

7. J.L. Medina Pasaje, M. González Harbour, J.M. Drake Moyano, "MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems". Proceedings of the 22th IEEE Real-Time Systems Symposium, London, UK, December, 2001, ISBN:0-7695-1420-0, pp. 245,256.

8. Object Management Group. CORBA Core Specification. OMG Document, v3.0 formal/ 02-06-01, Julio 2003.

9. Object Management Group. Realtime CORBA Specification. OMG Document, v2.0 formal/03-11-01, November 2003.

10. 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.

11. 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.

12. 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.