Managing Transactions in Flexible Distributed Real-Time Systems*

Daniel Sangorrín[1], Michael González Harbour[2], Héctor Pérez[2], and J. Javier Gutiérrez[2]

[1] Graduate School of Information Science
Nagoya University, Nagoya, Japan
dsl@ertl.jp
http://www.ertl.jp
[2] Computers and Real-Time Group
Universidad de Cantabria, 39005 - Santander, Spain
{mgh,perezh,gutierjj}@unican.es
http://www.ctr.unican.es/

Abstract. This paper describes the design and implementation of the Distributed Transaction Manager (DTM), a service that provides remote negotiation of contracts representing resource reservations in real-time distributed applications. We assume that there is an underlying middleware which can be used by the application to negotiate contracts locally: processor contracts have to be negotiated in the same processor where they will run, and network contracts have to be negotiated in a processing node connected to the specific network that will be used. In addition, the paper proposes the integration of the DTM in a distribution middleware based on CORBA and Ada's Distributed Systems Annex (DSA) which supports advanced scheduling mechanisms based on contracts. The use of the distribution middleware enhances some implementation aspects of the DTM and provides new capabilities as, for example, routing messages through different networks.

Keywords: flexible scheduling, real-time, distribution middleware, CORBA, Ada DSA, communications.

1 Introduction

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,

* 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 authors' views; the EU is not liable for any use that may be made of the information contained herein.

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 251–264, 2010.
© Springer-Verlag Berlin Heidelberg 2010

instead of asking the application to interact directly with the scheduling policies, scheduling services of a higher level of abstraction are being designed, usually based on the concept of resource reservations [1]. The FRESCOR European Union project [2] in which we have participated was aimed at investigating these aspects by creating a contract-based scheduling framework.

Future development of real-time distributed systems will be supported by high-level models as the one defined in the standard OMG Specification MARTE (Modelling and Analysis of Real-Time Embedded Systems) [3]. This standard includes the *end-to-end flow* as a basic entity for modelling the behaviour of distributed systems. This is the new name for the *transaction*, a set of interrelated operations with a timing constraint between its start and its completion (not to be confused with the meaning that the same word has in the domain of databases). In the context of this paper a distributed transaction consists of a sequence of activities that can be either task jobs in the processors or messages in the networks. This paper explores new trends in managing transactions for flexible scheduling systems.

Although the support for contracts defined in the FRSH API (FRESCOR framework application interface) [4] is enough to negotiate contracts locally, it puts a burden on the application to manage the negotiation process of a whole distributed transaction. The first objective of this work is to propose and implement a Distributed Transaction Manager (DTM) that extends the negotiation capabilities defined in FRSH to allow remote negotiations and renegotiations of contracts, and a centralized management of the results of the negotiation process. The implementation platform for the DTM is a FRESCOR environment using the C language and MaRTE OS [5] with RT-EP [6][7] and CAN [8] real-time networks. All of these resources have implementations supporting adaptive resource reservations with contracts.

In previous works we proposed mechanisms for the integration of middleware and advanced scheduling services, such as contract-based reservations, with transactions. In [9] some initial ideas were given to allow a distribution middleware to manage complex scheduling parameters specified by an application in a way that minimizes overhead. In that work, a proposal to integrate a generic technique to express complex scheduling and timing parameters of distributed transactions was presented. The implementation of these ideas over PolyORB [10][11] ported to MaRTE OS [5] was presented in [12]. PolyORB is a distribution middleware by AdaCore [13] which supports several distribution standards. Finally, in [14] we extended this generic technique and proposed it to be integrated as a part of the Distributed Systems Annex of Ada (DSA).

The first implementation of the transaction manager described in this paper limits its capabilities just to the management of remote contracts, and as a proof of concepts it was implemented directly over the network services (RT-EP and CAN Bus) [7][8]. We are now implementing a second version to provide a full support for the transactional model integrated with the capabilities of a distribution middleware. This proposal will be fully implemented in PolyORB over MaRTE OS. Currently we have implemented the key issues in order to check the feasibility of the proposal.

The document is organized as follows. Section 2 describes the specification of the Distributed Transaction Manager and gives the details on its main services. Section 3 deals with the architecture of the implementation of the DTM. The model of the transaction and its implementation in the middleware is presented in Section 4.

Section 5 proposes the integration of the DTM in distribution middleware based on standards. The extension of the DTM protocol to provide full support for the transactional model as well as some issues of the implementation of the DTM in PolyORB are pointed out in sections 6 and 7, respectively. Finally, Section 8 draws the conclusions and considers future work.

2 Specification of the Distributed Transaction Manager

The Distributed Transaction Manager (DTM) is a distributed application responsible for negotiation of distributed transactions in a FRESCOR contract-based scheduling framework implementation. FRESCOR contracts provide a resource reservation framework that, as such, provides protection among the different software components running on top of it, facilitating the independence of their development and execution. But the framework produced in FRESCOR goes well beyond the capabilities of other resource reservation frameworks by providing adaptive reservations that can make use of spare capacity available; management of QoS parameters expressed at the application level; an integrated management of multiple schedulable resources including CPUs or networks; management of time-protected shared objects; integration with component-based design methods; off-line schedulability analysis and simulation tools that allow the application developer to reason about the timing behaviour of the application before it is built; and an API called FRSH that makes the application independent of the underlying operating system, networks and scheduling policies. Unlike other adaptive QoS approaches [15], the FRESCOR framework guarantees the minimum budget requested via a schedulability analysis of the system.

The requirements of an application or 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 as part of the negotiation 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, 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. As a result of the negotiation process initiated by the application, if a contract is accepted, a virtual resource is created for it representing an adaptive resource reservation.

The objective of the Distributed Transaction Manager is to allow in distributed systems the remote management of FRESCOR contracts, including capabilities for remote negotiation and renegotiation of resource reservation contracts, and management of the coherence of the results of these negotiation processes. In this way, FRESCOR provides support for distributed global activities or transactions consisting of multiple actions executed in processing nodes and synchronized through messages sent across communication networks.

Fig. 1 shows the overall DTM architecture as viewed from the application. The DTM has been designed as a layer between the application and the FRSH API in order to avoid increasing the complexity of the FRSH implementation. The DTM is

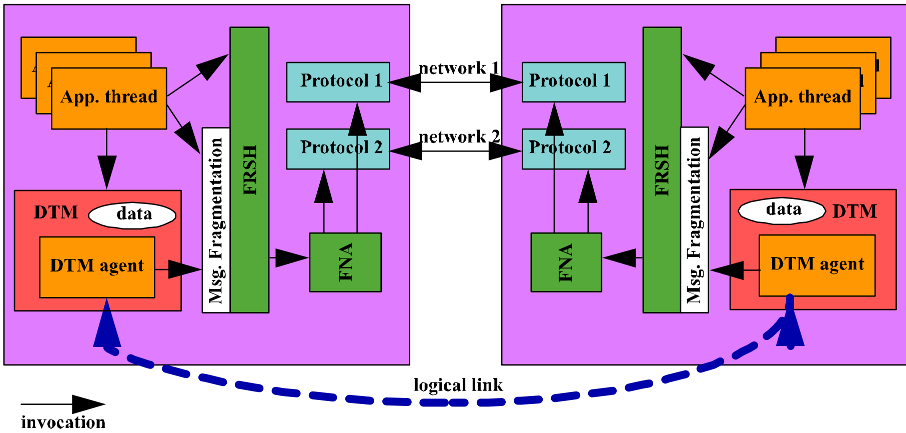

Fig. 1. Role of the DTM in the FRESCOR application architecture

instanced at every node in the system. Each instance stores necessary data for the negotiation process and a *DTM agent*, which will listen to negotiation messages either from the local node or from remote nodes, will perform the requested actions, and will send back replies when required.

The remote negotiation process requires bidirectional communication, either direct or through other nodes acting as routers, between the node requesting the negotiation and the node performing it, because the requester needs to get the results of the negotiation. As the level of resource reservations required for supporting the routing operations is difficult to configure in general, the DTM may use a static routing mechanism that utilizes the resource reservations assigned by the system designer to this service.

It is expected that in some networks the DTM messages, specially those containing contracts, will not fit into the maximum size of a packet. For these cases, a fragmentation layer is deployed between the DTM implementation and the FRSH distribution services. This layer is independent of DTM and can be used by the application to send large messages as can be seen in Fig. 1.

The services that the DTM offers to the application are defined by the DTM API [7][8]. The main services are briefly described as follows:

- **Assignment of Resources to the Transaction Manager.** This service allows the application to specify the resource reservations made locally for the operation of the transaction manager itself. Some processing bandwidth must be reserved for the DTM manager thread responsible of executing the DTM services. In addition, network bandwidth must be reserved for the messages that the DTM agents must exchange among them. In this way, the framework will guarantee that the manager does not overrun the capacity assigned for its execution.
- **Initialization of the Transaction Manager.** In order to use the services of the transaction manager it is necessary to ensure that the DTM agents and their data structures have been initialized in all nodes. Otherwise, a message sent to a remote node to request some service could be lost if nobody is at the other side to store and process that message. This service implies creating at each node the

necessary data structures and threads to implement the transaction manager, and synchronizing the initialization with all the nodes involved in the system, by waiting until all of them are initialized.

- **Creation and Management of a Transaction.** This service allows the application to specify the set of contracts that compose a transaction and the nodes where they have to be negotiated. Transactions may require that the periods of all or some of the virtual resources associated to it be the same, to achieve synchronized behaviour. This requirement will be specified by the application when a contract is added to a transaction, and will cause the negotiation process to reach a consensus on the common period to be used for those virtual resources requiring a synchronous period.
- **Negotiation of a Transaction.** This service implies:
  - Checking the consistency of the transaction.
  - Negotiating contracts in remote nodes. This is accomplished by sending a message to each involved DTM agent, with the corresponding contracts. Each DTM agent does the local negotiation and reports back to the original DTM agent the results of the negotiation.
  - Negotiating local contracts.
  - Collecting the results of the remote negotiations.
  - Making a decision based on the results. If one or more of the results implied that a contract was rejected, the whole transaction will be rejected. If all the contracts were accepted and one or more contracts require synchronous periods it is necessary to find out which is the most restrictive period, which will then become the synchronous period of the transaction.
  - Communicate the final decision to all the implied DTM agents, which in turn will adjust the periods of the contracts when necessary, to match the synchronous transaction period.

3 Architecture and Implementation of the DTM without Distribution Middleware

The implementation of the Distributed Transaction Manager in FRESCOR is a decentralized architecture in which the transactions can be created and negotiated from any processing node. Each DTM agent contains a global data structure where the information about the negotiations is kept. This data structure contains information about whole transactions that were created in the corresponding processing node, and also partial information about the parts of those transactions that were created remotely but have been negotiated locally.

The DTM agent contains multiple threads, as can be seen in Fig. 2. One of them, called the *DTM manager thread* is in charge of performing the actions related to the management and negotiation of transactions. The other threads, called the *DTM acceptor threads*, are in charge of listening to messages arriving through the different networks, coming from other remote DTM agents. Fig. 2 also shows the parts that would be replaced with the services provided by the middleware and that will be discussed in Section 5.

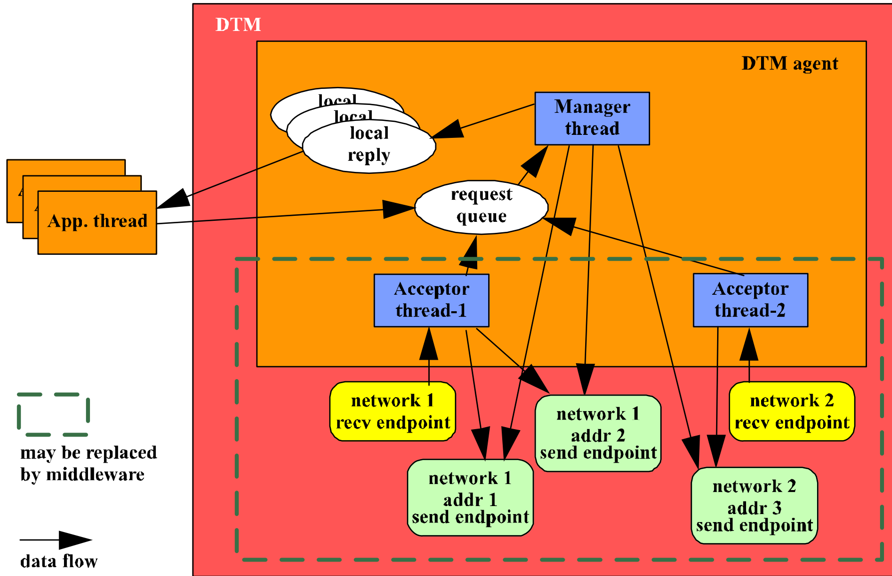

Fig. 2. Internal architecture of a DTM agent

The DTM acceptor threads perform two different services:

- *Routing*: when the DTM message arriving at the acceptor thread is addressed to a different node, it resends that message through the appropriate network according to a static routing table.
- *Handling*: when the DTM message arriving is addressed to the current node, the message type is checked, and a request is queued for the *DTM manager thread*.

The DTM manager thread waits for the arrival of DTM messages at the DTM request queue. To minimize the service times, this is a prioritized queue. When a DTM message arrives, either from a DTM acceptor or directly from a local thread in the application, the DTM manager processes that message. Depending on the type of message, new requests to other remote nodes may be necessary, in which case the manager will have to wait to get the replies from the DTM request queue. Once the request is serviced, a reply is correspondingly sent to the local thread or to a remote node.

The DTM agent contains one send endpoint per connected node through which it can send messages. This send endpoint is bound to the virtual resource that was specified by the application. The DTM agent also contains one receive endpoint per connected network, which is used to receive remote DTM messages.

When a local application thread invokes one of the services that requires intervention of the manager thread, a request is enqueued in the DTM request queue. The local application thread is then put in a *local reply* object, which is a synchronization object where it will wait until the DTM manager sends a reply. Since concurrent invocation of the DTM services is possible, a pool of these synchronization objects is

needed. The number of elements in this pool is configurable. Local reply objects will be identified through a handle value.

To avoid deadlocks, processing of local and remote DTM messages by the manager thread is preemptible. While a requesting application thread is waiting in the local reply object, the manager will continue to process other messages. Between the manager thread making a remote negotiation request and receiving the reply from the remote node it will also continue to process other DTM messages. The implementation allows concurrent negotiations being requested by multiple nodes.

4 Transactional Model in Middleware

In this section we summarize the architecture of the transactional model that we will use. This model was proposed in [14] to be included in the DSA, and it defines an interface to allow the middleware and the applications to manage event associations and the scheduling of the different components of a transaction. 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. 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:

- The *handler tasks* intended to execute remote calls in the processing nodes. These handlers are created explicitly with the appropriate scheduling information.
- 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.

In this transactional model, external events trigger the transactions. The only explicit operation that is required from the application code is to set an identifier for each of those events (the *Event_Id*). 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 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 defined for each part of the transaction. This *Event Id* is set internally by the middleware at the *transformation points* defined at the following actions: setting the initial event, the start of the execution of an RPC Handler, and the reception of the reply by a task waiting for a synchronous RPC.

Pre-configured scheduling parameters will be used according to the *Event_Id* for each application task or RPC Handler. 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.

5 Integration of the DTM with Distribution Middleware

As described in previous sections, the DTM is essentially a distributed application that contains an agent in every node. The agents listen to incoming local or remote messages, perform the requested actions, and send back the replies. This architecture was initially implemented directly over the network communication primitives, but could alternatively be implemented using different distribution standards (e.g. CORBA [16], DDS [17], or Ada DSA [18]), thus simplifying the complexity of the communication among agents.

The presence of distribution middleware implements part of the functionality inherent to DTM. In particular, this architecture could benefit from using a standard distribution model in the following aspects:

- *Simplicity of the protocol and the DTM application.* The current architecture implements a specific communication protocol which requires the management of all the data involved in the remote request (i.e. source node, destination node, size of contracts...). However, middleware could manage the distribution aspects in a transparent way for the user (e.g., using the IOR in CORBA [16] or the topic subscription in DDS [17]).
- *Initialization.* The initialization process in the DTM assures that every agent in the distributed system is ready to send and receive requests. In this case, a static approach has been proposed where the user has to specify all the connection paths within the system. This model could be replaced by a more dynamic approach like that used in DDS, where the discovery of new entities is made at runtime through the built-in topics (DDS entities to provide information to the application). Those special entities must also have another network contract associated, to limit the overhead introduced by the discovery process. Despite that neither CORBA nor DSA do not specify any mechanism to assert the initialization of entities, a solution based on the CORBA Naming Service [19] will be proposed in Section 7. Although this solution is intended to use a CORBA service, it could be also applied to DSA by means of the interoperable platform provided by PolyORB.
- *Removal of the Acceptor Tasks.* The two services provided by those entities could be replaced by middleware:
  - *Handling.* It is the process of listening for incoming remote calls, processing them with the appropriate scheduling parameters and notifying any new requests to the local DTM agent. In this case, the same functionality could be provided by the RPC Handler tasks.
  - *Routing Service.* In this context, routing means the capacity to interconnect different networks in systems where nodes are not connected directly.
  - *Fragmentation services.* Although this service could be developed as a new FRESCOR service, most distribution standards implicitly integrate it (e.g., GIOP in CORBA [16] or the DDS Interoperability Wire Protocol [20]), so the use of middleware would avoid the inclusion of unnecessary extra features into the framework.

However, a general purpose middleware could not satisfy all the requirements imposed by the DTM (e.g., complete support for the transactional model or a resource

reservation scheduler). Our previous work with GLADE [9] and PolyORB middlewares [21], which aimed to provide support for different scheduling policies and the distributed transaction, facilitates the transition to the integration by introducing:

- *Full support for the transactional model.* Our model manages internally all the transaction-related details: identification of the transaction, mapping between incoming and outgoing events in the transformation points [14] and the assignment of the appropriate scheduling parameters for each element of the transaction.
- *Transaction identification*. The identification of a transaction is performed through the *Event_Id* parameter, which is then interpreted by the middleware [9]. New transactions activated through the DTM will be required to have specific *Event_Ids* defined. This association could be automatically managed by middleware and will be discussed in the next section.
- *Routing*. As we pointed out before, this service should be included as part of the middleware to limit the overhead that would be incurred by crossing all the implementation layers up to the application level. Our approach focuses on the routing through the *Event_Id*. Once the routing node has identified the event parameter and thus the ongoing transaction, it inquiries the middleware whether the invoked object is located in the local node or not. If not, the implementation will route the same incoming message through the pre-configured send endpoint.

Finally, another minor enhancements could be introduced in the DTM:

- Removal of the restriction of waiting for a specific transaction. The current API requires the user to specify the name of the transaction whose negotiation is waiting for. Such restriction could be easily solved in Ada through the use of an empty string as the default value for this parameter in the API function. This could be useful in dynamic systems where a pool of threads are supplying different services (e.g., a multimedia system providing video and regular voice calls) and thus different transactions could be executed.
- Processing of different requests using the same RPC Handler. The proposed internal management of the middleware implementation allows that a single RPC handler task could process several requests corresponding to different transactions, since the handler task is just bound to a particular endpoint and not to a single event. The same applies now to the Acceptor Tasks because in the new implementation they become regular RPC Handler tasks.

6 Extension of the DTM Protocol to Provide Support for the Transactional Model

The current version of the transaction manager limits its capabilities just to the management of remote contracts. However, once integrated within the middleware, the automatic deployment of the complete transaction becomes desirable: after accepting each new transaction in the system, the user should set only the initial *Event_Id* parameter while the middleware would automatically direct the remote call through the appropriate endpoints and RPC handlers transparently using the negotiated contracts.

This automatic deployment requires an extension of the protocol used by the DTM to enable the middleware to create and automatically set up handlers and endpoints. In particular, the new integrated version would require:

- *The specification of the full transaction*. In addition to the contract-related parameters, the transaction flow should be specified by the user. A possible solution consists of making use of a configuration language. Through the use of this language, middleware can bind the scheduling parameters, identified by their contract's name, to the associated flow (e.g., the appropriate *Event_Ids*). Furthermore, the definition of *RPC_Handlers* requires differentiating these contracts from those related to regular application tasks which are not bound to any receive endpoint.
- *Choosing unused Event_Ids*. Since any node can start a new negotiation, the maintenance of a list of all the events being used along the distributed system would increase the complexity and the overhead introduced by the agents. The proposed solution is based on the reservation of a certain amount of *Event_Id* values per node. This permits that each node that starts a negotiation, can safely select all the *Event_Ids* to be used in the transaction flow. Then, this extra information would be sent with the contract's data in the same *DTM_REQ_NEG_SET* message (see Fig. 3). This approach matches most of the requirements imposed by small and medium-size distributed systems.
- *Choosing unused ports.* Middleware should select an unused port in those nodes where the receive endpoints must be created, and then this new information will be sent back to the root node (e.g. the node which started the negotiation) in the *DTM_REP_NEG_SET* message (see Fig. 3). However, the root node must share this information with those nodes specified in the transaction flow and could include it in the *DTM_REQ_STATUS* (see Fig. 3) message which informs that the transaction has been accepted. Only then, the middleware could create the handlers and the endpoints with the appropriate parameters to enable the communication.

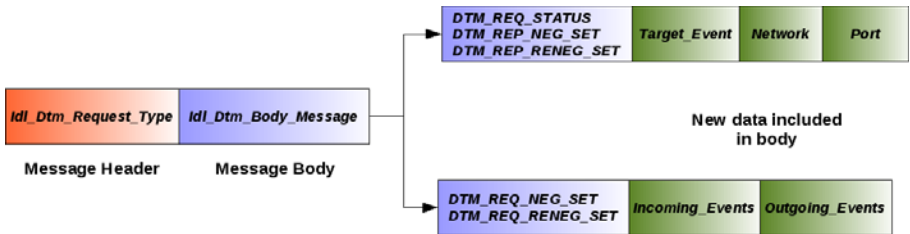

Fig. 3. DTM Protocol Extended

7 Implementation of the DTM in PolyORB

Nowadays, CORBA is considered a very mature technology to build distributed systems within the industry. However, a variety of problems still need to be addressed for systems with timing requirements [12]. Bringing the FRESCOR framework to a

CORBA-compliant middleware will enable to establish timing protection for the real-time distributed application components, that is, no tasks will be allowed to overrun their assigned and guaranteed budget execution time, or use more communication bandwidth than the reserved allocation. However, there could be systems that are part of a dynamic environment and thus the system requirements could not be completely known in advance at configuration time. In such systems the DTM emerges as a new mechanism to manage the negotiations and their results within all the nodes involved in the new distributed transaction. This section describes how to integrate this tool within the CORBA model and, in particular, in the PolyORB middleware.

The remote data types and services provided by the DTM must be described through the IDL language [16]. In particular, this has required the definition of:

- *Send_Message:* A remote call interface used to exchange the different kinds of messages between the DTM agents.
- *Idl_Dtm_Message*: A record used to store a generic message unit data. It consists of a header, indicating the message type, and a body, storing the specific data associated to a particular message type.
  - *Idl_Dtm_Request_Type*: An enumeration type to differentiate each of the 15 different types of messages to manage the initialization, the negotiation process and the status results.
  - *Idl_Dtm_Body_Message*: The specific data associated to a particular *Idl_Dtm_Request_Type*. Internally, it is an IDL union type which is mapped to a mutable discriminated Ada record [22] and therefore not optimized for memory requirements. However, the transmission of data is not affected by this kind of structure since the middleware will only send the amount of data strictly required through the network.
- Exchanged data types: Each kind of message has different data structures which will be exchanged between the agents. All of them must be described in IDL, including those consisting of complex types such as the contracts.

The initialization process may be configured statically as proposed in the original DTM or can make use of the CORBA Naming Service [19]. Each agent should register its IOR on it when it becomes ready to start or accept requests, and get the IOR from the remaining agents to assert that they are also in the ready state.

The *Routing Service* has been implemented in PolyORB within the GIOP protocol as shown in Fig. 4. Middleware requires to retrieve the *Event_Id* parameter stored in the GIOP *Service Context* field for any incoming message to identify the ongoing transaction. Once the transaction has been identified, the scheduling parameters can be updated if required and the middleware will try to locate the remote operation in the local node. If the location fails, then the message is routed through the appropriate send endpoint. Currently the implementation only supports routing using the same protocol personality (e.g., a FRESCOR network using RT-EP and CAN).

Fig. 4 represents a general overview of the DTM integrated into PolyORB. The current version is partially developed and uses PolyORB-CORBA, FRSH contracts for the scheduling policy, and RATO tasking policy (which supports the explicit creation of handler tasks and their association with endpoints) [21]. For the moment, it

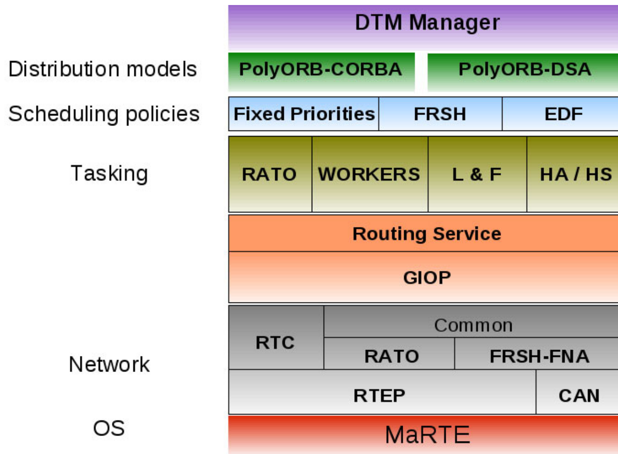

Fig. 4. DTM Manager integrated with middleware

only implements the basic contract parameters to specify the timing requirements and the proposed DTM protocol. A future version will integrate the PolyORB-DSA personality and any other resources managed by the framework.

We have evaluated the DTM over a platform consisting of four 800 Mhz embedded nodes connected through a 100 Mbps Ethernet with GNAT GPL 2008, MaRTE OS 1.9 and the modified version of PolyORB 2.4. Three tests have been run for two, three and four processing nodes. The tests consist of 10 independent asynchronous transactions, each one negotiating one contract for each processing node and one contract for each link over the network. Table 1 shows the times measured for these three tests. As it can be seen, the distributed negotiation process takes less than 20 ms for two processing nodes. In the cases of three and four processing nodes, this time is not increased significantly despite the higher overhead in the network and the extra negotiations of contracts. In spite of the fact that the DTM specification might seem at first complex and heavy we have found that we can get good performance time measurements.

Table 1. DTM metrics to negotiate one transaction using middleware (times in μs)

Num. of Nodes	Max	Avg	Min	Std. deviation
Two nodes	19508	19383	19133	111
Three nodes	25546	25219	24470	330
Four nodes	33255	32713	32565	199

8 Conclusions and Future Work

The paper presents the Distributed Transaction Manager (DTM) to extend the negotiation capabilities of the flexible contract-based scheduling framework defined in the

FRESCOR European Project. The DTM is intended as a middleware layer that allows the application components to perform remote negotiations and renegotiations of contracts, and a centralized management of the results of the negotiation process. The DTM has been successfully implemented in the FRESCOR framework demonstrating its usefulness.

As an extension of the basic capabilities, another implementation of the DTM has been proposed using a general purpose distribution middleware based on the CORBA and Ada DSA standards. This implementation provides full support for the transactional model and takes advantage of the distribution middleware to develop the DTM internals in a simpler way that is also more platform independent. It also extends some capabilities as for example the specification of the full transaction or the routing service that interconnects different networks over the standard layer of GIOP.

This features designed for DTM over the distribution middleware have been partially implemented. In the short term we plan to complete this implementation and evaluate its benefits and performance.

References

1. Aldea, M., et al.: FSF: A Real-Time Scheduling Architecture Framework. In: Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2006, San Jose, CA, USA (2006)
2. FRESCOR project web page: `http://www.frescor.org`
3. 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 (2008)
4. FRESCOR Architecture and contract model for integrated resources II. Deliverable (D-AC2v2), `http://www.frescor.org`
5. MaRTE OS web page, `http://marte.unican.es/`
6. Martínez, J.M., González Harbour, M.: RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet. In: Vardanega, T., Wellings, A.J. (eds.) Ada-Europe 2005. LNCS, vol. 3555, pp. 180–195. Springer, Heidelberg (2005)
7. FRESCOR Distributed Transaction Manager - proof of concept. Deliverable (D-ND5v1), `http://www.frescor.org`
8. FRESCOR Distributed Transaction Manager - Prototype Demonstration. Deliverable (D-ND5v2), `http://www.frescor.org`
9. López Campos, J., Javier Gutiérrez, J., González Harbour, M.: Interchangeable Scheduling Policies in Real-Time Middleware for Distribution. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 227–240. Springer, Heidelberg (2006)
10. PolyORB web page, `http://polyorb.objectweb.org/`
11. Vergnaud, T., Hugues, J., Pautet, L., Kordon, F.: PolyORB: a Schizophrenic Middleware to Build Versatile Reliable Distributed Applications. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063. Springer, Heidelberg (2004)
12. Pérez, H., Gutiérrez, J.J., Sangorrín, D., Harbour, M.G.: Real-Time Distribution Middleware from the Ada Perspective. In: Kordon, F., Vardanega, T. (eds.) Ada-Europe 2008. LNCS, vol. 5026. Springer, Heidelberg (2008)
13. AdaCore Technologies, The GNAT Pro Company, `http://www.adacore.com/`
14. Tijero, H.P., Gutiérrez, J.J., Harbour, M.G.: Support for a Real-Time Transactional Model in Distributed Ada. In: Proceedings of the 14th International Real-Time Ada Workshop (IRTAW 14), Portovenere (Italy). ACM Ada-Letters, New York (2009)

15. Loyall, J.P., Rubel, P., Schantz, R., Atighetchi, M., Zinky, J.: Emerging Patterns in Adaptive, Distributed Real-Time, Embedded Middleware. In: OOPSLA 2002 Workshop - Patterns in Distributed Real-time and Embedded Systems, Seattle, Washington (2002)
16. Object Management Group. CORBA Core Specification. OMG Document, v3.0 formal/02-06-01 (2003)
17. Object Management Group. Data Distribution Service for Real-time Systems. OMG Document, v1.2, formal/07-01-01 (2007)
18. Tucker Taft, S., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P. (eds.): Ada 2005 Reference Manual. LNCS, vol. 4348. Springer, Heidelberg (2006)
19. Object Management Group. Naming Service Specification. OMG Document, v1.3 formal/04-10-03 (2004)
20. Object Management Group. The Real-time Publish-Subscribe Wire Protocol. DDS Interoperability Wire Protocol Specification. OMG Document, v2.1, formal/2009-01-05 (2009)
21. Pérez Tijero, H., Javier Gutiérrez, J.: Experience in integrating interchangeable scheduling policies into a distribution middleware for Ada. In: Proceedings of the ACM SIGAda Annual International Conference on Ada and Related Technologies, SIGAda 2009, Saint Petersburg, Florida, USA (2009)
22. Object Management Group. Ada Language Mapping Specification. OMG Document, v1.2 formal/01-10-42 (2001)