A Real-Time Distributed Scheduling Service For Middleware Systems

Jiangyin Zhang, Lisa DiPippo, Victor Fay-Wolfe, Kevin Bryan, Matthew Murphy
University of Rhode Island, Department of Computer Science, Kingston, RI 02881 USA
{zhang, dipippo, wolfe, bryank, murphym}@cs.uri.edu

Abstract

The latest version of Real-Time CORBA defines a Distributable Thread primitive to support real-time computing in a dynamic distributed environment. However, this standard does not provide support for making and enforcing global decisions. This paper describes the framework for a Distributed Scheduling Service (DSS) that provides globally sound decision-making and scheduling enforcement to real-time distributed systems. The paper describes the design and implementation of the framework, as well as preliminary performance results.

1. Introduction

As theory and practice in distributed computing and in real-time computing matures, there is an increasing demand for automated solutions for dynamic distributed real-time middleware to support scheduling end-to-end timing constraints. The latest version of Real-Time CORBA (RTC), known as RTC1.2 (formerly known as RTC 2.0) [1], defines the Distributable Thread (DT) primitive to support real-time computing in dynamic distributed middleware systems. RTC1.2 provides a flexible means for expressing and propagating scheduling information across node boundaries in a distributed system. However, in RTC1.2, all scheduling decisions are assumed to be local – that is a local scheduler on each endsystem uses the same propagated scheduling information to make local scheduling decisions. These local schedulers do not have a global view of the overall system. This could lead to local enforcement decisions that fail to achieve maximum possible global system performance. For instance, consider a DT that spans three nodes in its end-to-end path. The first local scheduler in the chain would choose the RTC1.2 deadline as the deadline it must enforce. This scheduling decision could leave insufficient time for the remaining segments of the DT to execute on the subsequent nodes. What is missing is a scheduling decision that provides globally sound scheduling parameters to the local schedulers in the end-to-end chain.

In addition to lacking support for global scheduling decisions, most distributed real-time middleware does not provide adequate support for overload management [2]. Overload management is a function that the middleware must perform when timing constraints in one or more DTs cannot be met. In static real-time systems, overload management analysis can often be done a priori. In dynamic systems, where such offline analysis is either not possible or can not be as comprehensive, it is essential that the middleware identify and address overload in a runtime setting. RTC1.2 provides a cancel mechanism [1] that allows a DT to be cancelled. This overload management primitive, does not incorporate a global understanding of which DTs to cancel, nor does it address the overall effects of these cancelled DTs on the system. Other overload management techniques, such as Quality of Service adjustment [2], should also be managed using globally sound parameters and criteria.

This paper describes the design and implementation of a Distributed Scheduling Service (DSS) framework that works with application specified end-to-end scheduling parameters and with local scheduling mechanisms to make globally sound scheduling decisions for the system. We are currently implementing the framework in a Real-Time CORBA 1.2 environment, but have designed it to be applicable to other middleware systems that require global scheduling management. The goal of DSS is to achieve globally sound end-to-end scheduling and overload management using the local enforcement capabilities of the local endsystems.

2. Related Work

The DSS described in this paper is a framework for distributed global scheduling. One of its goals is to provide a mechanism for implementing some of the classic end-to-end scheduling algorithms that have been developed. This section describes and compares

Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)
0-7695-2347-1/05 $20.00 © 2005 IEEE

these algorithms. The section also discusses the features and deficiencies of existing distributed scheduling frameworks.

2.1. End-to-end Scheduling Algorithms

End-to-end scheduling theory has provided much the motivation for the development of the DSS framework. It is this theory that has lead to the notion that local scheduling should incorporate a global view of the overall system. Several researchers have recognized the need to compute intermediate deadlines when an application specifies a single, final deadline on an end-to-end task [3][4]. Various algorithms have been developed to compute the intermediate deadlines to maximize the possibility of all end-to-end tasks in the system meeting their specified deadlines. Ultimate Deadline uses the end-to-end task's relative deadline as the intermediate deadline for every subtask. This is the simplest computation for intermediate deadlines, and can lead to problems if the first subtask in the end-to-end task uses all of the available slack. Effective Deadline calculates the subtask's intermediate deadline by subtracting the sum of its successors' execution time from the end-to-end deadline. This algorithm puts all of the slack in the last subtask of the end-to-end task, so the intermediate deadlines of the earlier subtasks may be too restrictive. Proportional Deadline lets the relative deadline be proportional to each subtask's execution time. In another words, the more execution time, the longer intermediate deadline. Normalized Proportional Deadline improves upon Proportional Deadline by taking into account each processor's utilization such that subtasks on busier processors will be assigned longer intermediate deadlines. The computation of intermediate deadlines is an important global scheduling decision that cannot be done easily with local scheduling alone. We chose the Effective Deadline approach for our implementation of the deadline assignment algorithm in our DSS framework because of its simplicity and good performance results. The DSS is designed to be pluggable so that any deadline assignment algorithms may be substituted.

Another important global scheduling decision that the DSS framework must make involves how to synchronize the individual subtasks in the end-to-end task. Several synchronization protocols have been developed to ensure that subtasks are executed in the correct order. A greedy synchronization protocol requires the immediate release of a subtask when its predecessor completes [3]. Most Real-Time CORBA implementations use a greedy approach where servant threads are launched as soon as possible when a request arrives. Greedy synchronization can make the system of end-to-end tasks difficult to analyze because the start times of subsequent subtasks are not periodic. Instead, they rely on the end times of the previous subtasks. Non-greedy synchronization protocols, such as the Release Guard protocol [3], forces each subtask to be periodic by adjusting the release time of each job in the subtask. Forcing periodic behavior not only improves global worst-case response times over the system, it also facilitates analysis of the timing of end-to-end tasks. Our DSS framework implements a non-greedy synchronization protocol for this reason.

2.2. Dynamic Scheduling Frameworks

Other frameworks have been developed that are similar to our DSS framework, but none encompasses all of the important features that ours provides. Tempus is a middleware framework that supports the Distributable Thread construct [5]. The analysis theory for Tempus is based on Time/Utility Functions [6]. In Tempus, the DT propagates scheduling parameters to each local node. There is no global decision-maker. Instead each local scheduler makes the scheduling decision based only on the scheduling parameters that each DT carries. The QuO framework also supports dynamic distributed scheduling [7]. However, QuO does not provide a standard way to do so. It relies on each application's specific configuration. Recent research done at CMU integrates scheduling and resource management to enhance performance in phased-array radar systems [8]. Although the system architecture in this work considers scheduling and resource management together, they do not consider end-to-end tasks distributed across different processors. While each of these frameworks provides valuable features to support real-time systems, none provides all of the key elements that the DSS provides. Specifically, the DSS provides a global awareness of the system to allow it to make scheduling decisions that have the interest of the overall system in mind; the DSS provides an application independent overload management mechanism; and the DSS provides a natural extension of the RTC 1.2 standard with minimal changes to the standard interface.

2.3. Real-Time CORBA 1.2

Real-Time CORBA is a QoS enabled extension of CORBA middleware. The Real-Time 1.1 specification is designed for static distributed system where the number of tasks and their scheduling parameters are known a priori. The Real-Time CORBA 1.2 specification extends RTC1.1 to encompass both static and dynamic systems. In a dynamic system, tasks enter

and leave the system at times that cannot be calculated a priori. In order to effectively manage the dynamic task set, RTC1.2 introduces the Distributable Thread scheduling primitive. A DT is an abstraction of a chain of method calls by multiple threads at multiple processors. According to its definition, a DT can span nodes boundary and carry scheduling parameters to each node in the chain. At each local node, the correspondent local scheduler will schedule that DT based on its scheduling information. The interface from the DT to the local scheduler is well defined. The begin_scheduling_segment() (BSS), update_scheduling_segment() (USS) and end_scheduling_segment() (ESS) are the three most important operations in the interface. BSS begins the Real-Time portion of the program, and passes the application provided scheduling parameters to the local scheduler. The local scheduler then determines a schedule. The schedule is enforced until BSS, USS or ESS is encountered, forcing a change in the schedule. Two essential arguments for BSS are a name and a scheduling_parameter. The name identifies scheduling segment. The application defined scheduling parameters reflect the scheduling algorithm. USS updates the existing scheduling parameter of a DT. ESS indicates the end of the scheduling segment, which means that the Real-Time region of code is finished.

TAO [11] is an open-source middleware implementation that supports the RTC1.2 standard. Currently, TAO implements two local schedulers. The first is a fixed-priority local scheduler that expects to be supplied with priorities for the DTs that it schedules. The second is Kokyu [12]. Kokyu is a flexible middleware framework for managing threads on various platforms. As a local scheduler for TAO, Kokyu has implemented several well-known scheduling disciplines including Fixed Priority, EDF, MIF and MUF.

3. Design of DSS

Figure 1 depicts our overall system architecture. There are six essential components in this framework, Distributable Thread (DT), Local Scheduler, DSS Proxy, DSS, Resource Manager (RM), and System Repository. These components are independent and coordinated with each other.

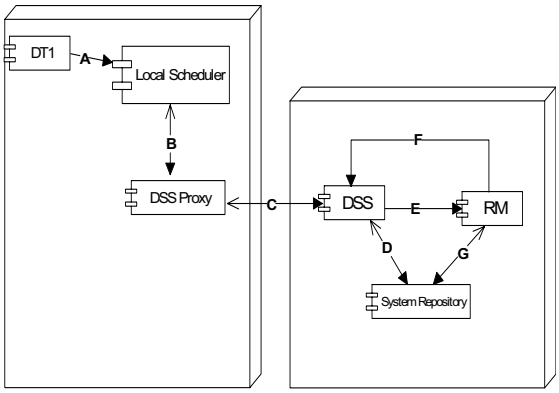

Figure 1. System Architecture

A *Distributable Thread* (DT) is the schedulable entity in our system architecture. When a DT is spawned by the application it carries its specified scheduling parameters, including its end-to-end deadline, along as it traverses the nodes in its path. The Local Scheduler is defined in RTC1.2 to manage the local portion of a DT. In our architecture, we extend the definition and allow the local scheduler to interact with both the DT and the DSS Proxy, so that the local scheduler can obtain and use global information. Applying the Interceptor pattern we put a wrapper around the RTC1.2 local scheduler to intercept calls between DT and DSS Proxy. The Local Scheduler can implement various scheduling mechanisms, such as earliest deadline first, fixed priority, and deadline monotonic. The DSS Proxy is a running daemon that works as a proxy to the DSS and is always located on the same node as Local Scheduler. This collocation reduces the overhead of making remote calls to the DSS whenever a global scheduling decision is required. The DSS is a centralized scheduling service that has several responsibilities: 1) online schedulability analysis of an end-to-end task; 2) computation of globally sound scheduling parameters; 3) triggering of overload management if necessary. When the system becomes unschedulable the Resource Manager (RM) applies overload management solutions such as QoS adjustment. The System Repository stores system information that is shared between the DSS and the RM. This information includes end-to-end tasks, nodes on which they will execute, and their scheduling parameters. The synchronization is enforced at each local scheduler that implements any of the synchronization protocols.

When a DT is spawned by an application, the DT communicates with the DSS to determine if it is schedulable alongside the existing DTs in the system, and receives from the DSS its globally sound scheduling parameters. In RTC1.2, the interface specifies that the DT pass its scheduling parameters to the Local Scheduler. We have preserved this interface, and extended the Local Scheduler to allow it to send these parameters to the DSS Proxy. The DSS Proxy then makes a remote call to the DSS, which returns the globally sound scheduling parameters to be returned to the DT. Sometimes the DSS Proxy may have enough information to make the global scheduling decisions without making a remote call to the DSS. For

example, during some idle time of the CPU, the DSS Proxy has a chance to call DSS to get the global decision to be cached in the DSS Proxy, then when a DT call DSS Proxy, it will get this information without making a remote call. If the DSS determines that executing the DT will make the system unschedulable, it invokes an overload management mechanism. Possible overload management mechanisms include denial of the DT's request, cancellation of an existing DT, and QoS-based adjustment of the system. If the RM adjusts the parameters of any DTs in the system, it updates the System Repository so that the DSS will have accurate system information.

Figure 1 not only shows the components in the system architecture, but also labels the interactions between the components. Label A shows the invocation of an application thread to its local scheduler. This invocation is made for three different operations: BSS, USS, and ESS. The first essential step for a DT to interact with the DSS is to invoke one of these operations. A DT first sends its scheduling information to a local scheduler whenever the DT makes a request to begin, update or end a scheduling segment. The parameters passed along are determined by the scheduling discipline chosen by both DT and the local scheduler such as RM, DM, EDF and MUF. If the DT spans multiple nodes our design dictates that it must pass an end-to-end deadline and a sequence of subtasks to the local scheduler. If the system does not have DSS the local scheduler applies its own scheduling mechanism, such as a priority-based mechanism, to do the schedule for the DT. However in our design, this scheduling parameter passed by the DT is not used by the scheduler's mechanism to schedule the DT. Instead, it is propagated further to the DSS Proxy. At Label B, the DSS Proxy applies the well-known Remote Proxy pattern [9] to provide a local representative for an object that resides in a different address space. After the DSS Proxy gets the scheduling information from the local scheduler, it can either return back the desired information, for example, a cached cancel message, or hand the scheduling parameter further to DSS has the parameters from a DT and has the information necessary in the System Repository to perform a schedulability analysis. If the system is schedulable, DSS returns globally sound parameters, such as intermediate deadline, back through steps C and B. At point D, the DSS stores the DT's scheduling parameters in the System Repository, associating it with a globally unique id. This procedure is recognized as the Local Enforcement pattern [2], which implies that a globally meaningful scheduling parameter is enforced locally.

If the system is not schedulable, (i.e., the system is in an overloaded situation) the DSS triggers the overload management module, which is designed using Overload Management patterns [2]. There are four patterns that deal with overload. Two of these patterns, Deny Request and Cancel, are handled by DSS. The other two patterns, QoS Adjustment and Resource Reallocation, are handled by RM. If DSS denies an incoming DT or cancels an existing DT, it throws back a "Not Schedulable" or a "Cancellation" message to a DT through step C, B, A. If DSS elects to adjust QoS to reallocate resources, it invokes the appropriate methods in the RM's interface labeled as step E. At step F, RM has finished performing overload management and returns its results to DSS so that the DSS may reanalyze the system. This E-F-E loop repeats until a feasible solution returns the system to a schedulable state. This cycle represents the integration of DSS with RM [10]. At step G analysis is complete and the RM puts the current system information back into System Repository.

4. Implementation

The DSS is built upon TAO [11], the widely used open source CORBA environment. There are two versions of DSS that has been developed. The first one is a DSS that can set priority for a Fixed Priority local scheduler. This version of DSS takes the scheduling parameters from a DT and calculates a globally sound priority for each local scheduler. The second version of the DSS sets intermediate deadlines for a Deadline Monotonic (DM) local scheduler. In this version the DSS uses the end-to-end deadline and subtask execution times to calculate an intermediate relative deadline for each subtask. In the discussion of scheduling points below, the local enforcement mechanism is a DM local scheduler. We chose the DM local scheduler as our local enforcement mechanism because it uses a fixed priority to reduce the online scheduling overhead and DM is optimal among fixed priority scheduling algorithms.

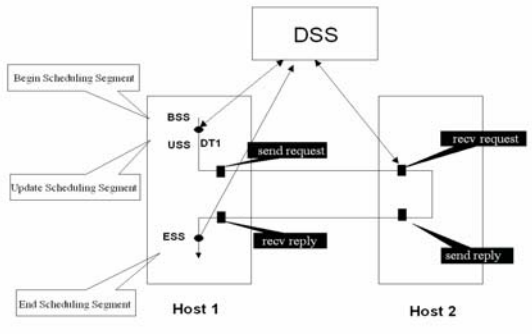

Figure 2. RTC1.2 Scheduling Points and DSS

In RTC1.2 scheduling points are the points in time and/or code at which the local scheduler is invoked by the application, which may result in a change in the current schedule. Figure 2 shows that all seven scheduling points which may have interactions with our DSS. The seven scheduling points are Begin_Scheduling_Segment (BSS), Update_Scheduling_Segment (USS), End_Scheduling_Segment (ESS), send_request, receive_request, send_reply, and receive_reply. In our current implementation, we use four of the seven scheduling points, BSS, USS, ESS and receive_request, to interact with the DSS. At BSS, the DT sends all of its scheduling parameters to the DSS. These scheduling parameters include a system wide unique name of the DT and a sequence of scheduling parameters for subtasks. The scheduling parameter for a subtask contains a processor name that the subtask will execute on, an execution time for that subtask and an intermediate deadline. DSS calculates a globally sound intermediate deadline using the Effective Deadline algorithm. For example, consider an end-to-end DT that has a relative deadline of 10 and two subtasks. Subtask1 is assigned in Node1 and Subtask2 in Node2. The execution times for Subtask1 and Subtask2 are 1 and 2 respectively. By Effective Deadline, the intermediate relative deadlines for Subtasks are 8 and 2. Rather than using a deadline of 10 for all subtasks, the local scheduler on node 1 will use an intermediate deadline of 8 and the local scheduler on node 2 will use an intermediate deadline of 2.

The implementation for USS scheduling point is similar to BSS. USS is used when an existing DT requires a change in its scheduling parameters. For instance, if a QoS adjustment has shortened the intermediate deadline for a subtask, the DT will invoke USS and update its scheduling parameters. Any such changes to scheduling parameters may require a reanalysis of the system, and changes to the information stored in the System Repository. The ESS scheduling point is simply sending a message that this DT is no longer in the system, therefore DSS can remove its information from System Repository. The receive_request scheduling point allows a subtask on a new node to capture an incoming request from its predecessor subtask, and then request scheduling information from the DSS. In this case, the subtask will not pass all of the scheduling parameters to the DSS, but only its unique name. According to this name, the DSS will calculate the intermediate deadline for this subtask, and return it to the subtask.

We implemented a DM RTC1.2 local scheduler using TAO [11] the widely used open source CORBA environment. This DM scheduler utilizes Kokyu [12] as its local scheduler and dispatcher. Kokyu is a very flexible middleware framework for managing threads on various platforms. Kokyu has implemented several well-known scheduling disciplines including Fixed Priority, EDF, MIF and MUF. Since Kokyu does not provide an implementation of DM local scheduler, we revised the provided MUF local scheduler to be a DM local scheduler by setting the criticality to be a function of the relative deadline and added operations at each scheduling points mentioned above to call DSS. Besides the flexibility, Kokyu schedulers are the only local schedulers that implemented Release Guard [3] synchronization protocol that forces each subtask to be periodic, allowing low overhead online analysis.

The focus of our current implementation is to set up the DSS framework. The DSS Proxy and Overload Management mechanisms such as Deny Request and Cancel are still in the design phase. Once the basic functions of DSS work smoothly with DM local scheduler, we will move to implement the DSS Proxy and Overload Management mechanisms. One challenge in implementing Cancel is where to propagate the cancel message to the DT. The possible places for this cancel message to be sent are 1) the head of the DT defined by RTC1.2, which is the current execution point of the DT, 2) the beginning of the DT, 3) all the DSS Proxies by a broadcast. Because this framework is aiming to support a better real-time behavior, a timely cancel message is important to achieve our goal. Thus a deeper investigation of the RTC1.2 cancel mechanism and an efficient cancel algorithm is required. Another improvement can be done is to let DSS interact with send_request and send-rely to optimize network behavior.

5. Experimental Evaluation

We have performed two types of experiments on our DSS framework. The first is a set of evaluation tests using a version of the DSS that interacts with the Resource Manager and local schedulers. The second set of tests was implemented in the RTC1.2 framework to evaluate the benefit of using the DSS over using local schedulers alone. In this section we describe the results of these tests.

DSS with RM Tests. These tests demonstrate that the DSS with the RM improves real-time performance. The full results of these tests have been reported in [10]. The experiments were based on a distributed video delivery application for Unmanned Aerial Vehicles (UAVs). The UAV application consists of senders, which acquire and send out video frames;

viewers, which either display the video to human monitors or feed the video to automatic target recognizers; and distributors, which replay the videos frames from senders to viewers. In this experiment, we collected data from a simulated UAV application that runs in RMBench [10], a tool designed for performance evaluation of real-time embedded systems. The experiment consisted of six periodic end-to-end tasks, which were sender/distributor/viewer chains. Tasks 1 through Task 6 were assigned importance 1 through 6 (1= lowest, 6 = highest), respectively. Each task consisted of three sub-tasks: sender, distributor, and viewer. Each task could be run at 10 different service levels, with level 10 providing the highest video quality and level 1 the lowest. We ran three instances of the experiment. The baseline experiment did not use the RM or the DSS to control timeliness. The RM experiment used only the RM, which reacted to host overload. The DSS / RM experiment used the integrated DRM and DS services.

The results indicate that the use of the RM and the DSS improve both the latency and the number of missed deadlines significantly for this experiment, but at the cost of reducing the average quality of the less important active distributed task. Employing the DRM service alone decreased the number of missed deadlines from 26% to 10%. The integrated DSS/RM reduced the number of missed deadlines to less than 1%.

RTC 1.2 Tests. The current DSS framework is implemented to support the Deadline Monotonic scheduling algorithm. It uses the RTC1.2 Fixed Priority local scheduler as its local enforcement mechanism. The DSS framework is implemented with three possible deadline assignment heuristics: Ultimate Deadline (UD), Effective Deadline (ED) and Proportional Deadline (PD). The greedy synchronization protocol is used for subtasks in this experiment.

The experiment depicted in Figure 3 is set up with two nodes. Each node has two tasks running. Task 1 and Task 3 reside on different nodes are only doing local execution. Task 2 is an end-to-end task spanning both nodes. All the time values in the experiments are in seconds. This experiment is not meant to be a comprehensive study of the scheduling framework. Rather, it is a proof-of-concept to indicate that the DSS framework can implement the various intermediate deadline assignment algorithms, and the results will be as expected based on the theory.

Table 1 shows the scheduling parameters for the tasks in the experiment. These parameters were borrowed from [3] because this set of tasks was shown to be non-schedulable when intermediate deadlines were not computed, but schedulable under both ED and PD.

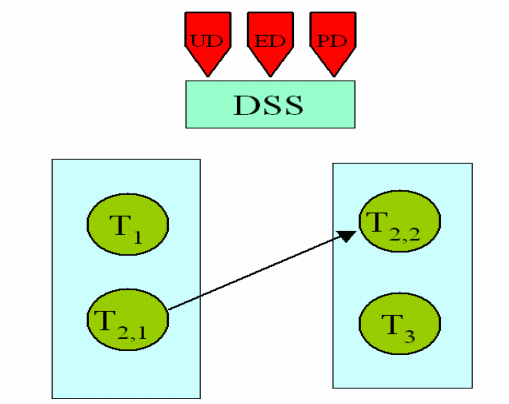

Figure 3. Experiment Setup

Table 1. Task set with scheduling parameters

Task	Host	Period	Etime
1	P1	8	3
T2,1	P1	10	5
T2,2	P2	10	3
T3	P2	4	1

Table 2. Relative Deadline Assignment

Task	w/oDSS	DSS -UD	DSS -ED	DSS -PD
T_1	8	8	8	8
$T_{2,1}$	10	10	7	6
$T_{2,2}$	10	10	3	3
T_3	4	4	4	4

Table 3. Percentage of missed deadlines based on 100 periods.

Task	RTC1.2	DSS - UD	DSS - ED	DSS - PD
T1	0	0	0	0
T2,2	99	99	0	0
T3	0	0	0	0

Table 2 shows the deadlines for each of the tasks, including the intermediate deadlines that were computed by the various algorithms. Table 3 shows the results of the experiments. It compares the percentage of missed deadlines for the various deadline computation algorithms that the DSS has implemented. With ED and PD, intermediate deadline assignment by DSS, none of the tasks missed their deadline. In contrast, without DSS or using UD 99% deadlines were missed. The results showed that the DSS is able

to implement these known deadline computation algorithms, and the results are as expected.

6. Conclusion

In this paper, we presented a Distributed Scheduling Service (DSS) framework for Real-Time CORBA 1.2. In the design of the service, a global view of the system is especially emphasized. By incorporating DSS into a RTC1.2 system, global scheduling decisions can be made that take into account not just each endsystem's requirements, but the requirements of the entire system. The design of the DSS does not interfere with the application's interface with the RTC1.2 system. Through the use of well-known design patterns, the DSS design provides a seamless way to incorporate global information into the RTC1.2 system. The current implementation supports four out of seven RTC1.2 scheduling points at which the DSS is invoked to provide global scheduling information. Future implementations will support all of the seven scheduling points.

The results of the evaluations tests that we performed indicate that DSS can be beneficial in determining when resource management is necessary, and in helping to implement proper strategies. The test results also showed that the DSS framework can implement various well-known algorithms that have been proven in theory to enhance the schedulability of end-to-end tasks in a real-time distributed system.

7. References

[1] Real-time CORBA (Dynamic Scheduling) specification, version 1.2, Nov., 2003, http://www.omg.org/cgi-bin/apps/doc?formal/05-01-04.pdf.

[2] L. DiPippo, V. Fay-Wolfe, J. Zhang, M. Murphy, P. Gupta, "Patterns in Distributed Real-Time Scheduling", Proceedings of the 10th Conference on Pattern Language of Programs 2003, Monticello, IL, Sept. 2003

[3] J. Sun, "Fixed priority scheduling of end-to-end periodic tasks", Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[4] B. Kao,H. Garcia-Molina: "Deadline Assignment in a Distributed Soft Real-Time System," in the proceedings of the 13th International Conference on Distributed computing Systems. 1993.

[5] P. Li, B. Ravindran, H. Cho and E. D. Jensen, "Scheduling Distributable Real-Time Threads in Middleware," IEEE International Conference on Parallel and Distributed Systems, pages 187 - 194, July 2004

[6] E. D. Jensen, "Asynchronous decentralized real-time computer systems," in Real-Time Computing, W. A. Halang and A. D. Stoyenko, Eds., NATO Advanced Study Institute. Springer Verlag, October 1992.

[7] Praveen K. Sharma, Joseph P. Loyall, George T. Heineman, Richard E. Schantz, Richard Shapiro, Gary Duzan. Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems. International Symposium on Distributed Objects and Applications (DOA), Agia Napa, Cyprus, October 25-29, 2004.

[8] S. Ghosh, R. Rajkumar, J. Hansen, and J. Lehoczky, "Integrated Resource Management and Scheduling with Multi-Resource Constraints," In Proceedings of 25th IEEE Real-Time Systems Symposium, 2004.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Adisson Wesly, Riding, MA, 1995.

[10] K. Bryan, L. C. DiPippo, V. Fay-Wolfe, M. Murphy, J. Zhang, D. T. Fleeman, D. W. Juedes, C. Liu, L. R. Welch, D. Niehaus, and C. D. Gill, Integrated CORBA Scheduling and Resource Management for Distributed Real-Time Embedded Systems, RTAS 2005, San Francisco, California, March 7 - 10, 2005.

[11] D.C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-performance Endsystem Architecture for Real-time CORBA," IEEE Communications Magazine, February, 1997.

[12] C. Gill, D. Levine, D. C. Schmidt, and F. Kuhns, "The Design and Performance of a Real-Time CORBA Scheduling Service," International Journal of Time-Critical Computing Systems special issue on Real-Time Middleware, 1999.