Distributed Adaptive Scheduler for QoS Support in Compose|Q

Shivajit Mohapatra & Nalini Venkatasubramanian
University of California, Irvine
{mopy,nalini}@ics.uci.edu

Abstract

In this paper, we present the design and implementation details of a flexible reflective scheduling framework, that supports conjunctive scheduling of both tasks and messages within a distributed message based environment. In future, distributed environments will need to fine tune their systems to provide diverse services, oftentimes implementing dissimilar policies and functionality. We understand that future distributed systems would require their schedulers to be tailor-made or customizable to suit the diverse workloads at different times. The framework is therefore fashioned to provide both coarse and fine grained scheduling, for better tunability and improved performance. Though this model is designed to work with any thread based system, we have investigated the applicability of these concepts on actors (active objects) within the *Compose|Q* framework. Scheduling of soft real time tasks are handled by the framework to conform to guarantees, even in the presence of normal time-sharing tasks. We expect that the proposed solution would be scalable while providing higher flexibility than simple task based scheduling.

1 Introduction

The advances in networking and hardware technology has spawned a dramatic growth in middleware systems for distributed environments. Distributed Middleware Systems are becoming increasingly pervasive and effective in information access and dispersion, often in real time. Middleware solutions [21] require minimal changes to be made to the underlying systems, thereby eliminating most portability issues, and attempt to provide high level customizations, adaptations and quality of service guarantees. As middleware systems become increasing complex in order to accommodate diverse system and user requirements, several design challenges arise, in particular issues concerning adaptation and customization in such dynamic environments. This raises the challenge of creating flexible, customizable middleware services for systems that have minimal support for timeliness and dependability guarantees. The scheduling framework forms an integral part of any distributed system, and flexible scheduling mechanisms are critical to the design of customizable middleware frameworks.

With our framework, we address the challenge of providing a customizable and adaptive scheduling mechanism for distributed messaging environments. Our solution is exclusively a middleware solution requiring no changes to be made to underlying native schedulers. The overall goals and the key challenges for our scheduler are to provide: 1)Improved performance of the overall distributed system 2)Predictable and guaranteed operations for individual tasks. Timeliness and QoS guarantees for soft real-time applications. 3) Adaptivity to changing system and network conditions. 4) Coarse and fine grained customizations to fine tune the behavior of the scheduler. 5) Support for varying Quality of Service requirements. In the middleware context, we extend notion of scheduling to include the scheduling of tasks within nodes (peripheral scheduling) and the scheduling of tasks across nodes in a distributed environment(focal scheduling). The focal and the peripheral schedulers can collude to provide the overall scheduling for the system. Any customizations made to the focal scheduler would result in a system wide change in the scheduling behavior, which we call coarse grained scheduling. Changes made to the peripheral schedulers would provide a more fine grained scheduling, exploiting the capacity and resource availability of that peripheral node.

It is widely accepted that adaptability is indispensable to wide area distributed environments. Consider a distributed environment that provides a variety of multimedia services(video servers, language translators, video conferencing) on general purpose machines without any dedicated operating system or hardware support for the co-existence of real-time applications and traditional time sharing applications(word processors, compilers and browsers). In this distributed environment,

systems are heterogenous with varying resource availabilities; clients have diverse performance requirements and computing power. The onus of providing uninterrupted services and guarantees in such a dynamic system environment now lies on the middleware. In addition to handling variations in system conditions, adaptation can be used to address application QoS requirements. Middleware scheduling services can be used to exploit native scheduler properties to provide real-time support, fair scheduling and performance benefits over normal priority based scheduling. Additionally, middleware modules can be effectively used to implement load-balancing policies.
By *customizable scheduling* we mean that the scheduling properties of the system can be dynamically moderated; the various "scheduling" related policies could be altered on the fly, in order to fine tune the system for optimal functioning. Understandably, the overall system schedulability can be bettered if the scheduler(s) had the ability to switch seamlessly between underlying soft real-time scheduling policies [4](e.g Rate Monotonic to EDF); to dynamically change the priority levels of task execution without violating any QoS or real-time commitments, adapt to transient overloading of systems and the capacity to handle special scheduling situations.
The next section introduces the scheme we use for conjunctive priority assignments to both tasks and messages. In the later sections we present the design outline and implementation issues in our scheduling framework within the $Compose|Q$ framework.

2 Conjunctive Priority Assignments

We now present a generalized policy for fixing message and task priorities in any distributed messaging environment.

2.1 System Priorities

Let the underlying distributed environment be composed of homogeneous nodes, each of which supports P priority levels. If the scheduler is implemented on an intermediate platform such as the Java virtual machine then the above assumption can be made without loss of generality. Otherwise, P could be set to the number of priority levels supported by the node having the least priority levels. Therefore a task running on any node can have a priority from zero to *P-1* .

2.2 Tasks

The individual tasks within a mode are grouped based on their absolute priorities. Typically, all realtime tasks would have highest priorities, followed by the high priority tasks and so on. With this approach, tasks with similar importance get bracketed into the same group. Moreover, most operating systems support only a limited number of priority levels. Collecting tasks into groups also allows for efficient use of the underlying operating system priorities.
Let there be G_{max} such groups.For each group G_i assign a static priority to the group. P(G_i) = $BASE_i$.

2.3 Real Time tasks

All the realtime tasks that are admitted for scheduling into a node are clubbed together into the realtime group. The realtime group is assigned an absolute priority, such that no real time task can ever be preempted by a task from a lower group. Also, the message priorities are ignored for the real time tasks. All the realtime tasks are of the same priority and the scheduling mechanisms used for these tasks are different from the scheduling methods for the all the other tasks. The realtime tasks are scheduled by the RT sub scheduler (ASRT) described in the next section.

2.4 Messages

Priorities are introduced into the individual messages. These priority levels are logical and are introduced by one of the runtime components and have no relation whatsoever with the underlying system priorities. The scheduler however makes use of the message priorities to translate individual task priorities. For example, if two tasks within the same group get two messages with different priorities, then the task receiving the high priority message needs to get scheduled before the other task.
Based on their priorities the messages can also be grouped together. Messages of the same importance can be put into the same group. Eventually, message priorities are translated by the scheduler into system priorities, so batching messages into groups also helps in the efficient use of the underlying system priorities. Let there be L_{max} message priority levels within the system. For each L_i ($1 <= i <= L_{max}$) assign a priority to each message P(L_i) = $Boost_i = i - 1$.

2.5 Priority Translations

After fixing the individual priorities of the tasks and the messages we are now in a position to determine rules that would square up the final priority assignments to

individual tasks. These priority assignments would carry the combined weights of the individual task and message priorities and would ensure that the appropriate task gets scheduled. Presented below are some of the basic rules that perform the priority translations.

Rule 1 Aggregate Priority of a Task:
AP($ActorG_i$,$MsgL_k$) = $Base_i + Boost_k$

Rule 2 The Real-Time Group gets an absolute priority over all the other groups. This means that the aggregate priority of the realtime group is higher than the sum of the priorities of the next highest priority group and the highest message priority. With this no normal task can interfere with the schedule for the realtime tasks.
$P_{GRT} > P_{GHP} + Boost_{Lmax}$

Rule 3 For other groups its a policy decision. Depending on the different task priority levels and the different message priority levels we can decide how to make the absolute priority assignments. For example, we can decide on a policy that states that a highest priority message to a task in Group A_{i-1}, would make that task get a higher priority than a task in Group A_i which has received the lowest priority message.

Obviously, the values of G_{max} and L_{max} are restricted by the number of priority levels of the underlying native systems.

3 The Customizable Scheduling Framework

In most distributed environments, the individual tasks running on the machines are candidates for CPU scheduling. In message based systems, these tasks communicate with each other via messages. A single task can send and receive messages from any number of independent tasks, running either on the same or on different machines. In such a scenario, the messages are processed in the order of their arrival at the destination task and all messages are assumed to carry the same importance. To provide a more fine grained notion of priority we propose to introduce priorities into individual messages in the system. The different tasks have priorities assigned to them when they start out and the task priorities are independent of the individual message priorities. The scheduling framework provides application actors methods for setting message priorities and also monitors the assignments so that they conform to the acceptable priority levels. With the knowledge of priorities of the individual messages and the destination tasks, we can combine the message and task priorities to come up with unique task priority assignments that can be used to optimally schedules tasks (both real-time and non real-time), delivering a stricter notion of priority scheduling. This approach could significantly change the way we provide adaptability and customizability and introduces a new perspective to schedulers in distributed environments. The scheduling framework proposed here tries to achieve the overall scheduling goals described above, within a distributed framework that employs active objects and the messaging based communication paradigm. The model is designed as a two-tier architecture, with the following primary components(Fig.1):

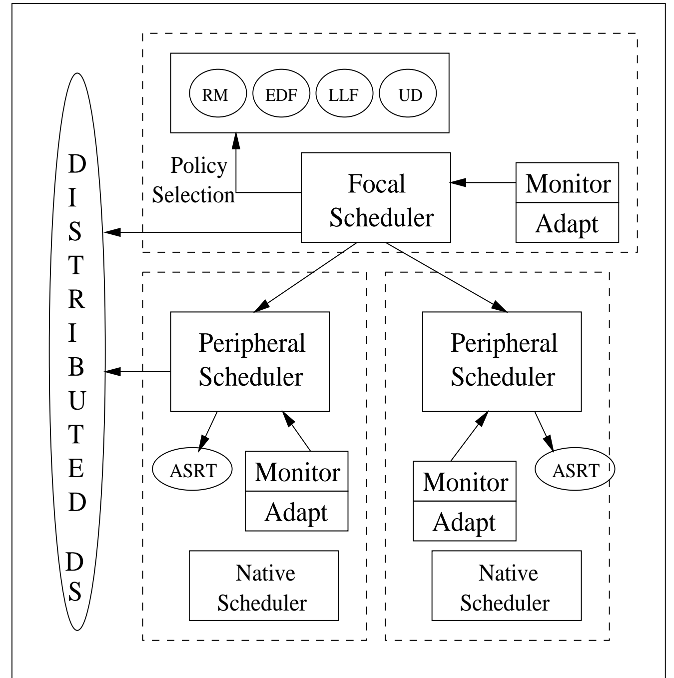

Figure 1: Two Level Meta Scheduler

FMS (Focal Meta Scheduler) - The focal scheduler is a system wide global scheduling entity, that is responsible for high level decisions and adaptation that ultimately determine the scheduling characteristics of the system.

PMS (Peripheral Meta Schedulers) - The peripheral or local meta schedulers are system level entities that run on the individual nodes (as a part of the node runtime) of the distributed system. They schedule the independent tasks on that node and are also responsible for realizing the conjunctive scheduling mechanism.

The various components of the two level meta scheduler are shown in Fig.1 above. Both the focal and peripheral schedulers have monitoring and adaptation modules built into them. The monitoring modules check for task compliance. The adaptation module makes use of this information to make intelligent adjustments that could improve the overall system performance. Moreover, each peripheral scheduler can be logically considered to have two components for scheduling the real-time and the non real-time tasks. The proposed architecture handles real time tasks differently from the time-sharing tasks, and it is therefore reasonable to separate them into different parts.

4 Design Outline

Compose$|$*Q* [23] is a QoS-enabled reflective middleware framework that is currently under development at the University of California, Irvine. It is based on a meta-architectural model that facilitates specifying and reasoning about the composability of multiple resource management services in Open distributed systems. *Compose*$|$*Q* uses actors (active objects) as individual application and system entities and the actors communicate through asynchronous message passing. We have attempted to integrate our conjunctive scheduler with this framework, in order to study its applicability and practicality in message based environments. This section presents the different components of the scheduling model and introduces our rationale for estimating a combined task and message priority that is eventually used for scheduling the tasks. Using the combined task and message priorities we can arrive at a number of policies that dictate the overall scheduling behavior of the system.

The scheduler is designed in a two-tier architecture with a system wide unique focal scheduler and a peripheral scheduler on each distributed nodes. An obvious extension is to make the focal scheduler distributed, to avoid a single point of failure. To simplify matters, we currently assume that the focal scheduler is always running and never fails. The failure of a peripheral node is directly going to affect the tasks on that node and no guarantees can be made for tasks running on that node. Upon recovery, the peripheral node announces its availability to the focal scheduler and all the existing tasks on the node are considered lost.

4.1 The Focal Meta Scheduler

The focal scheduler forms the core of the reflective scheduler and incorporates the policies that determine the overall scheduling property of the distributed environment. It is responsible for the scheduling policies for the peripheral nodes, and for fixing the different message and task priority levels. Traditional real-time scheduling algorithms (Rate Monotonic, Earliest Deadline First etc.) can be used in conjunction with priority scheduling to provide support for both real-time and the time sharing tasks. The FMS should be able to dynamically switch between these policies to fully exploit the system resources and maximizing performance. In a system that allows for migration of tasks from one node to another, it also has to maintain the rules for dynamic translation of scheduling parameters from one policy to another. The focal scheduler maintains information about the global resource availability and current commitments of individual nodes by way of the directory service. It therefore is in the best position to schedule tasks onto the individual nodes. All incoming tasks are routed to the individual nodes through the focal scheduler. Each task specifies its resource and QoS requirements to the focal scheduler; the scheduler then polls the peripheral nodes that have the necessary resources to check whether the new task can be admitted. From the set of nodes that can admit the task, it can then choose a node (using some optimal algorithm) that optimizes the resource utility and maximizes the task acceptance count into the system.

4.2 The Peripheral Meta Scheduler

The peripheral meta schedulers run on the individual nodes(for dispatching individual tasks) and are responsible for realizing the conjunctive scheduling paradigm. The salient feature of this scheduler is that it is entirely in middleware and makes no changes to the underlying native scheduler. By carefully controlling the task priority assignments, it creates an optimal schedule for task executions. When the node starts, its peripheral scheduler reads the necessary startup information(scheduling policy, priority levels) from the directory service and incorporates them in its functionality. The local scheduler also periodically updates the node specific state information in the directory service. It performs admission control tests to check whether a new task can be admitted and conveys the result to the global scheduler. Once the global scheduler dispatches a task to the local scheduler, it simply has to execute the task. The monitoring module check tasks for overrun and might decide to terminate or alter task characteristics decided by some policy. The PMS handles real-time tasks separately from the non real-time tasks. The time-sharing or the non real-time tasks require no special attention as they are implicitly slotted by the native scheduler when

the real-time tasks are not executing. As no changes need to be made to the native operating system, system heterogeneity of the nodes is not of concern. Moreover, the use of an interpreted language like java resolves the issues relating to mapping of native priorities to some intermediate priority that can be used by the middleware framework.

4.3 Addressing Soft Real-Time Tasks

The peripheral scheduler comprises of a soft realtime sub-scheduler(ASRT: Adaptive Soft Realtime Scheduler) that attempts to provide soft real-time guarantees to such tasks, while still ensuring that the normal time-sharing tasks within the system do not starve. The native operating system task priorities are carefully exploited to enforce the correct order in which the real-time tasks need to be executed. Moreover, a schedule is calculated so that each task gets adequate resources to finish execution within its specified deadline. Both the Earliest Deadline First (EDF) and the Rate Monotonic Scheduling (RMS) algorithms are supported and the adaptation module provides for dynamic switching between the above scheduling algorithms based on certain predetermined constraints. The real-time sub scheduler always runs at the highest possible fixed priority and only one real-time task runs at the second highest fixed priority at any time. The scheduler periodically wakes up and dispatches the next the task on the schedule. If there is no real-time task waiting, the scheduler suspends itself letting the native scheduler dispatch one of the timesharing tasks. If there are some real-time tasks waiting to be scheduled, the PMS changes the current active real-time task, from the running state to waiting state, and then dispatches another real-time task from the waiting pool based on the pre-calculated schedule.

5 Implementation Issues

A detailed discussion of the design and implementation of the *Compose*|*Q* runtime and the scheduling model is outside the scope of this paper. So we simply present the outline of the details. We first present a brief overview of the *Compose*|*Q* runtime architecture [23] followed by a discussion of some of the implementation issues encountered while incorporating our scheduler into the *Compose* runtime. The entire runtime is implemented using JDK-1.3 and the LDAP directory service interface.

5.1 The Compose|Q Architecture

The *Compose*|*Q* reflective framework uses Actors [1], a distributed computing paradigm that uses a model of concurrent active objects that communicate via asynchronous messaging. The *Compose*|*Q* includes

1 Active Objects called Actors. Every actor consists of (a) a globally unique (across time and nodes) ActorId, (b) a mail queue to store incoming messages, (c) a thread to process those messages, (d) the ability to create new actors and (e) the ability to send messages to other actors.

2 A NodeManager that manages and co-ordinates various components on a node. These tasks include: (a) Creating and destroying actors, (b) Starting up and shutting down the various run-time modules, and (c) Communicating with the distributed middleware components.

3 A NodeInfoManager that manages information needed by the local actors and interfaces with the directory service.

4 A communication sub-system that handles messaging between actors.

To start *Compose*|*Q* on a node, the NodeManager has to be started first. When a new actor is created it registers itself with the NodeManager. The NodeManager enters the new actor into a local-table which helps keep track of the actor for activities such as node checkpointing and node shutdown. The NodeManager also initiates the various other modules such as the NodeInfoManager and communication components such as the Router, Postman, and RemoteMessageReciever and the adaptive scheduler. The NodeInfoManager is a repository of information as well as an interface to the main directory service in the distributed architecture.

The message transport layer provides a framework for sending the outgoing messages to the appropriate node (routing) and resolving incoming messages to their appropriate actor queues (resolution). The communication transport layer consists of the following components (implemented as threads in the runtime system): a Router, a Postman and a RemoteMessageReceiver (See Figure 2). The transport layer maintains two message queues on a node for all incoming and outgoing messages (on that node) called SendPot and ReceivePot respectively. When an actor on a node sends a message, the message is put into the node's SendPot. The Router picks up messages from the SendPot, extracts the target of the message, and then consults the NodeInfoManager to obtain the current location (node) of the target actor. If the location of the target actor is local (i.e. on the same node), the Router puts the message directly into the node's ReceivePot. If the target actor is remote, the Router sends the message to the remote node.

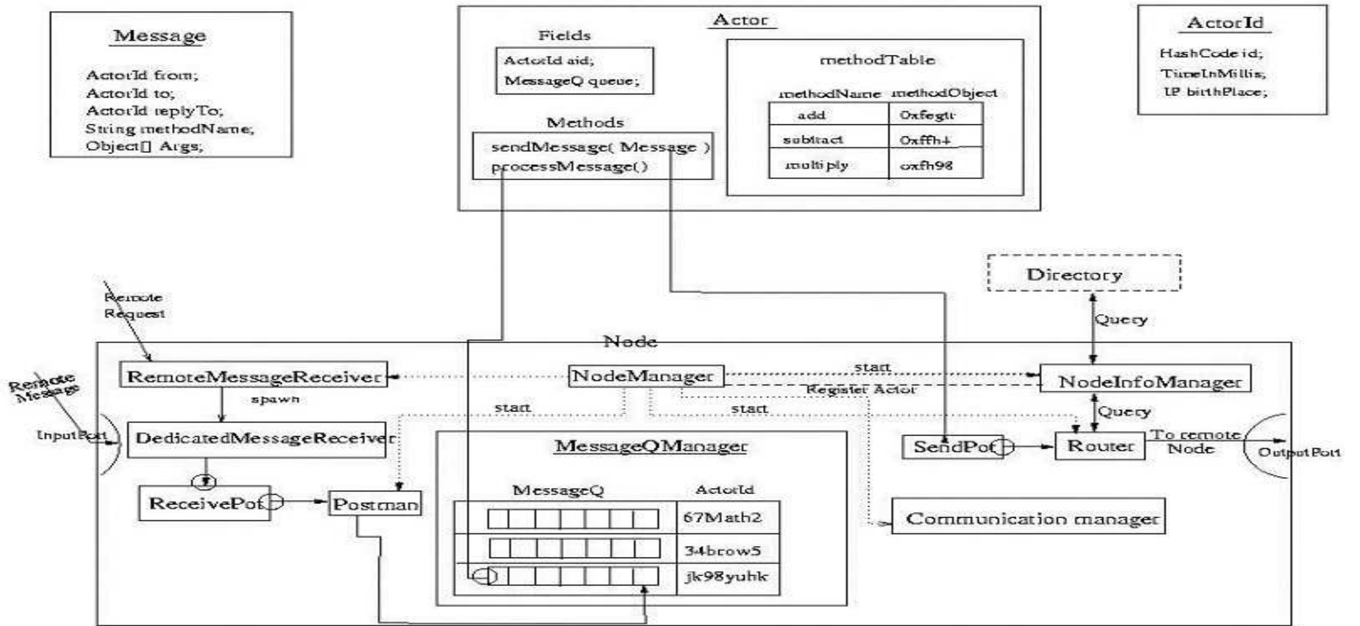

Figure 2: The *Compose|Q* Runtime

The RemoteMessageReceiver (RMR) on the target node handles incoming messages. It extracts the message and puts it into the node's ReceivePot. The Postman then picks up the message and adds it to the target actor's message queue. Messages are currently sent using TCP sockets and as an optimization the Router maintains a cache of open-connections that it reuses while sending out messages.

5.2 Extending the *Compose|Q* Runtime

The peripheral meta scheduler module runs on each individual node in the distributed system and it schedules locally originating tasks along with the tasks assigned to it by the focal scheduler. This scheduler performs the conjunctive message and task scheduling based on the system wide meta-rules established by the focal scheduler. Each peripheral node is aware of the different message and task priority levels and can therefore generate a unique priority ordering based on the independent task and message priorities. Any global changes made by the focal scheduler are reflected in the directory service and the peripheral schedulers periodically update their local information from the directory service. Therefore, there might be some delay before the global policies are reflected in all the distributed nodes. Fig.3 shows the changes required to the node runtime to achieve conjunctive scheduling. The rules mentioned in section 2 are used to determine the final task priorities.

At this point, it is important to note that the PMS performs scheduling based on the tasks running on its node and the messages inbound to them. In order to ascertain the individual message priorities, the scheduler needs to peek into each message. Within *Compose|Q*, the Postman routes all messages to their destination queues and is the only entity that touches each message. We therefore introduce a small scheduling agent into the Postman that reads the individual message priority. The individual actor message queues are also modified to support priority messages. i.e. messages can be inserted into the message queues based on their priorities (no longer a FIFO queue). The runtime can now be modified as follows to provide for conjunctive scheduling.

- The scheduler agent (SA) peeks at the message, and reads the message priority level (L_i) and the destination actor ID (aid). The actor ID uniquely identifies the actor, and therefore the priority group $Base_i$ of the actor.

- If ($Base_i$+$Boost_i$)is greater than the destination Actor priority, the scheduling agent boosts the priority of Actor to the new value $Base_i$+$Boost_i$ and the scheduler agent registers the ID with the PMS. At some later point, when the actor receives a low

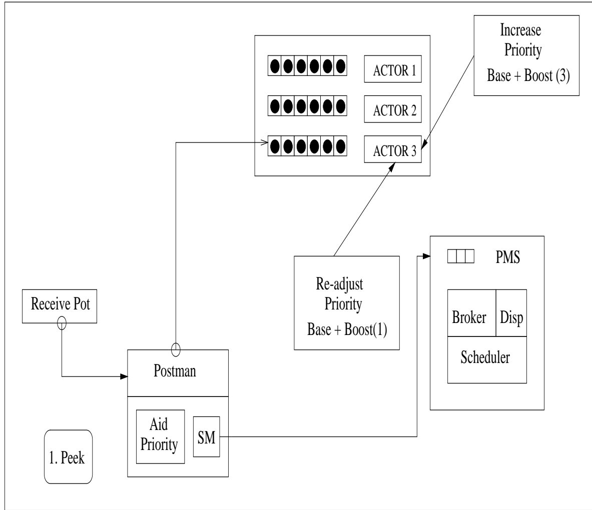

Figure 3: Conjunctive Scheduling in $Compose|Q$

priority message its aggregate priority needs to be reduced.

- The agent then directs the Postman to place the message in the appropriate place in the actor queue. A high priority message is positioned ahead of all messages with a lower priority level than it.

- The scheduling of the real-time actors is carried out as before and the real-time actors are not affected by other actors at lower priority levels.

- For the non real-time actors, the node runtime re-adjusts the priorities of all the actors that had their priorities boosted. The priority is adjusted to Base + Boost(for the highest priority msg in the queue). With these changes, the highest priority message to any node always gets preference over the low priority messages. The problem of starving is handled by employing an aging policy where low priority messages get an extra priority slice when they have been in the queue for a predetermined time.

Some of the issues that are hard to resolve are fixing of the system wide invariants. For example, what would be an ideal scheduling interval for the real-time sub-scheduler? Which runtime component needs to control the priority assignments to messages? Arriving a priority distribution for the meta actors is a concern, especially if there are a few priority levels supported. In our implementation we are limited by the ten priority levels provided by java. A more flexible mapping would enable more fine grained customizations.

6 Related Work

A tremendous amount of research has already been done in the domain of providing real-time capabilities to general purpose operating systems. Most of the solutions follow closely or can be approximated to the request, reserve and grant resources approach. Though most of the proposed solutions are dissimilar to our approach in the fact that they involve modification of the native OS schedulers or some system level component, nevertheless, the results and issues provide valuable insight into the design and implementation of our framework. Extensible operating systems extend kernel functionality to improve scheduling characteristics of the operating systems [6, 11]. Some of the other systems are Choices, Spin, Vino, x-kernel, Synthesis, Flux, Scout, Space and the Spring systems. The second approach enhances existing schedulers within general purpose operating systems to improve scheduling [25]. Examples are the Hierarchical CPU scheduler [9], CPU inheritance scheduling [7],the Processor Capacity Reserve [17],SMART [20],Nemesis [15], the Constant Utilization Server [5], the User-level Realtime Scheduler [13]and the Real Time Upcall [8].
Finally, the middleware advances try to guarantee upgraded and refined scheduling by exploiting system resources optimally and getting the maximum out of the system. We now review briefly the approach that has contributed and has a major bearing on our system design and implementation. The Soft Real Time Scheduling Server [3] is a middleware solution that provides soft real-time guarantees and provides for the co-existence of both real-time and time-sharing tasks. It provides for CPU overrun and adaptation. The DSRT is a generalized approach for scheduling both realtime and non realtime tasks on a general purpose OS, but it has no provision for combining task and message priorities. Our method uses a similar approach as DSRT for scheduling real-time applications, but is significantly different from DSRT as we use a aggregated task priority assignment scheme, that takes into account both task and message priorities. Moreover, we use a two-tier scheduling architecture to provide the coarse and fine grained schedulability. [4] describes the meta-programming techniques applied in Juno, an extension for realtime CORBA for scheduling policy translations using ordering of priority equivalence classes.

7 Future Work & Conclusion

In this paper, we proposed a middleware scheduling scheme for scheduling tasks on a CPU in distributed message oriented environments that weighs both task

and message priorities. Rules for arriving at the conjunctive priorities were presented and details of integrating this model into an existing message based framework were discussed. Our model is distinctive in the sense that it presents an absolute notion of priority in an environment where simply considering tasks would be inadequate. We are in the process of incorporating all the features of our scheduler into the $Compose|Q$ framework. In addition we have identified a number of extensions to the current scheduling architecture. We need to identify and implement more optimal algorithms for the focal scheduler for effective load balancing. Moreover, distributing the focal scheduler would boot out the single point of failure in our approach. Currently, we have not considered in detail the consequences of node failures and the various approaches to recover from failures. We understand that the directory service can be effectively used for recovery but we need to explore this issue in more depth.
In general, the dynamic nature of applications and environments imply that middleware mechanisms must be dynamic, adaptive and customizable. Such middleware would provide the foundations of large distributed systems.

References

[1] G. Agha *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass., 1986.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* In Proc. of the 13th ACM Symposium on Operating Systems Principles, pages 95-109, October 1991.

[3] Hao-Hua Chu. *A Soft Real Time Scheduling Server in UNIX Operating System*, Report No. UIUCDCS-R-97-1990, UILU-ENG97-1704, UIUC, Jan 1997.

[4] Angelo Corsaro, Douglas C. Schmidt, Chris Gill, and Ron Cytron, *Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Policies in Open Distributed Real-Time Systems*, Proceedings of the 3rd International Symposium on Distributed Objects and Applications, September 8-10, 2001, Rome, Italy.

[5] Z. Deng, J.W.-S Liu, J. Sun. *Dynamic Scheduling of Hard Real-Time Applications in Operating System Environment.* Technical Report No. UIUCDCS-R-96-1981, UIUC, Oct. 1996.

[6] Dawson R. Engler, M. Frans Kaashoek, and James OToole Jr. *Exokernel: An Operating System Architecture for Application-Level Resource Management.* In Proc. of the 15th ACM Symposium on Operating Systems Principles, December 1995.

[7] Bryan Ford and Sai Susarla. *CPU Inheritance Scheduling.* In Proc. of the 2nd Symposium on Operating Systems Design and Implementation, pages 91-105, Seattle, WA, October 1996.

[8] R. Gopalakrishnan. *Efficient Quality of Service Support Within Endsystems for High Speed Multimedia Networking.* PhD Thesis, Washington University. Dec. 1996.

[9] Pawan Goyal, Xingang Guo, and Harrick Vin. *A Hierarchical CPU Scheduler for Multimedia Operating System.* The proceedings of Second Usenix Symposium on Operating System Design and Implementation.

[10] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barrera, III. *Modular Real-Time Resource Management in the Rialto Operating System.* In Proc. of the 5th Workshop on Hot Topics in Operating Systems, May 1995.

[11] [KYO96] Jun Kamada, Masanobu Yuhara, Etsuo Ono. *User-level Realtime Scheduler Exploting Kernel-level Fixed Priority Scheduler.* Multimedia Japan, March 1996.

[12] Ian Leslie, Derek McAuley et.al. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE Journal on Selected Areas in Communications, 14(7):1280-1297, September 1996.

[13] C. L. Liu and J. W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.* JACM, 20(1): 46-61, 1973.

[14] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. *Processor Capacity Reserves: Operating System Support for Multimedia Applications.* IEEE International Conference on Multimedia Computing and Systems. May 1994.

[15] Klara Nahrstedt, Hao-hau Chu. *QoS-Aware Resource Management for Distributed Multimedia Application.* Technical Report No. UIUCDCS-R-97-2030, UIUC, Oct. 1996.

[16] Jason Nieh, Monica Lam. *The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications.* Proceedings of the Sixteenth ACM Symposium on operating Systems Principles, St. Malo, France, Oct. 1997.

[17] Object Management Group, *Real Time CORBA*, OMG Document No: orbos/99-02-12.pdf, 1999.

[18] Nalini Venkatasubramanian, Mayur Deshpande, Shivjit Mohapatra, Jehan Wickramasuriya et. al., *Design & Implementation of a Composable Reflective Middleware Framework*, ICDCS-21, April 2001.

[19] David K. Y. Yau and Simon S. Lam. *Adaptive Rate-Controlled Scheduling for Multimedia Applications.* ACM Multimedia Conference, Boston, MA, Nov 1996.