Scheduling and Priority Mapping For Static Real-Time Middleware

LISA CINGISER DIPIPPO, VICTOR FAY WOLFE, LEVON ESIBOV, GREGORY COOPER AND RAMACHANDRA BETHMANGALKAR

lastname@cs.uri.edu

Department of Computer Science, University of Rhode Island, Kingston, RI 02881

RUSSELL JOHNSTON

russ@nosc.mil

NCCOS, RDT&E DIV (NRaD), San Diego, CA 92152

BHAVANI THURAISINGHAM AND JOHN MAUER

thura@mitre.org, johnm@mitre.org

MITRE Corporation, Bedford MA

Editor: Wei Zhao

Abstract. This paper presents a middleware real-time scheduling technique for static, distributed, real-time applications. The technique uses global deadline monotonic priority assignment to clients and the Distributed Priority Ceiling protocol to provide concurrency control and priorities for server execution. The paper presents a new algorithm for mapping the potentially large number of unique global priorities required by this scheduling technique to the restricted set of priorities provided by commercial real-time operating systems. This algorithm is called Lowest Overlap First Priority Mapping; we prove that it is optimal among direct priority mapping algorithms. This paper also presents the implementation of these real-time middleware scheduling techniques in a Scheduling Service that meets the interface proposed for such a service in the Real-Time CORBA 1.0 standard. Our prototype Scheduling Service is integrated with the commercial PERTS tool that provides schedulability analysis and automated generation of global and local priorities for clients and servers.

Keywords: real-time, CORBA, distributed, static, priority mapping, scheduling

1. Introduction

The advent of *middleware,* software such as the Object Management Group's (OMG) CORBA (OMG 1996), that enables communication and coordination among clients and servers in a distributed system, has provided the ideal vehicle to enforce real-time scheduling policies across a distributed system. Researchers have developed powerful real-time scheduling techniques for distributed systems (Liu 1999), but these techniques have not previously made their way into commercial products or mainstream applications. A major obstacle to the employment of these real-time distributed scheduling techniques in mainstream applications has been that the nodes in most real-world distributed systems use autonomous, often heterogeneous, commercial real-time operating systems. These distributed systems have lacked the ability to coordinate among the heterogeneous nodes. Such coordination is necessary to effectively implement distributed scheduling policies. Middleware can provide these distributed coordination capabilities.

We have developed a model, design, and middleware implementation of a distributed deadline monotonic priority assignment technique along with a distributed priority ceiling

resource access protocol for fixed priority static distributed systems. Most distributed scheduling techniques, including our technique, assume unique priorities for the tasks in the system. However, in actual systems, the enforcement of these priorities is typically done by local real-time operating systems that have a restricted set of available priorities. For instance, the POSIX1c standard (IEEE 1990) mandates a minimum of 32 priorities, and the VxWorks (WindRiver) and LynxOS (Lynx) real-time operating systems each provides 256 priorities. Thus, implementing these conceptual distributed scheduling techniques introduces *the priority mapping problem.* The issue here is how to map a potentially large number of global unique priorities required by the distributed scheduling techniques to the restricted number of local priorities provided by the real-time operating systems, and how to do associated schedulability analysis. For instance, the schedulability analysis must account for blocking of higher global priority tasks by lower global priority tasks that is caused by the higher and lower global priority tasks being mapped to the same local priority, and thus being scheduled first-in-first-out (FIFO) by POSIX1c-based real-time operating systems. Some work on priority mapping has been done (Lehoczky 1986, Katcher 1995), and we review that work in Section 2. However, a priority mapping solution for enforcing global priorities in middleware that must interact with local real-time operating systems, has not been developed.

In this paper we present a priority mapping algorithm for real-time middleware and prove that is optimal within a particular class of algorithms.

We also describe our implementation of the distributed deadline monotonic + distributed priority ceiling + priority mapping middleware scheduling technique using the OMG's Real-Time CORBA 1.0 (RT CORBA) Scheduling Service standard interface (OMG2 1998). This implementation is integrated with the PERTS scheduling tool from Tri-Pacific Software (TriPacific) that was originally developed at the University of Illinois (Liu 1993). We have augmented PERTS with the priority mapping algorithm described in this paper so that PERTS produces a schedulability analysis, using rate-monotonic analysis techniques, an assignment of unique priorities across the distributed system, and the mapping to the actual priorities that should be used on the local operating systems. The PERTS schedulability analysis accounts for the additional potential blocking introduced by the priority mapping. We have implemented the RT CORBA Scheduling Service to automatically retrieve the local priorities specified by PERTS and use them to assign the mapping required by the RT CORBA standard, and to assign priorities for client and server threads in the distributed system.

Section 2 provides general background on the CORBA and RT CORBA standards, real-time middleware, and distributed scheduling techniques. Section 3 describes our model and our distributed deadline monotonic with distributed priority ceiling scheduling technique. Section 4 provides detail on our priority mapping algorithm for real-time middleware and proves its optimality under certain conditions. Section 5 describes our implementation of the RT CORBA Scheduling Service and associated PERTS real-time analysis tool that work together to perform distributed deadline monotonic scheduling with distributed priority ceiling and priority mapping in RT CORBA middleware. Section 6 summarizes.

2. Background

In this section we describe previous work that has been done in middleware scheduling. We start off with a review of the CORBA standard. We then discuss the recent RT CORBA draft specification. We briefly describe several real-time middleware architectures and implementations, including some academic research in real-time

CORBA. We then go on to discuss approaches that have been taken in the past towards scheduling in distributed systems. We describe several general algorithms that have been developed for distributed real-time scheduling. We also describe how current middleware implementations support real-time scheduling.

2.1. CORBA

The CORBA standard specifies interfaces that allow seamless interoperability among clients and servers under the object-oriented paradigm. The standard is produced by the Object Management Group. CORBA version 1.1 was released in 1992, version 1.2 in 1993, and version 2.0 in 1996. The V1.2 standard deals primarily with the basic framework for applications to access objects in a distributed environment. This framework includes an object interface specification and the enabling of remote method calls from a client to a server object. Issues such as Object Request Broker (ORB) inter-operability, naming, events, relationships, transactions, and concurrency control are addressed in version 2.0 of the CORBA standard (OMG 1996). Most of these are addressed with *Common Object Services*, which are modules whose interface and semantics are specified for a particular common function, like determining the object reference to a remote object using a name (*Naming Service)* or providing well-defined concurrency control for servers (*Concurrency Control Service).*

The OMG is still growing in participation and in the scope of CORBA's capabilities. Vendors are producing software that meets the standards very soon after each revision to the standard comes out. New extensions are coming out of every meeting and many more are on the OMG's roadmap. A complete list of OMG working groups, their whitepapers, current standards, and draft standards, can be found on the OMG Web site at http://www.omg.org . Included is the work of the Real-Time Special Interest Group, which has produced the Real-Time CORBA 1.0 standard (OMG2 1998) and is continuing to develop it.

2.2. Real-Time CORBA

The OMG's Real-Time CORBA 1.0 standard is designed for *fixed priority* real-time operation. Dynamic scheduling, where priorities can vary during execution is being addressed in RT CORBA 2.0, which at the time of this writing, is being developed.

The RT CORBA standard specifies real-time support in the ORB and in a *Scheduling Service.* The RT CORBA standard assumes priority-based scheduling capabilities in the operating systems on nodes in the system and does not assume any real-time capabilities in the network. That is, the RT CORBA specification only addresses real-time issues in the scope of the CORBA software. While support in the CORBA software is necessary for a complete distributed real-time solution, it is not sufficient because all parts of the system must be designed for real-time to get sufficient real-time support.

2.2.1. *Real-Time ORB.* RT CORBA defines a *thread* as its schedulable entity. The RT CORBA notion of thread is consistent with the POSIX definition of threads (IEEE 1990). At any instant in time, a thread has two priorities associated with it: a *CORBA Priority* and a *Native Priority*. The CORBA Priority of a thread comes from a universal node-independent priority ordering of threads used throughout the CORBA system. It corresponds to the notion of *global priority* that we use in this paper. The Native Priority of a thread is the priority used by the underlying system; it corresponds to the notion of *local priority* that we use in this paper. RT CORBA specifies *Priority Mapping* to map the CORBA Priority of thread to the Native Priority it needs to execute on the underlying

system. Real-time ORBs provide a default priority mapping algorithm, but RT CORBA also specifies a `install_priority_mapping` method to allow application code, or the RT CORBA Scheduling Service to install its own mapping. The lowest overlap first priority mapping algorithm described in this paper is ideally suited for this priority mapping functionality.

RT CORBA Clients express CORBA Priority in their threads by creating a local object called a `CORBA::Current`. When created using the RT CORBA interface, the Current object contains, in addition to other attributes, a *Priority* attribute, which will hold the CORBA Priority of the thread that created the Current object. A thread sets its CORBA Priority by writing the *Priority* attribute of its Current object. The RT ORB will map this CORBA Priority to the Native Priority on the local real-time operating system to execute the thread. The RT ORB also has access to the CORBA Priority in the Current object so that it can do things such as propagate the CORBA Priority to any CORBA servers that the thread calls.

CORBA is designed to allow servers to be written independently of the clients that will access them. A CORBA *server* can be thought of as a process in which reside the CORBA objects that clients use. That is, a CORBA object provides methods that perform the service for the client; the server is the process that contains the CORBA objects and assigns threads to invoke the methods of the CORBA objects on the client's behalf. When the server creates/assigns a thread to invoke a method on a CORBA server on the client's behalf, the server is said to *dispatch* a thread. RT CORBA provides primitives to control the dispatching of server threads; it does not provide primitives to use in the application code of CORBA objects themselves.

A CORBA server process uses one or more objects called *Portable Object Adapters* (POAs) (OMG1 1998) to manage the creation/deletion of CORBA objects and the dispatch of threads. A CORBA POA object is constructed with a set of *policies* specified in the CORBA standard. These policies specify how the POA creates/deletes objects and dispatches threads for the objects that it manages. The RT CORBA specification (OMG2 1998) defines POA policies that are specific to real-time systems. One such real-time policy establishes *thread pools* at some original priority to receive requests from clients. The *Server Priority* POA policy specifies what priority the thread will execute at after it has been dispatched. There are two models currently specified in RT CORBA: "Client Priority Propagation", where the server thread executes at the CORBA Priority of the client that requested it; and "Server-Set Priority", where the server thread executes at a priority set as a parameter to the policy. A *priority Transform* policy allows the client's priority to be offset with a base priority, as is required in real-time Distributed Priority Ceiling Protocols (see Section 2.3.2).

2.2.2. Fixed Priority Scheduling Service. RT CORBA also specifies a Scheduling Service that uses the RT CORBA primitives to facilitate enforcing various fixed-priority real-time scheduling policies across the RT CORBA system. The Scheduling Service abstracts away from the application some of the complication of using low-level RT CORBA constructs, such as the POA policies. For applications to ensure that their execution is scheduled according to a uniform policy, such as global Rate Monotonic Scheduling, RT ORB primitives must be used properly and their parameters must be set properly in all parts of the RT CORBA system. A Scheduling Service implementation will choose CORBA Priorities, POA policies, and priority mappings in such a way as to realize a uniform real-time scheduling policy. Different implementations of the Scheduling Service can provide different real-time scheduling policies.

The Scheduling Service uses "names" (strings) to provide abstraction of scheduling parameters (such as CORBA Priorities). The application code uses these names to specify CORBA Activities and CORBA objects. The Scheduling Service internally associates these names with actual scheduling parameters and policies. This abstraction improves portability with regard to real-time features, eases use of the real-time features, and reduces the chance for errors.

The Scheduling Service provides a `schedule_activity` method that accepts a name and then internally looks up a pre-configured CORBA Priority for that name. The Scheduling Service also provides a `create_POA` method to create a POA and set the POA's RT CORBA thread pool, server priority, and communication policies to support the uniform scheduling policy that the Scheduling Service is enforcing. For instance, if the Scheduling Service were enforcing a scheduling policy with priority ceiling semantics, it might create thread pools with priority lanes at the priority ceiling of the objects it manages to ensure that threads start at a high enough priority before dispatch. The Scheduling Service provides a third method,

```
0       install_priority_mapping(. . .);

Client
C1    sched = create scheduling service object;
C2    obj = bind to server object
C3    sched->schedule_activity ("activity1");
C4    obj->method1( params );   // invoke the object
C5    sched->schedule_activity ("activity2");
C6    obj->method2(params );

Server Main
S1    sched = create scheduling service object;
S3    poa1 = sched->create_POA(. . .);
S4    obj = poa1->creat_object ( params );   // create object
S5    sched->schedule_object(obj, "Object1" );
      ...
```

Figure 1 - Example of RT CORBA Static Scheduling Service

called `schedule_object`, that accepts a name for the object and internally looks up scheduling parameters for that object. For instance, it could set its priority ceiling so that it can do a priority ceiling check at dispatch time.

The example in Figure 1 illustrates how the Scheduling Service could be used, and also indicates some of the issues in creating RT CORBA clients and servers. Assume that a CORBA object has two methods: `method1` and `method2`. A client wishes to call `method1` under one deadline and `method2` under a different deadline.

In Step 0, the Scheduling Service installs a priority mapping that is consistent with the policy enforced by the Scheduling Service implementation. For instance, a priority mapping for an analyzable Deadline Monotonic policy might be different than the priority mapping for an analyzable Rate Monotonic policy.

The `schedule_activity` calls on lines `C3` and `C5` specify names for CORBA Activities. The Scheduling Service internally associates these names with their respective CORBA Priorities. These priorities are specified when the Scheduling Service is instantiated at system startup. For instance, our implementation described in Section 5 specifies deadline monotonic priorities through a configuration file.

The server in the example has two Scheduling Service calls. The call to `create_POA` allows the application programmer to set the non-real-time policies, and internally sets the real-time policies to enforce the scheduling algorithm of the Scheduling Service. The resulting POA is used in line `S4` to create the object. The second Scheduling Service call in the server is the `schedule_object` call in line `S5`. This call allows the Scheduling Service to associate a name with the object. Any RT scheduling parameters for this object, such as the priority ceiling, are assumed to be internally associated with the object's name by the Scheduling Service implementation.

At the time of this writing, Real-Time CORBA 1.0 is being finalized by the OMG. Real-Time CORBA 2.0, which addresses dynamic scheduling, is expected to be developed in 1999 and 2000.

2.3. Real-Time Middleware

As defined in Section 1, middleware in a distributed system is software that enables communication between clients and servers on multiple platforms. Real-time middleware provides support for priority scheduling. In this section we describe some real-time middleware architectures, including several extensions of the CORBA standard that incorporate support for real-time applications.

2.3.1 Real-Time Middleware Research. At The University of Michigan, the ARMADA Middleware Suite (Abdelzaher 1997) was developed as a modular collection of middleware services on OSF MACH-RT, a standard real-time operating system. ARMADA provides services for managing computation and communication resources, for providing access to shared and/or replicated data in a networked environment, and for managing system dependability.

The MidART (Midleware and network Architecture for distributed Real-Time systems) project (Gonzalez 1997) being developed at The University of Massachusetts Amherst provides a set of real-time application specific, but network transparent programming abstractions for supporting individual application data monitoring and control requirements. It provides Real-Time Channel-Based Reflective Memory (RT-CRM), an association between a writer's memory and a reader's memory on two different nodes, and Selective Channels that allow applications to dynamically choose the remote node(s) from which data is to be viewed and to which it is to be sent at run-time.

2.3.2 Real-Time CORBA Research. In this section we describe some research projects that have provided the foundation on which much of the RT CORBA standard is based.

Early work at MITRE (Krupp 1994, Bensley 1996) prototyped a Real-Time CORBA system by porting the ILU ORB from Xerox to the Lynx real-time operating system. This system provided a static distributed scheduling service supporting rate-monotonic and deadline-monotonic techniques.

The ACE ORB (TAO)(Schmidt 1997), developed at The University of Washington in St. Louis, is a high-performance endsystem architecture for real-time CORBA. It provides support for specification and enforcement of quality of service (QoS), as well as a real-time scheduling service that relies on an off-line, rate-monotonic scheduler to

guarantee timing constraints. Current work at the University of Illinois Urbana-Champaign extends TAO to allow for on-line schedulability testing using admissions tests on dynamic tasks (Feng 1997).

Previous work at the University of Rhode Island has developed a dynamic real-time CORBA system (DiPippo 1999) that provides expression and best-effort end-to-end enforcement of soft real-time client method requests. The system provides end-to-end enforcement of the specified timing constraints through extensions to CORBA's object services. It provides a Global Priority Service for assigning and maintaining global priorities, a real-time Event Service for prioritizing delivery of events, and a real-time Concurrency Control Service that provides priority inheritance for requests that are queued on a server.

2.4. Distributed Scheduling

Scheduling tasks in a distributed system involves assigning priorities to the tasks and controlling access to shared resources. In general, two classes of protocols have been developed for scheduling distributed tasks (Liu 1999). The first class of distributed scheduling protocols is known as End-to-End Scheduling Protocols. We describe some of the protocols in this class below. The Distributed Priority Ceiling Protocol (DPCP) (Rajkumar 1991) makes up the other class of protocols. Much of our work is based on the DPCP, and so in this section we describe the protocol in some detail.

2.4.1. End-To-End Scheduling Protocols. The model on which this class of protocols is based does not allow nesting of requests for resources on different processors. Thus, each periodic task that requires resources on more than one processor can be thought of as an end-to-end periodic task. There are two parts of any end-to-end scheduling protocol: 1) synchronization of the execution of sibling subtasks on different processors and 2) scheduling subtasks on each processor. Because no subtask requires resources on more than one processor, there is no need for schedulers on different processors to use a common resource access control protocol.

Two approaches have been developed for synchronizing the execution of end-to-end tasks: greedy and non-greedy approaches (Liu 1999). In greedy synchronization, each scheduler releases each successor task as soon as the immediate predecessor job completes. Schedulers on the different nodes on which these tasks will execute pass synchronization signals among them. While this is a simple solution, the schedulability analysis for greedy synchronization is complex. Non-greedy synchronization protocols reshape the release-time patterns of later subtasks. The worst case end-to-end response time for the non-greedy synchronization protocols is much better than the greedy protocols.

A general approach to scheduling tasks with known start times, deadlines, and precedence relations among the tasks is presented in (Parnas 1993). This approach incorporates many of the characteristics needed in distributed middleware scheduling.

The schedulers on each processor can be completely independent of each other. Several heuristics have been developed to compute intermediate deadlines of the subtasks of an end-to-end task (Liu 1999). Once this is done, priority assignment of each individual task reduces to uni-processor task scheduling.

Another approach to mutli-node scheduling that takes into account resource requirements, but does not require periodic tasks, is presented in (Ramamritham 1989). Here all resources are allocated to tasks a priori in a reservation-based system that allows utilization calculations of each resource to check schedulability.

2.4.2. The Distributed Priority Ceiling Protocol. In a single node system, schedulability of hard real-time tasks that require resources can be computed using well-known analyses (Liu 1973, Lehoczky 1989) that take into account the timing and resource requirements of all tasks in the system. In a distributed system, this analysis is complicated by the fact that tasks may require resources that reside on other nodes than their own.

The Distributed Priority Ceiling Protocol (DPCP) (Rajkumar 1991) extends the priority ceiling protocol (PCP) (Rajkumar 1991) by taking into account accesses to remote resources. In the DPCP, a resource that is accessed by tasks allocated to different processors than its own is called a *global resource*. All other resources (those only accessed by local tasks) are *local resources*. A critical section on a global resource is referred to as a *global critical section* (GCS). A *local critical section* (LCS) refers to a critical section on a local resource. The base priority (BP) of a system of tasks is a fixed priority, strictly higher than the priority of the highest priority task in the system. We assume that higher numbers correspond to higher priorities. As in the single-node PCP, the priority ceiling of a local resource is the priority of the highest priority task that will ever access it. The priority ceiling of a global resource is the sum of the BP and the priority of the highest priority task that will ever access it. When a task executes a GCS, the task suspends itself on its local processor, and the GCS executes at a priority equal to the sum of the BP and the priority of the calling task on the host processor of the resource. Each processor in the system runs the PCP given the priorities and priority ceilings as described above.

The schedulability analysis of the DPCP is an extension of the schedulability analysis of the PCP. The only difference is that there are more forms of blocking due to access of remote resources. For instance, the DPCP analysis must take into account blocking that occurs when a task requests a global resource on another node, but must wait for a lower priority task that currently holds the resource.

2.4.3. Priority Mapping. The chosen priority assignment scheme along with the DPCP provide global priorities with which to schedule tasks. In a actual systems, these global priorities must be mapped to local priorities that can be handled by the operating system on which each task is executing. A further complication in the mapping of global to local priorities involves the fact that many current real-time operating systems provide a limited number of priorities. This implies that on such an operating system, it is possible that several tasks with different global priorities may have to be mapped to the same local priority, introducing the possibility of priority inversion. An implementation of a distributed scheduling algorithm should handle this priority mapping problem.

Several approaches have been suggested to handle this priority mapping problem. In (Lehoczky 1986), sufficient conditions were developed for schedulability of periodic tasks in a system with limited local priorities. The mapping algorithm presented in (Lehoczky 1986), assigns local priorities but does not take into account the execution time of the tasks involved. Rather, it assumes that all tasks have equal execution time. The work presented in (Katcher 1995) develops both necessary and sufficient conditions for determining schedulability on a single node system with limited priorities. A metric called the *degree of schedulable saturation* is introduced to evaluate the impact of task groupings on schedulability. By minimizing this metric, the best mapping can be found.

2.4.4. Middleware Distributed Scheduling. Middleware software provides the mechanisms for interaction between clients and servers in a distributed system. Therefore, in a real-time distributed system, an obvious place for a distributed scheduling

algorithm to be carried out is in the middleware. Here we briefly discuss how some of the middleware systems discussed in Section 2.1 handle real-time distributed scheduling.

The ARMADA middleware suite of software (Abdelzaher 1997) relies on the scheduling framework of underlying operating system, in the current design, the OSF MACH-RT. In the MidART system (Gonzalez 1997), a *Scheduler* module is responsible for scheduling active objects. The choice of scheduling algorithm is currently being considered. A *Global Connection Admission Control* (GCAC) module is responsible for performing a schedulability analysis of the QoS requirements before admitting a particular operation.

The TAO real-time CORBA system (Schmidt 1997) schedules tasks through the Scheduling Service by assigning rate-monotonic priorities, and performing off-line analysis of the system. In the extensions to the TAO system proposed by the work at UIUC (Feng 1997) a run-time Scheduling Service object, which is also referred to as a *scheduling broker*, performs schedulability analysis of all IDL operations that register with it.

The dynamic RT CORBA system developed at URI (DiPippo 1999) schedules client requests using earliest deadline first with importance, where importance takes precedence over deadline. Importance is an application specific property defined on. The Priority Service uses timing constraints and importance expressed by a client to compute a global priority that is used throughout the system to enforce the timing constraint. The MidART system and the TAO system, and its proposed extension, each perform schedulability analysis of tasks. However, none of the above middleware scheduling solutions considers the priority mapping problem discussed in Section 2.2. We have developed a formal model for scheduling fixed priority tasks through middleware, along with algorithms for implementing the model and for mapping unlimited global priorities to limited local priorities. We describe our solution in the next section.

3. DM+DPCP Middleware Scheduling

Here we present our fixed priority middleware scheduling solution, which is based on a global deadline monotonic priority assignment and a distributed priority ceiling resource access protocol. We first present a model on which the algorithms are based. We then describe our approach to fixed middleware scheduling, including our solution to the priority mapping problem discussed in Section 2.2

3.1. Model

3.1.1. Real-Time Distributed Fixed Priority Scheduling Model. We assume that the distributed system consists of one or more *processors* connected via a *network.* A *task* is a periodic, schedulable unit of execution. While this model does not explicitly handle non-periodic tasks, a periodic sporadic server (Liu 99) could be used to model non-periodic tasks. Each task is assigned a fixed priority based on its timing constraints using a chosen priority assignment scheme. With each task is associated a *local processor* on which the task executes. Each processor has a *local scheduler* with possibly a fixed number of priorities. The local scheduler schedules the tasks that execute on the associated processor.

A *resource* is any item in the system that can be shared usefully. Some resources require that access by tasks to the resource be guarded by a critical section to ensure that the resource state remains consistent. The processor on which a resource resides is called

its *synchronization processor*. A task in a distributed system may access resources throughout the system. A resource that is accessed by any task whose local processor is different from the resource's synchronization processor is known as a *global resource*. Any other resource is called a *local resource*. When a task accesses a resource, it executes a *critical section* on the resource's synchronization processor. The definitions for local critical section (LCS) and global critical section (GCS) and the priorities at which they execute are the same as in the DPCP as described in Section 2.2. When a task accesses a global resource, it suspends itself on its local processor during the execution of its GCS. A *global scheduler* for the entire system assigns priorities to tasks and critical sections regardless of which processor they will execute on.

3.1.2. Real-Time Middleware Model. The primitives just described allow us to model scheduling in real-time middleware. A middleware *client* is a program that performs some local processing, and also makes requests to servers in the system. A client may have multiple deadlines specified within a single period of its execution. The locations of the servers in the distributed system are transparent to the client. A middleware *server* is an entity that provides services to clients in the system. Our model is object-based and thus the services provided to clients are represented through methods on a server object. A client accesses the server by invoking these methods.

Figure 2 shows an example middleware application. There are three nodes in the system. Node 1 contains two clients. Node 2 contains two servers and one client. Node 3 contains a server. The servers are modeled as objects with methods for their well-defined interface, as in CORBA. We show detail for the code of one of the clients on Node 1. It is periodic with period *p1*. Within each period, this client first executes a block of code denoted *client code 1*. The client then binds to *server1* so that it may request a service from it. Similarly it binds to *server2*. The client then specifies an intermediate deadline of *d1* by which the code that follows it must complete. We assume that *d1* is a pre-period deadline within the client's period. Under deadline *d1*, the client invokes a method (requests service) on *server1* and then executes another block of local client code denoted *client code 2*. Next, the client specifies a second intermediate deadline *d2* under which a call to *server2* must be made. Finally, the client has a block of code denoted *client code 3* that must complete by the end of the period.

In order to analyze the schedulability of clients and servers that are described in the middleware model, they must be mapped to the schedulable entities of the lower-level real-time distributed scheduling model. A client with period p, final deadline d, and m intermediate deadlines d_1 to d_m maps to $m+1$ dependent tasks t_1 to t_{m+1} each with period p. Tasks t_1 to t_m have deadlines d_1 to d_m, and task t_{m+1} has deadline d. The tasks depend on each other in that for $i<j$, task t_i must complete before task t_j starts, for all $i,j=1..m+1$. Each server is modeled as a resource. A method invocation by a client on a server S maps to a critical section on the resource represented by S.

Figure 3 displays the same example that was illustrated in Figure 2 with the mapping explicitly expressed. Since the client has two intermediate deadlines, it maps to three tasks. The dependency graph displayed at the bottom of the figure indicates the required ordering on the tasks. The client's invocation of *server1 method1* will execute as a GCS on Node 2, the synchronization processor of *server1*

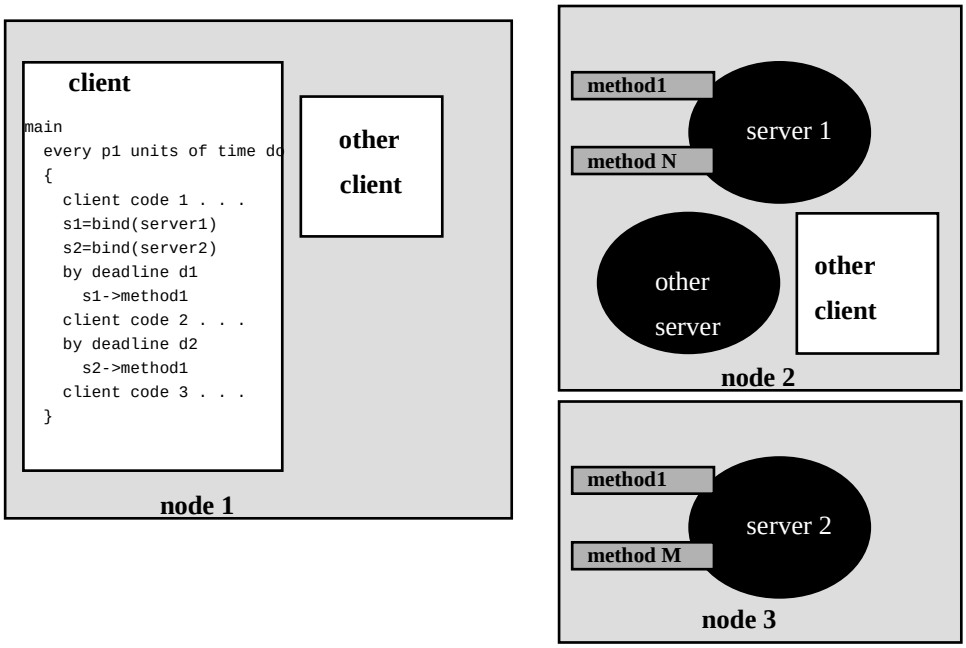

Figure 2 - Example Middleware Application

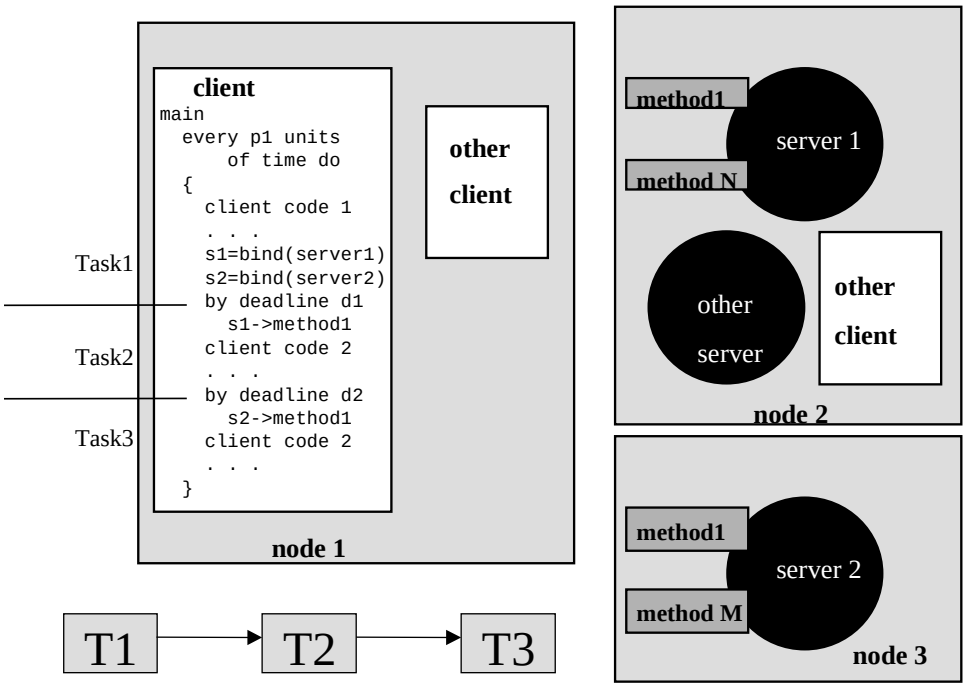

Figure 3 - Example Modeling Middleware Clients

3.2. DM+DPCP Scheduling Approach

While the above real-time distributed scheduling model does not assume any particular priority assignment scheme, we present here an approach to scheduling the elements of the model. The deadline monotonic (DM) priority assignment scheme assumes periodic tasks and statically assigns highest priority to tasks with the shortest deadline (Liu 1999). This technique works well with our model for several reasons. First, DM is a fixed priority assignment scheme, which is required by the model. Second, the periodic tasks in the model have deadlines that are possibly different from their periods, and so DM is a better choice than rate monotonic (RM), which only takes period into account. Third, the schedulability analysis of DM is well-known (Liu 1999), although not optimal in a distributed system (Sun 1997). In fact, it has been shown that the problem of scheduling any non-trivial system of tasks requiring ordered execution on more than two processors is NP-hard (Lenstra 1977), and that most forms of real-time multi-processing scheduling require heuristics (Stankovic 1995).

In our scheduling approach, we use the distributed priority ceiling protocol (DPCP) for resource access, such as the access of servers by clients.

3.3. Priority Mapping Problem

The theory behind the analysis of DM+DPCP assumes unique priorities assigned to tasks and GCS's. However, consider an example with 100 clients on a node, each with 2 intermediate deadlines, which map to 300 tasks, all invoking methods (GCSs) on other nodes. If the node was running VxWorks as its local real-time operating system, there would be only 256 local priorities with which to execute the 300 tasks. This is an instance of the priority mapping problem discussed in Sections 1 and 2.4.3. This problem can be defined in terms of the model primitives as follows. Suppose that in a distributed system, processor *n1* has *m* local tasks and *n* GCSs that will execute on it. The global scheduler will assign a unique priority to every task and GCS in the system. Thus, for processor *n1*, *m+n* unique priorities will be assigned. If the operating system running on processor *n1* has fewer than *m+n* local priorities, then some tasks and GCSs of different priorities will execute at the same local priority. This could cause priority inversion since most operating systems use FIFO scheduling within the same priority. This priority inversion is caused by the fact that a high priority task could be blocked by lower priority tasks ahead of it in the FIFO queue. This is a new form of blocking that must be taken into account when computing schedulability of the system.

To analyze the schedulability of a system in the described situation, we check how the limited priorities affect the time demand function introduced by the Lehoczky's schedulability criterion that is used in rate-monotonic analysis (Lehoczky 1989). We assume FIFO scheduling of tasks with the same local priority and make the worst case assumption that each task or GCS falls at the end of the FIFO queue for its priority. The time demand function should be modified as follows

$$W_i(t) = \sum_{\substack{j=All\ tasks\ of \\ higher\ priority}} C_j \lceil t / T_j \rceil + \sum_{\substack{k=All\ tasks\ of \\ the\ same\ priority}} (C_K * M_k) + C_i + B_i \le t$$

(Equation 1)

Here C_l represents an execution time of the task T_l, B_I is the blocking time of task T_I and M_k is a factor defined as

$$M_k = \min\{\lceil t / T_k \rceil, n_g + 1\}$$

where n_g is a number of remote GCSs executed by task T_l during a single period. The origin of this factor is that the task T_i may wait for the end of some same priority task's execution. It may wait once, when the task T_i is initialized, and every time it releases its CPU for an execution of a remote GCS, since a task of the same priority may get the CPU at that time period. At the same time it may not happen more often than the frequency of the same priority task T_k

$$\left\lceil t / T_k \right\rceil$$

Despite this obvious modification of the demand function, there is also a hidden modification of the blocking time B_i. This modification is due to the new feature of the global blocking

$$GB = n_g b'_g$$

where

$$b'_g = b_g + \sum_{\substack{k=All\_gcs's\_of \\ the\_same\_priorities}} CS_k$$

In the original time demand function, a GCS could be blocked for a duration of the longest lower priority GCS, b_g. In the new function, along with this blocking a GCS may also be blocked by the duration of all GCSs of the same priority.

In the next section we provide a solution to the priority mapping problem that maps global priorities to local priorities while attempting to maintain the schedulability of the distributed system.

4. Priority Mapping Solution

In this section we present a solution to the priority mapping problem. We begin by describing an algorithm that is optimal within a particular class of algorithms. We then describe a heuristic that is sub-optimal, but improves on the performance of the optimal solution. We provide a complexity analysis of the heuristic and describe an example to illustrate how the heuristic works.

4.1. Lowest Overlap First Priority Mapping Algorithm

Our algorithm for mapping global priorities to local priorities is based on the concept of overlapping multiple tasks and GCSs together into a single local priority. We start out with as many priorities as the global scheduler requires to schedule all tasks and GCSs. We then scan through the tasks in increasing global priority order, overlapping as many tasks and GCSs as possible without allowing the system to become non-schedulable. On each node, we overlap tasks and GCSs (map two or more tasks or GCSs to the same local priority) as many times as necessary to end up with the number of available local priorities.

Figure 4 displays a high-level flowchart of the mapping algorithm. The details of each part of the algorithm follow.

4.1.1 The Algorithm

Assign Global Priorities. The first step in the algorithm is to assign unique global priorities to all tasks and GCSs on all nodes according to the chosen priority assignment

mechanism (RM, DM, etc.) under the assumption that the number of available priorities is unlimited.

Perform Analysis. The algorithm next performs schedulability analysis on the tasks and GCSs using their global priorities. If the system is schedulable, it goes on to the next step. If the system is not schedulable, then the algorithm quits the mapping since no mapping will ever improve the schedulability of a system

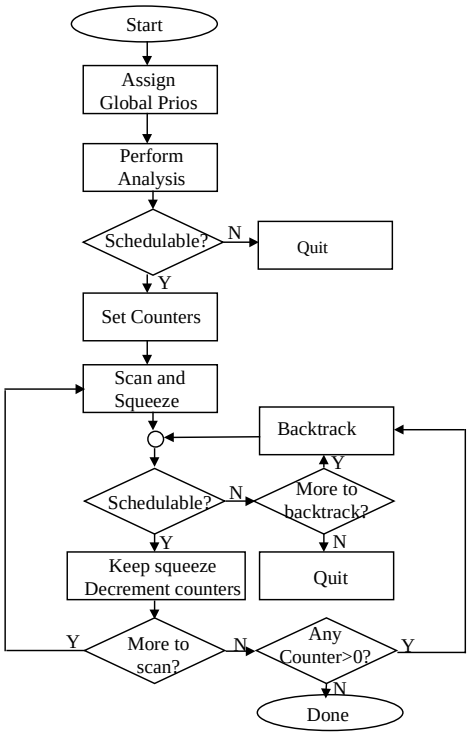

Figure 4 - Lowest Overlap First Priority Mapping Algorithm Flowchart

.

Set Counters. For every node, a counter is stored that represents the difference between the number of global priorities used on the node and the number of local priorities available. This counter can also be thought of as representing the number of priority overlaps required on the node. Thus, initially we have COUNT=#TASK+#GCS-#LOCAL, where COUNT is the counter, #TASK is the number of tasks on the node, #GCS is the number of GCSs on the node, and #LOCAL is the number of local available priorities on the node.

Scan and Overlap. This is the heart of the priority mapping algorithm. The goal here is to assign tasks and GCSs to local priorities, overlapping as many as necessary into the highest non-empty local priority without making the system unschedulable. If, on any node, the counter becomes non-positive, then no priority overlaps are necessary, and so tasks and GCSs on that node are assigned to the next available local priority. The algorithm scans the tasks and GCSs in increasing global priority order, regardless of which nodes they reside on.

During the mapping, tasks and GCSs have separate sets of local priorities into which they will be mapped. We will refer to these sets as *local task priorities* and *local GCS*

priorities. This is done because, under DPCP, the GCS priorities must be higher than the task priorities. After the mapping is complete, this distinction goes away, and we are left with at most the number of local priorities available on the node.

When a task is chosen during the scan, if its node has a non-positive counter, it is assigned to the next empty local task priority. Otherwise, if the chosen task is the first (lowest global priority) task on its node, it is assigned to the lowest local task priority. If the chosen task is not the first on its node, it is assigned to the highest non-empty local task priority, causing an overlap.

When a task is assigned a local task priority, each of its GCSs must also be assigned to some local GCS priority on its own node. If the counter on a GCS's node is non-positive, it is assigned to the next empty local GCS priority. Otherwise, if the GCS is the first scanned GCS on the node, it is assigned to the lowest local GCS priority. If the GCS is not the first on its node, it is assigned to the highest non-empty local GCS priority. Thus, in its initial attempt at assigning local priorities to a task and all of its GCSs, the algorithm overlaps all of them.

After the assignment of a task and all of its GCSs is done, the algorithm tests the schedulability of the task, accounting for priority mapping (e.g. Equation 1 of Section 3.3). If it is found to be schedulable, the counters on the task's node and all GCSs' nodes are decremented, and the scan goes on to the next higher global priority task. If the task is found to be non-schedulable, the algorithm backtracks, trying another combination of overlaps and non-overlaps of the task and its GCSs.

After scanning through and assigning a local priority to the highest global priority task, the algorithm completes if the counters on all nodes are non-positive. If there are no more tasks to scan and there are still some positive counters, that is there are still some overlaps required, then the algorithm backtracks to try to find another possible combination of overlaps and non-overlaps of the tasks and their GCSs.

Backtrack. The decisions about whether or not to overlap each global priority form a binary tree. The leaves of the tree represent all of the possible combinations of overlaps and non-overlaps in the system. The backtracking involves choosing another one of these combinations and testing its schedulability. While in general, the entire tree may be searched in order to find a successful combination (one that is schedulable), in the Section 4.2 we describe several heuristics for choosing which of the tasks and GCSs to overlap. If, after completely backtracking through the tree, no schedulable solution is found, the algorithm reports that it cannot find a schedulable solution, and then it quits. Otherwise when a schedulable combination is found, the counters on the appropriate nodes (those where overlaps occurred) are decremented and the scan continues.

4.1.2 Example. We now present an example to illustrate how our priority mapping algorithm works. Figure 5 shows a series of snapshots of a system of tasks and GCSs in the process of being mapped. The required number of overlaps for each node is displayed across the top of the figure. The solid lines represent the tasks and the striped lines represent the GCSs. An arrow from a task to a GCS indicates that the task originated the GCS. The brackets indicate a local priority to which tasks or GCSs have already been mapped. In each part of the figure, the tasks or GCSs being considered are highlighted in gray. Figure 5A represents the system before any local priorities have been mapped. In part B, the lowest global priority task is mapped to the lowest local task priority on its node. Part C shows the next two lowest global priority tasks mapped into the lowest local task priorities on their nodes, and the GCS associated with one of them is mapped to the lowest local GCS priority on its node.

Notice that up to this point, no overlaps have been performed because each task and GCS that has been considered has been the first on its node. In part D of the figure, the

indicated task is assigned to the highest non-empty local task priority. After the schedulability is tested, the counter on the task's node is decremented. In part E the task being considered is overlapped into the highest non-empty local task priority, and its GCS is also overlapped into the highest non-empty local GCS priority. However, with both the task and the GCS being overlapped, the task is not schedulable. So in Figure 5F the algorithm has backtracked to attempt another combination of overlaps and non-overlaps of the task and its GCS. The task remains overlapped, but the GCS is assigned to the next lowest empty local GCS priority. Since this configuration is schedulable, the overlap is kept, and the counter on the task's node is decremented. Omitting some intermediate steps, in figure 5G the scan is complete, but one of the nodes has a positive counter. Thus, the algorithm backtracks to the configuration shown in part E of the figure. Recall that in this configuration, when it was found not possible to overlap both the task and its GCS, the algorithm chose to unoverlap the GCS. Now, the algorithm attempts another combination of overlaps and non-overlaps of the task and its GCS. In this case, shown in Figure 5H, the task is unoverlapped and the GCS is overlapped. If the unoverlapped task is schedulable, the algorithm scans the next higher global priority task and continues this way until all tasks have been scanned, and each node has a non-positive counter, or until there are no more configurations to try.

4.1.3. Optimality. Our priority mapping algorithm produces a *direct mapping* of global to local priorities. A direct mapping is one in which if any task (GCS) *i* has higher global priority than any task (GCS) *j*, then task (GCS) *j* cannot have higher local priority than that of task (GCS) *i*. That is, the mapping does not change the relative ordering of task (GCS) priorities. In this section we prove that in the class of direct mappings, the Lowest Overlap First Priority Mapping Algorithm is optimal. That is, if there is a direct mapping of global to local priorities that is schedulable, then the mapping produced by our algorithm is also schedulable.

Theorem 1: For a given schedulable system of tasks and GCSs with global priority assignments, if there is any direct priority mapping under which a the system is schedulable, the system is also schedulable under the Lowest Overlap First Priority Mapping Algorithm.

Proof: The approach we take to proving this theorem is to assume that some schedulable direct mapping exists, and to show that we can derive a Lowest Overlap First mapping from it that is also schedulable.

Let us assume that some direct mapping of global priorities to local priorities exists for a particular node in the system. Assume also that the mapping provides schedulability of the considered system. Let the operating system on the node have *n* local priorities (where *n* is the lowest priority). Because the mapping is direct, any task with local priority *i*, higher than local priority *j*, has higher global priority than any task with local priority *j*. Take the lowest global priority task that is assigned to local priority *n-1* ($t_{n-1,l}$) and temporarily change its local priority to *n*. We can think of this as moving task $t_{n-1,l}$ out of local priority *n-1* and overlapping it into the lower local priority *n*. Figure 6 illustrates this move.

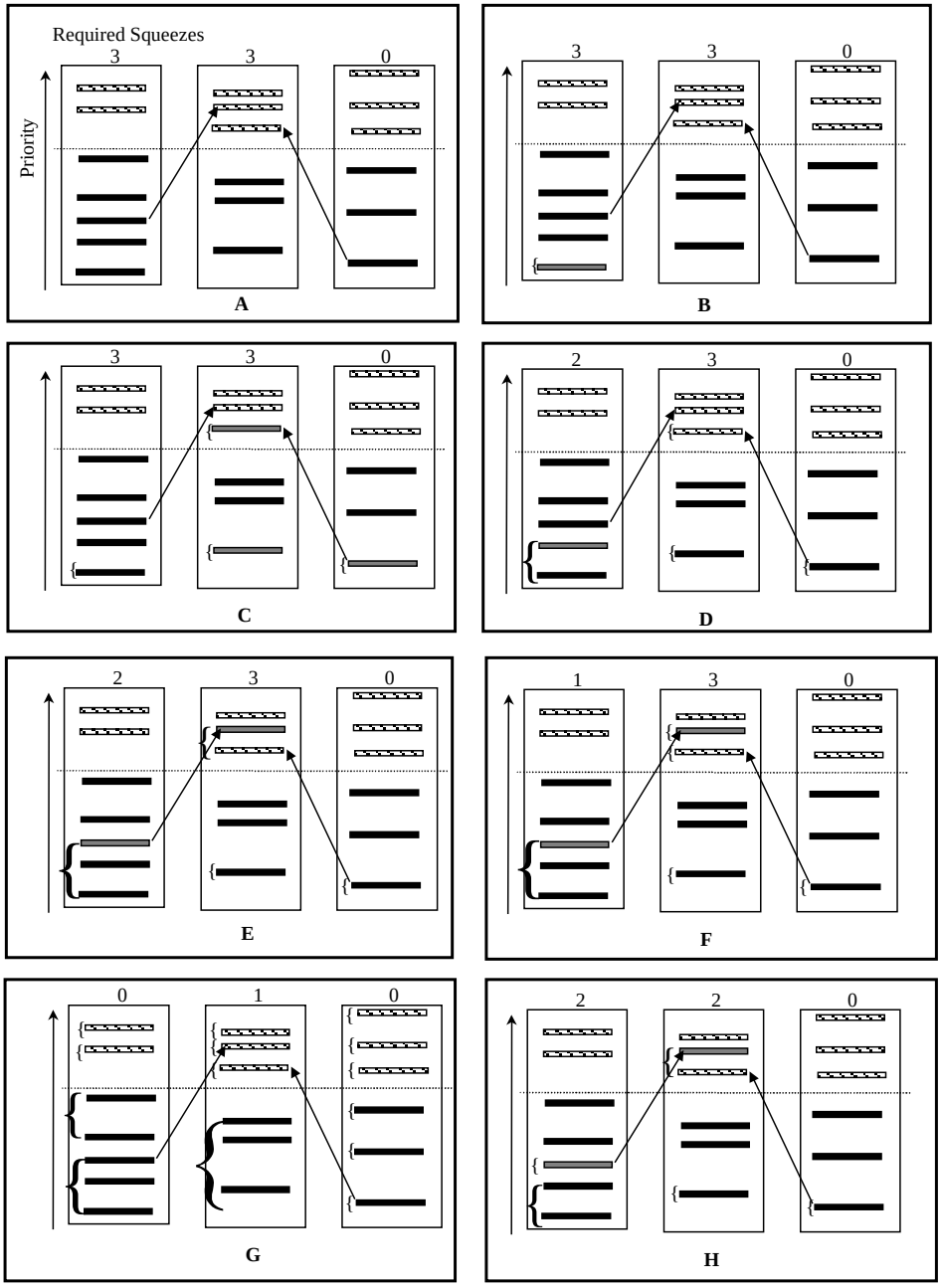

Figure 5 - Lowest Overlap First Priority Mapping Algorithm Example

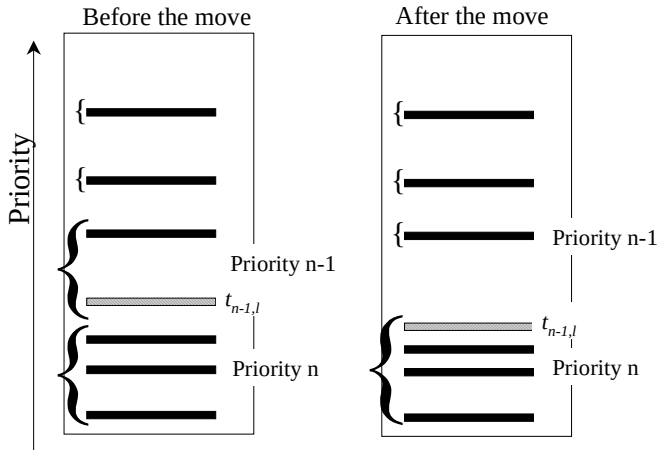

Figure 6 - Example Priority Move

We now examine which tasks' schedulability might be affected by this move.

1. The tasks in local priority *n-1*: The worst case completion time of any task with local priority *n-1* will not increase because all of these tasks could previously have been blocked under FIFO by $t_{n-1,l}$, and now they cannot.
2. The tasks in local priority *n*: The worst case execution time of any task with local priority *n* will not increase because, before the move, any task with local priority *n* could have been preempted by $t_{n-1,l}$. After the move, the tasks in local priority *n* can be blocked due to FIFO scheduling within the same priority. The blocking time cannot be greater than the preemption time.
3. Task $t_{n-1,l}$: The worst case completion time of $t_{n-1,l}$ may be affected by the move, making it unschedulable. However, if this were the case, the Lowest Overlap First algorithm would not have made this overlap in the first place, but rather would have mapped $t_{n-1,l}$ to local priority *n-1*.

If task $t_{n-1,l}$ remains schedulable after "moving" it to priority *n*, we repeat this procedure moving the next lowest global priority from local priority *n-1* to local priority *n*, as long as the moved task remains schedulable. Clearly, if we continue this procedure for local priorities *n-2*, *n-3*, etc., the resulting mapping will be the one that would have resulted from using the Lowest Overlap First Priority Mapping Algorithm. The procedure for moving GCSs is identical with the exception that on every move, we check the schedulability of the task that generated the GCS in question. Since the schedulability of all tasks is not affected by any of these moves, the system remains schedulable and the theorem is proven.■

4.2. A Heuristic for Choosing Overlaps

While Theorem 1 proves the optimality of the Lowest Overlap First Priority Mapping Algorithm in the class of direct mappings, in the worst case the algorithm must search the entire tree each time an overlap is needed, making it an NP-hard problem. We have developed a heuristic for determining what combination of a task and its GCSs to overlap if it is not possible to overlap all of them. The heuristic uses information about which

node is most difficult to overlap, and attempts to overlap the task and as many of its GCSs as possible in a particular order.

4.2.1. Overlap Coefficient Heuristic. We define a value called the *Overlap Coefficient* (OC) for each node at any given time in the mapping process. The OC for node *n* is defined as follows:

$$OC_n = \frac{COUNT_n}{|TASK_{scan,n}| + |GCS_{scan,n}|} X \frac{\sum_{i \in TASK_{scan,n} \cup GCS_{scan,n}} exec_i}{\sum_{i \in TASK_{scan,n} \cup TASK_{GCSscan,n}} slack_i}$$

where:

a) $COUNT_n$ is the number of overlaps to be made on node *n*
b) $TASK_{scan,n}$ and $GCS_{scan,n}$ are the sets of tasks and GCSs (respectively) on node *n* that are left to be scanned
c) $TASK_{GCSscan,n}$ represents those tasks whose GCSs are still to be scanned on node *n.*
d) $exec_i$ represents the worst case execution time of task or GCS *i.*
e) $slack_i$ represents the slack time of task *i*, which is the difference between the task's deadline and worst case completion time

We use this coefficient to determine the order of the nodes on which to attempt to perform overlaps.

In using this heuristic, the Lowest Overlap First Priority Mapping Algorithm executes as described above. However, if at any time in the process it is not possible to overlap a task *t* and all of its GCSs, the following routine is executed:

1. For each node *n* on which resides either task *t* or any of its GCSs, compute OC_n.
2. In decreasing order of OC, try to perform overlaps on each node.
3. If after attempting the overlaps for a particular node, the task remains schedulable, keep the overlap and go on to the node with the next lower OC.
4. If the overlaps on the node render the task unschedulable, then assign the task or GCSs on the node to the next empty local priority. In other words, do not do the overlaps on this node.
5. Go to the node with the next highest OC and continue.

The overlap coefficient is used to help make an intelligent decision about which nodes are least likely to allow overlaps. The coefficient is proportional to the counter because the higher the counter the higher the "priority" of the node to perform the overlap because it needs more overlaps. At the same time, the more tasks and GCSs to be scanned on a node, the more chances there are to make the necessary number of overlap. Simply put, the more tasks and GCSs there are to be scanned, the more possible overlapping combinations there are and the more likely that some of them are schedulable. This is why the coefficient is inversely proportional to the number of tasks and GCSs to be scanned. Also, the longer the execution times of the tasks and GCSs, the harder it is to overlap them. This is why the coefficient is proportional to the sum of the execution times. Finally, the larger the slack times of tasks to be overlapped, and tasks whose GCSs need to be overlapped, the easier it is to overlap the tasks and GCSs. This is why the coefficient is inversely proportional to the sum of slack times.

4.2.1. Complexity Analysis. In the process of calculation of OC for each node, the sums of execution times and slack times are calculated once, before the beginning of the scan. Individual execution times and slack times are subtracted when the task or GCS has been scanned. To estimate the worst case time complexity of the Overlap Coefficient

heuristic we assume that there are *n* nodes, *t* tasks, *g* GCSs in the system and c_i local priorities on the node *i*. Before the beginning of the scan we calculate the sum of slack and execution times of all tasks (and GCSs) on all nodes. The time complexity of this procedure is *t+g+t*.

Every "scan" step involves a decrement of the number of tasks and GCSs to be scanned, which has time complexity is *t+g*. This step also requires the subtraction of the execution and slack times from the totals for the task and GCS under consideration. Thus in the worst case the time complexity associated with the update of the sums of the execution and slack times is *2(t+g)*. Every "overlap" step involves a counter decrement and therefore the time complexity of overlap is $t+g-min\{ c_i \}$. Whenever it is not possible to overlap a task and all of its GCSs, the heuristic compares all the coefficients in order to choose the appropriate node for the "overlap" procedure. In the worst case this could happen *t* times. This involves calculation of the coefficients on candidate nodes (all *n* in the worst case); 3 multiplications/divisions on each node (*3*n* in total); and the sorting of *n* coefficients (*n*log(n)*). Thus summarizing all contributions the time complexity of the heuristic is:

$$t+g+t+t+g+2*(t+g)+t+g-\ \min\{c_i\}+t*(3*n+n*\log(n))=$$
$$t*(6+3*n+n*\log(n))+5g-\min\{c_i\}=O(t*n*\log(n))$$

Note that we do not take into account the time complexity of the schedulability analysis since it is not introduced by this heuristic.

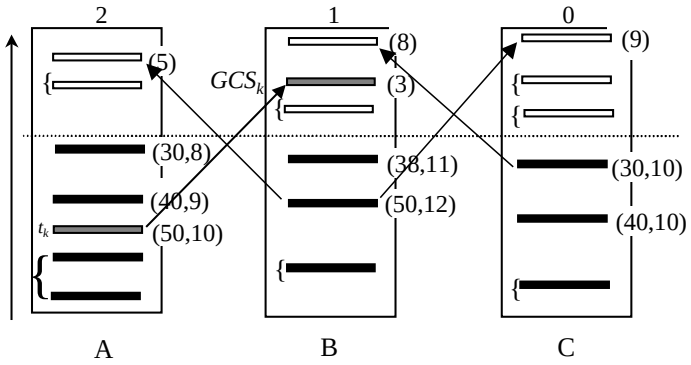

Figure 7 - Overlap Coefficient Heuristic Example.

4.2.1. Example. Consider the example illustrated in Figure 7. The task t_k on node A is the task under consideration, and it is not possible to overlap it and GCS GCS_k. There are 3 nodes to be considered, labeled A, B, and C. The current counter for each node is displayed at the top of the node. For each unscanned task the execution time and slack time are displayed (exec,slack). For each unscanned GCS, the execution time is displayed. The OCs for each node are computed as follows:

$$OC_A = \frac{2}{3+1} X \frac{50+40+30+5}{10+9+8+12} = 1.603$$

$$OC_B = \frac{1}{2+2} X \frac{50+38+8+3}{11+12+10+10} = 0.576$$

$$OC_C = 0$$

Notice that the Overlap Coefficient for node C is zero because there are no overlaps required on the node. We calculate this coefficient only as an example. In reality it would not be calculated since no overlaps induced by task t_k will take place on node C. The Overlap Coefficient heuristic will attempt to overlap the task t_k on node A first, followed by its GCS, GCS_k on node B, testing for schedulability after each overlap. In the implementation that we describe in the next section, we used a much simpler heuristic in the mapping algorithm. We describe the details of this heuristic and the practical reasons why we chose it in Section 5.5.

5. Implementation

We have implemented our *DM + DPCP + Lowest Overlap First Priority Mapping* distributed real-time scheduling technique in a system called *RapidSched* that uses the standard RT CORBA 1.0 Scheduling Service interface described in Section 2. This system consists of a PERTS front-end to generate scheduling parameters and a set of libraries to enforce the semantics of our scheduling approach using the RT CORBA 1.0 Scheduling Service interface.

5.1. PERTS Front-End.

We have developed an extended version of the PERTS (TriPacific) real-time analysis tool to determine the schedulability of a RT CORBA system. PERTS provides a graphical interface to allow users to enter real-time task information, such as deadline, execution time, resource requirements. It then computes a schedulability analysis on the given system using well-known techniques, such as rate-monotonic analysis (Liu 1973,

Lehoczky 1989). PERTS models real-time systems using tasks and resources, the primitives that were described in Section 3.1. We have extended its graphic user interface to facilitate specification of RT CORBA clients and servers using the modeling techniques for those entities described in Section 3.1. This extended version of PERTS analyzes the RT CORBA system using deadline monotonic scheduling and distributed priority ceiling protocol for concurrency control. We have further extended PERTS to allow a user to input the number of local priorities on each node in the system and to apply the lowest overlap first algorithm to compute the priority mapping of tasks and critical sections on each node. If the analysis performed by PERTS deems the system to be schedulable, the extended PERTS system produces global priorities for each client task, priority ceilings for each server resource in the system, the priorities at which to execute all server threads in the system, and the mapped local real-time operating system (RTOS) priorities at which to execute all of these entities. If the system is found to be non-schedulable, PERTS produces graphs and other information for each client task to indicate what caused the system to be non-schedulable.

As an example, consider the case study that is discussed in (Katcher 1995). A high-speed network connects one or more multimedia servers to multimedia workstations where traffic consists of a mixture of video, audio, voice, MIDI, and large file transfers in addition to periodic and aperiodic network management messages.

Figure 8 shows a PERTS output screen with the example. In the top-left box on the screen-shot, the “Algorithm” specified is RM+PCP. This indicates that PERTS is using Deadline Monotonic scheduling with Priority Ceiling resource management (since it is a distributed system, PERTS is using DPCP). In the same box, the “Schedulability Result” indicates that the system is “Schedulable”. The utilization numbers in the top-left box, along with the pie chart in the top-right box indicate utlilzations. The lower half of the

screen shows the six multimedia tasks (PT1-PT6). The static task characteristics "Period" and worst case execution time ("Exec. Time") are shown for each task. Also shown for each task is the "Global Priority" computed by the analysis, and the "Local Priority" computed by the priority mapping.

5.2. Scheduling Service Libraries.

Our DM+DPCP+Priority Mapping scheduling approach described in Sections 3.2 and 4 requires that the Scheduling Service be able to set the CORBA Priority of a client, perform priority mapping to the Native Priority on the client's RTOS, and perform DPCP cuncurrency control and priority setting at the server. RapidSched uses the Real-Time CORBA 1.0 standard interface (see Section 2) implemented with six main library calls that are designed to facilitate its portability to various ORBs and RTOSs. RapidSched makes use of a shared memory segment on each node. In this segment it places the configuration information from the PERTS output file, and also dynamic information such the priority ceilings of servers that are currently executing. This general technique is illustrated in Figure 9

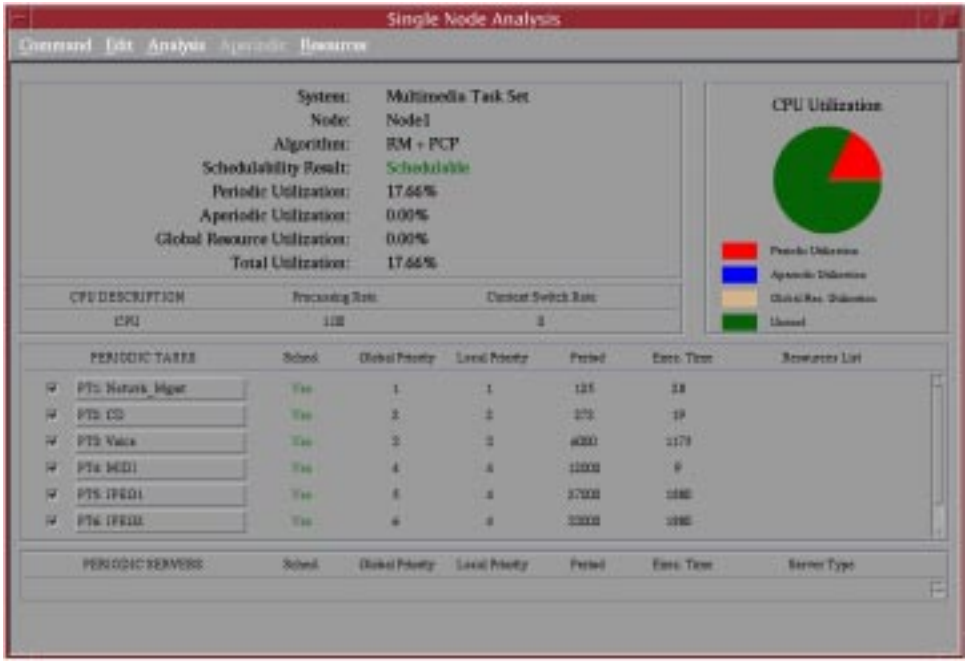

Figure 8 – PERTS Output of for Multimedia Task Set Example

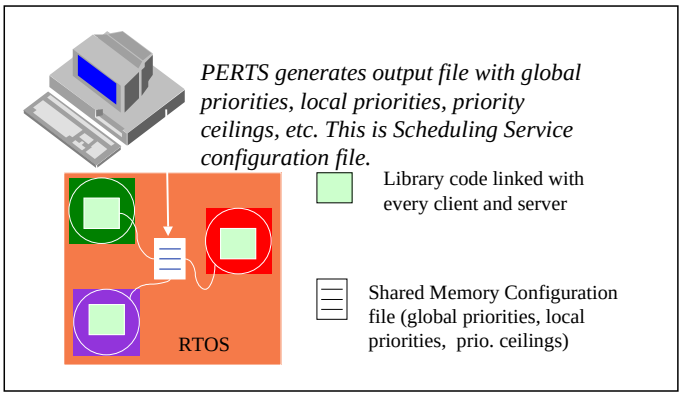

Figure 9 - RapidSched Static RT CORBA Scheduling Service

RapidSched uses three RTOS-specific libraries: for providing a Shared Memory interface, for setting a thread's local (native) priority, and for doing mutual exclusion with priority inheritance. In many POSIX 1003.1c-compliant RTOS's these three library implementations are trivial because the RTOS directly supports the functionality. In other psuedo-RTOS's, like Sun's *Solaris*, the libraries are more complex.

RapidSched has three ORB-specific libraries that must get installed as *interceptors* in the ORB. An interceptor provides the capability for the application programmer to insert code to be executed at various points in the CORBA method invocation. Interceptors are supported by many ORBs and are being standardized by the OMG (OMG2 1998). One RapidSched interceptor is used for performing actions when a CORBA call leaves a client. This interceptor looks up the *activity name* that appears in the client's Scheduling Service call (see Section 2.2.1) in the configuration file in shared memory to obtain the CORBA Priority for the part of the client's execution. Recall that these CORBA Priorities were determined by PERTS, and the PERTS output is the source for RapidSched's shared memory information. A second interceptor is used when the call arrives at the CORBA server. It is this interceptor that performs the DPCP concurrency control check and establishes the correct priority for the servant thread to execute under DPCP. The priority ceiling used in the DPCP check is obtained using the name of the object in the Scheduling Service call and looking up the priority ceiling in RapidSched's shared memory segement (again, this name-to-priority-ceiling association was originally generated by the PERTS analysis). The third interceptor is used for the return of the call from the server; it releases the DPCP "lock" on the CORBA server.

RapidSched has been implemented on various platforms including Iona's *Orbix* on two operating systems: WindRiver's *VXworks*, and Sun's *Solaris*; Sun's *COOL* ORB on Sun's *Chorus* operating system; Objective Interface System's *ORBExpress* ORB on *Solaris*; and Lockheed Martin's *Hardpack* ORB on Lynx's *LynxOS* operating system. Details on these implementations can be obtained from Tri-Pacific Software (TriPacific).

6. Conclusion

This paper has described a technique for real-time fixed priority scheduling and a new priority mapping algorithm and heuristic in middleware for static applications. It assumes the existence of preemptive priority-based scheduling in the real-time operating systems on the nodes in the system. In our technique, client threads have their priorities set using

deadline monotonic assignment of global priorities across the distributed system. Server threads have their priority and concurrency control set using the Distributed Priority Ceiling protocol implemented in the middleware.

The main emphasis of the paper was the presentation of the Lowest Overlap First Priority Mapping algorithm and some associated heuristics. This algorithm is used by the middleware to map the potentially large number of unique global priorities generated by our DM+DPCP approach to the limited priorities provided by commercial real-time operating systems. We proved that the Lowest Overlap First Priority Mapping algorithm is optimal in the class of direct priority mappings. Due to the algorithm's complexity being NP-hard, we also presented the Overlap Coefficient heuristic, which uses the Lowest Overlap First algorithm to gain better performance with a complexity of $O(t*n*log(n))$, in the number of tasks (t) and the number of nodes (n).

The paper also described our implementation of the *DM+DPCP+Lowest Overlap First Priority Mapping* real-time scheduling technique as a Real-Time CORBA Scheduling Service that adheres to the Real-Time CORBA 1.0 Scheduling Service interface. Our Scheduling Service is integrated with an enhanced version of the commercial PERTS real-time analysis tool that provides schedulability analysis and a mapping of global to local priority settings. These settings are automatically used by the Scheduling Service to relieve the application programmer from determining and entering them by hand.

Acknowledgements

This work is supported by the U.S. Office of Naval Research grant N000149610401.

References

T. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaickh, K. Shin, Z. Wang, H. Zou. ARMADA Middleware Suite. In *Proceedings of the 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, December 1997.

E. Bensley, et. al. Object-Oriented Approach for Designing Evolvable Real-Time Command and Control Systems. In *WORDS '96*, February, 1996.

L. DiPippo, V.F. Wolfe, R. Johnston, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh. Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System. *Real-Time Systems*. to be published.

W. Feng, U. Syyid and J. W.-S. Liu. Providing for an Open, Real-Time CORBA. In *Proceedings of the 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, December 1997.

O. Gonzalez, C. Shen, I. Mizunuma, M. Takegaki Implementation and Performance of MidART. In *Proceedings of the 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, December 1997.

IEEE, *IEEE Standard Portable Operating System Interface for Computer Environments (POSIX) 1003.1*, IEEE, New York, 1990.

IEEE. *Proceedings of the 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, December 1997.

Daniel I. Katcher, Shirish S. Sathaye, Jay K. Strosnider. Fixed Priority Scheduling with Limited Priority Levels. *IEEE Transactions on Computers*. Vol. 44, No. 9, pp. 1140-1144, Sept. 1995.

P. Krupp, A. Schafer, B. Thuraisingham, and V.F. Wolfe. On Real-Time Extensions to the Common Object Request Broker Architecture. In *Proceedings of the Object Oriented*

Programming, Systems, Languages, and Applications (OOPSLA) '94 Workshop on Experiences with CORBA, Sept. 1994.

J. Lehoczky and L. Sha. Performance of Real-Time Bus Scheduling Algorithms. *ACM Performance Evaluation Review*, Special Issue, vol. 14, May 1986.

J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the IEEE Real Time Systems Symposium*, 1989.

J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of Machine Scheduling Problem. *Annals of Discrete Mathematics*, 1:343-362, 1977.

C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, vol. 30, pp. 46-61, January 1973.

Jane W. S. Liu, et. al. PERTS: A Prototyping Environment for Real-Time Systems. *Technical Report UIUCDCS-R-93-1802*, The University of Illinois, Urbana, May 1993. Commercial version information available at www.tripac.com.

J. W.-S. Liu. *Real-Time Systems*. To be published by Prentice-Hall, Fall 1999.

Lynx Real-Time Systems, Inc. at http://www.lynx.com/.

OMG. *CORBAServices: Common Object Services Specification*. OMG, Inc., 1996.

OMG. Real-Time Special Interest Group's Request For Proposals. Electronic document at http://www.omg.org/docs/realtime/97-05-03.txt.

OMG (1). *Common Object Request Broker Architecture – Version 2.2*. OMG, Inc., 1998.

OMG (2). *Realtime CORBA*. Electronic document at http://www.omg.org/docs/orbos/98-10-05.pdf.

Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Boston, MA. 1991.

K.Ramamritham, J. Stankovic, and W Zhao. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*. Vol 38, No. 8, pp 1110=1123, Aug 1989.

D. Schmidt, R. Bector, D. Levine, S. Mungee, G. Parulkar. TAO: A Middleware Framework for Real-Time ORB Endsystems. In *Proceedings of the 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, December 1997.

John A. Stankovic, Marco Spuri, Marco DiNatale, and Giorgio Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. IEEE Computer, Vol. 28, No. 6, June 1995.

Jun Sun. *Fixed-Priority End-to-End Scheduling in Distributed Real-Time Systems*. Ph.D. Thesis. University of Illinois, Urbana-Champaign, 1997.

TriPacific Software. at www.tripac.com.

WindRiver Systems at http://www.wrs.com/.

J. Xu and D. Parnas. Scheduling Processes With Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*. Vol. 16, No. 3, pp 360-369, March, 1993.