I Introduction

This thesis provides contributions in real-time scheduling theory and tools for distributed systems. In particular it addresses techniques to perform schedulability analysis of Real-Time CORBA systems.

A *real-time system* is one in which some (or all) jobs have timing constraints. By *job* we mean a basic unit of work to be scheduled and allocated resources. A simple example of a job could be an I/O operation or a granule of computation. The basic timing constraint is a *deadline* – a moment of time by which a job is required to complete.

Traditionally, a real-time system developer starts with programming the application software, and then validates timing constraints, often by using extensive simulations. This approach is excessively time and resource consuming. Under this approach, it is difficult to extend and maintain existing real-time systems: small changes in the application software or underlying hardware can produce unpredictable timing effects that can only be detected and corrected after exhaustive testing. This problem demonstrates a need for automated tools that allow schedulability analysis of a real-time system in the earliest stages of its design or modification. Such a tool, called Prototyping Environment for Real-Time Systems (PERTS), has been developed at the University of Illinois (Urbana, IL) and currently is supported by Tri-Pacific Software division of the Tri-Pacific Consulting

Corporation (Alameda, CA). The development of this tool became possible due to a significant breakthrough in the solution of numerous schedulability problems [1]. PERTS is described in detail in Chapter II.

Distributed object computing has become a widely accepted programming paradigm for applications that require seamless interoperability among heterogeneous clients and servers. The Object Management Group (OMG) has developed the Common Object Request Broker Architecture (CORBA) as a standard software specification for such distributed environments. A great demand for Real-Time (RT) CORBA has motivated the Real-Time Research group at the University of Rhode Island to develop the first version of the RT CORBA in 1997 [2, 3]. Real-Time CORBA is described in Chapter IV.

The possibility of preliminary schedulability analysis of the designed system has a great value for the developers of distributed real-time applications. Unfortunately, PERTS does not fully support the analysis of RT CORBA distributed applications. After detailed analysis of PERTS capabilities, we have concluded that the present PERTS version may be modified to model and analyze RT CORBA, as we describe it in Chapter IV.

In Chapter V we discuss necessary PERTS modifications and describe the implementation.

The test cases that demonstrate the correctness of the modified and new PERTS components are presented in Chapter VI.

Along with the goal of modeling RT CORBA and implementing necessary PERTS modification, we have considered various theoretical issues. PERTS uses two widely accepted schedulability criteria based on Liu-Layland’s and Lehoczky’s conditions [1, 10]. In Section 2.3.3 we prove that the satisfaction of Liu-Layland’s condition automatically guarantees the satisfaction of Lehoczky’s condition. We also have considered the nontrivial case of a task system that contains harmonic tasks. In Section 7.2.1 we demonstrate that, in this case, Lehoczky’s schedulability condition is not necessary, but sufficient only. We present the modifications for the criterion to make it necessary in the described case.

Thus, in this thesis we address techniques to perform schedulability analysis of Real-Time CORBA systems. In addition, we discuss a list of various aspects of the schedulability theory, including: modification of the Lehoczky’s schedulability criterion for the systems containing harmonic tasks and for the RT systems built on operating systems with limited available priorities; comparison of the Liu-Layland’s and Lehoczky’s criteria for the systems with shared resources; description and comparison of two resource access protocols.

II PERTS

In this chapter we review PERTS 3.0 and its abilities to describe and analyze real-time systems. For its complete description, we refer the reader to the PERTS manual available on-line [4]. In this chapter we concentrate only on those PERTS features that are important for its extension to analyze RT CORBA.

PERTS is a Prototyping Environment for Real-Time Systems. It contains tools for the analysis, validation and evaluation of real-time systems. It includes an extensible library of priority scheduling algorithms and resource access protocols.

In order to validate real-time system timing constraints and evaluate its performance, the system parameters must be described. This description includes: the workload to be executed, the resources available to support the workload, and the algorithm used to assign priorities and allocate resources. PERTS provides such a description environment through the *Task Graph Editor, Resource Graph Editor,* and *Schedulability Analyzer*, described in the next three subsections.

2.1 Task Graph Editor

A Task Graph describes the application system, called the *task system*. It includes a set of tasks of the system being modeled. The tasks could be *periodic*, when time between two consecutive Ready Times is constant, or *aperiodic*, in other cases. Since aperiodic tasks go beyond the scope of our study, we will exclude from future consideration in this project all features associated with them. The tasks may be dependent on each other.

PERTS calls a collection of all tasks and their dependencies a *Task Graph*. Every task and dependency is characterized by a set of parameters. To describe a Task Graph, the user must provide a complete set of parameters for every task and dependency in that Task Graph. The demand of a user-friendly interface to create and edit Task Graphs has stimulated a development of PERTS Task Graph Editor.

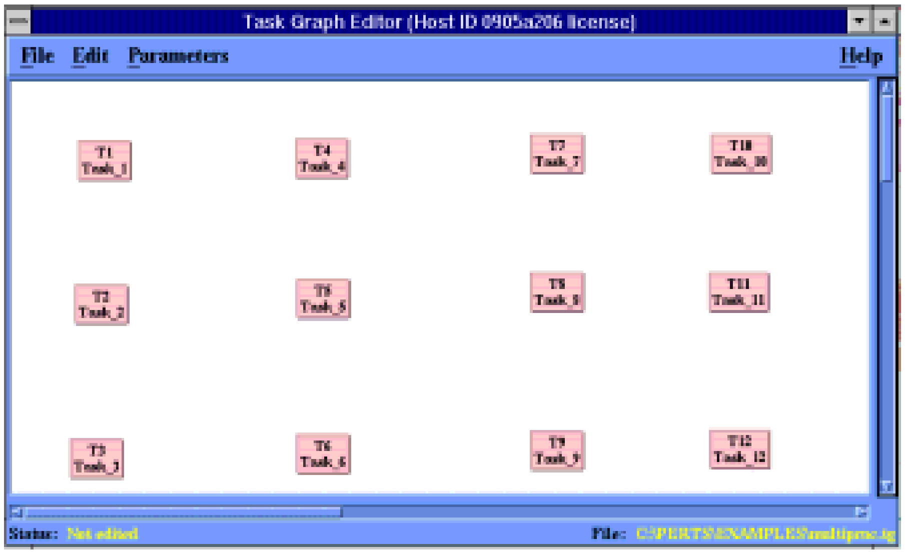


Figure 2.1. Schematic view of the Task Graph Editor.

The Task Graph Editor, shown in Figure 2.1, enables the user to create and update a Task Graph. It provides a graphical representation of the Task Graph. All tasks of a Task Graph are represented by rectangular nodes, all dependencies are presented by directed edges connecting the appropriate nodes. A description of a Task Graph in this environment is performed by choosing an appropriate operation in a menu bar and clicking on the appropriate node or edge. To describe the set of available in Editor operations/commands, we present them in groups, as they are arranged in menu bar.

File Commands enable the user to create a new Task Graph (*New*), open an existing Task Graph (*Open*), re-initialize already open Task Graph (*Reopen*), save current Task Graph (*Save*), save a new copy of current Task Graph (*Save As*), print current window (*Print Entire Window*), create a report of Task Graph information (*Generate Report*), launch any of the other PERTS tools or exit the Task Graph Editor (*Quit*).

Edit Commands enable user to manipulate task nodes and task dependencies. User can add task node (*Add Task*), add dependency edge (*Add Dependency*), copy task characteristics (*Copy Task Parameters*), move task nodes to the new position on a screen (*Move Task*), delete task nodes or dependencies (*Delete*) or undo an unintentional edit command (*Undo* (*Add* or *Delete*)).

Parameter Commands enable user to enter and change task parameters for each task in the task graph. Since we are interested in the periodic tasks only, we describe here only menu (and operations) for the periodic task parameters. It includes the options to enter and edit the *General Task Data*, *Optional Intervals*, *Non-Preemptable Sections*, *Resource Requirements* and *User Specified Priorities*.

By clicking on General Task Data menu bar, the user pops up an edit dialog window, which enables input and edit of general task information for any task in the open Task Graph. General Task Data include the following parameters (we omit here some of the parameters irrelevant to our study):

- *Task Name*,
- *Ready Time* – the earliest time instant at which the task may begin execution,
- *Relative Deadline* – time frame after Ready Time within which the task must finish execution (reader can find in the literature a term Absolute Deadline, which is a sum of Ready Time and Relative Deadline),
- *Period* – constant length of time between two consecutive Ready Times of the task,
- *Phase* – the time at which the task starts its first period,
- *Active Resource* – the CPU the task should run on,
- *Amount of Work* – the execution time for the task.

The General Task Data Edit Dialog allows user to enter the appropriate task data in the window, update the General Task Data (by clicking on *OK*), print the screen to a file or to a printer (*Print*), cancel the update (*Cancel*) and view the help window (*Help*).

In addition to the described General Task Data, every task is characterized by a list of *Optional Intervals*, *Non-Preemptable Sections* and *Resource Requirements*.

Normally, a task, once scheduled, executes entirely. However, some tasks contain optional parts, which are specified by means of Optional Intervals. They are characterized by a Start and End Time. The task may contain more than one Optional Interval.

In a preemptive environment a task may be preempted by another task of higher priority. Sometimes a task should not be preempted during some certain sections of its execution called Non-Preemptable Sections. Similar to Optional Intervals, they are characterized by a Start and End Time. A task may contain more than one Non-Preemptable Section.

A task may require a use of one or more resources during its execution. The resource requirements are described by *Resource Name*, *Start Time* and *End Time*.

To edit one of the described objects (Optional Intervals, Non-Preemptable Sections or Resource Requirements), the user clicks on appropriate menu bar to pop up a corresponding Edit Dialog Window. Every Window contains the appropriate fields for editing the object parameters, including summary on all objects of specified type. User can enter the object parameters into the Window Dialog, add an appropriate object (by clicking on *Insert*), remove an object (*Delete*), modify parameters of an existing object (*Modify*), update the data (*OK)*, cancel the update (*Cancel*) and view the help window (*Help*).

2.2 Resource Graph Editor

The Resource Graph describes the physical and logical resources available to the task system. It includes all the resources of the system and their "relationships". By relationship PERTS means that the resources may be included (*a-part-of* type) or accessed (*accessible-from* type) by another resources. A database residing at a Node is an example of *a-part-of* relationship (where the database is a part of the Node). A database accessible from another Node is an example of *accessible-from* relationship (where a database is accessible from the Node).

To describe a Resource Graph, the user must provide a complete description of every resource and its relationship with other resources. The demand of a user-friendly interface to create and edit Resource Graph has stimulated a development of the Resource Graph Editor.

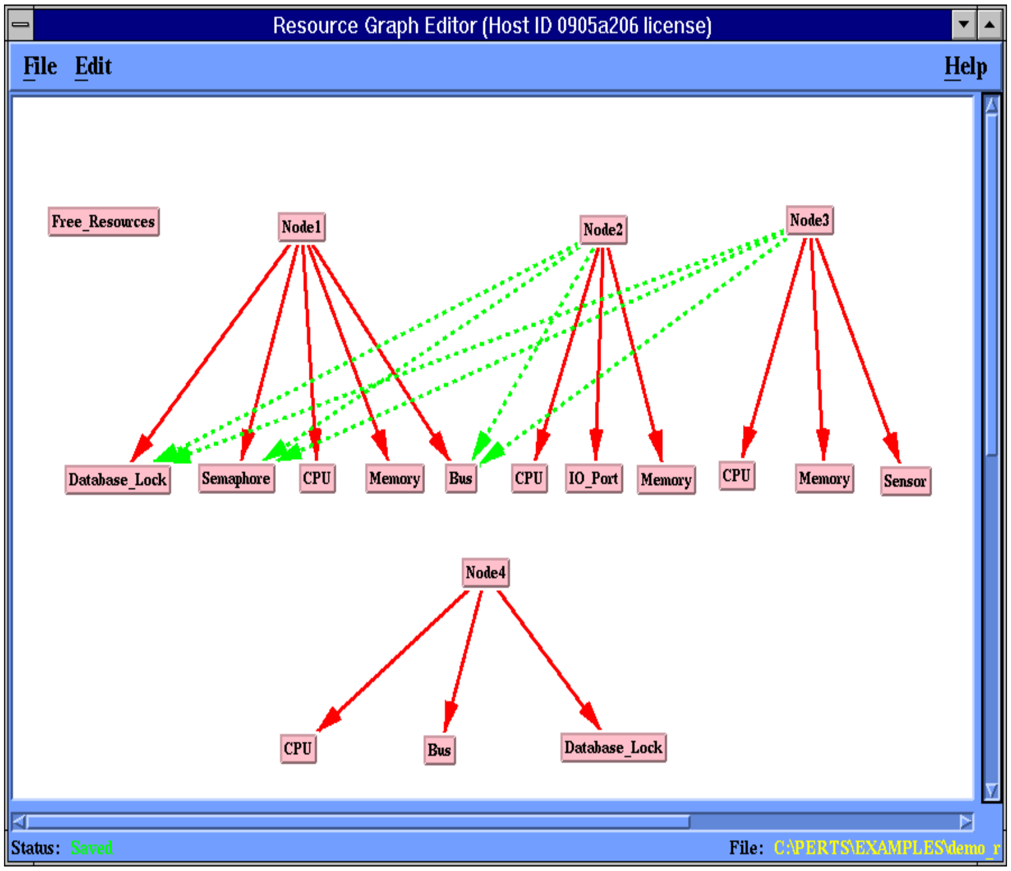


Figure 2.2. Schematic view of the Resource Graph Editor.

The Resource Graph Editor, shown in Figure 2.2, enables the user to create and update the Resource Graphs. All resources of a Resource Graph are represented by rectangular nodes, all relationships – by directed edges connecting the appropriate nodes (solid red for *a-part-of* and dashed green for *accessible-from*). Similar to the Task Graph Editor, the description of a Resource Graph in this environment is performed by choosing an appropriate operation in a menu bar and clicking on the appropriate node or edge.

2.3 Schedulability Analyzer

There are two complementary techniques in evaluation of the timing behavior of a real-time system: 1) schedulability analysis based on theoretical calculations and 2) simulation. The schedulability analysis provides rigorously derived results on whether timing constraints are met, but requires an analyzable model of the studied system. On the other hand, the simulator provides no guarantees; it determines whether timing constraints are violated, relying on user's specification of the worst-case configuration. However, the simulator can deal with a higher complexity model of the system than the schedulability analysis. This project has concentrated on the schedulability analyzer, which can guarantee system schedulability. We would like to emphasize that since the schedulability analysis is based on sufficient (not necessary) criteria, it could not guarantee non-schedulability of a task system. If a task system does not satisfy these criteria it does not mean that it is not schedulable, but instead it means that theory is unable to guarantee its schedulability.

The Schedulability Analyzer is the last of the three PERTS key components. It performs the schedulability analysis for the systems that have been described using the Task Graph and Resource Graph. However, the system is not described completely until the user specifies the *Priority Assignment Mechanism* and the *Resource Access Protocol* for the system. Using the Schedulability Analyzer, a user can choose the appropriate Priority Assignment Mechanism and Resource Access Protocol.

2.3.1 Priority Assignment Mechanisms and Resource Access Protocols

The list on priority assignment mechanisms in the Schedulability Analyzer includes:

- *Rate Monotonic (RM)* [1] - which assigns higher priority to a task executing at higher rate,
- *Deadline Monotonic (DM)* [5] - which assigns higher priority to a task with shorter relative deadline,

- *Earliest Deadline First (EDF)* [1] - which assigns a higher priority to a task with faster approaching deadline.

There are three other Priority Assignment Mechanisms currently supported by PERTS, Cyclic Executive (CE), Harbour-Klien-Lehoczky (HKL) and Sun-Gardner-Liu (SGL), not applicable for our study because of their limitations. CE is applicable for a system containing harmonic tasks only, while HKL and SGL prohibit resource accesses.

The list of available Resource Access Protocols includes:

- *Priority Ceiling Protocol (PCP)* [6,7,8] - which avoids deadlocks, limits blocking time and guarantees that the blocking time is a function of duration of critical sections only;
- *Basic Inheritance Protocol (BIP)* - which is similar to PCP. It is easier in implementation than the latter, but does not limit the number of times a task may be blocked and does not prevent deadlocks;
- *Stack Based Protocol (SBP)* [9] - which assigns a fixed preemption level to every task inversely proportional to its relative deadline. It avoids deadlocks and multiple blocking, but applicable only to Single-Node systems.

After a user specifies the Priority Assignment Mechanism and the Resource Access Protocol and chooses the appropriate textual files with description of the Task and Resource Graphs, the system is completely described.

2.3.2 Schedulability Analysis Regimes

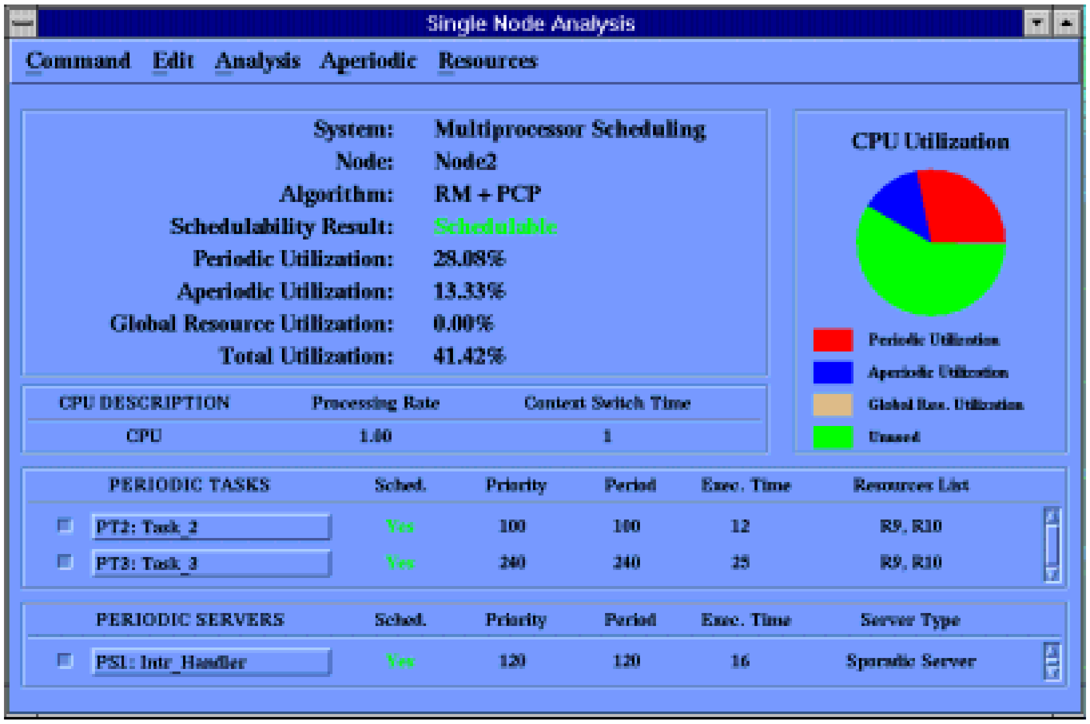
There are three different regimes of analysis provided by PERTS: *Single-Node*, *Multiple-Node* and *End-to-End*.

Single-Node Analysis, shown in Figure 2.3, determines whether the node is schedulable. A *task is schedulable* if it always completes its execution before its deadline; a *node is schedulable* if all the tasks assigned to that node are schedulable. In addition to the report on schedulability of the node, the Single-Node Analysis reports the CPU utilization, and it provides the list of all tasks indicating their schedulability. The user can modify the system parameters to allow “what if?” modeling.

The title “Single-Node” can be misleading. This type of analysis may be used for the systems consisting of a single node, as well as for multiple node systems. In the latter case, the user should choose for analysis either Multiple-Node or End-to-End Analysis, described below. However, to obtain details on the particular node of the distributed system the user may use the Single Node Analysis.

Multiple-Node Analysis, shown in Figure 2.4, examines the schedulability of multiple-node real-time systems. To allow “what if?” modeling, the Multiple-Node Analysis interface enables modification of the binding of tasks and resources to different system nodes. The binding may be either manual or automatic, using such algorithms as best fit, first fit, next fit and worst fit.

The benefit of being able to analyze system architectures that have more than one node and share resources can be critical for distributed real-time system developers. PERTS can help point out potential overhead problems and blocking problems that may be introduced by sharing resources across the nodes. Individual entities may be schedulable as stand-alone entities, but when put in a multiple-node architecture with the resource sharing, they may become non-schedulable. Multiple-Node Analysis reports the system schedulability and then user may select individual nodes to analyze it with Single-Node Analyzer.



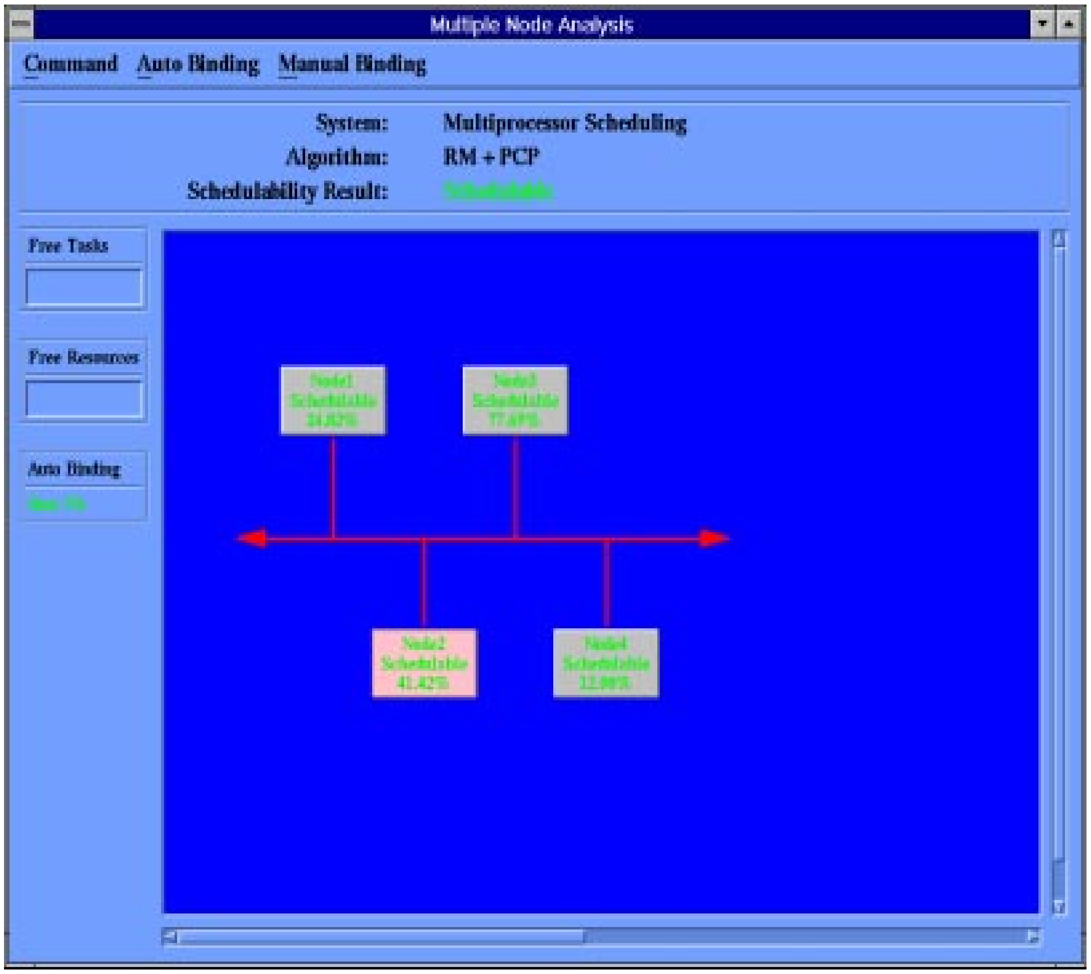
System:	Multiprocessor Scheduling
Node:	Node2
Algorithm:	RM + PCP
Schedulability Result:	Schedulable
Periodic Utilization:	28.08%
Aperiodic Utilization:	13.33%
Global Resource Utilization:	0.00%
Total Utilization:	41.42%

CPU DESCRIPTION	Processing Rate	Context Switch Time
CPU	1.00	1

PERIODIC TASKS	Sched.	Priority	Period	Exec. Time	Resources List
PT2: Task_2	Yes	100	100	12	R9, R10
PT3: Task_3	Yes	240	240	25	R9, R10

PERIODIC SERVERS	Sched.	Priority	Period	Exec. Time	Server Type
PS1: Intr_Handler	Yes	120	120	16	Sporadic Server

Figure 2.3. Schematic view of the Single Node Schedulability Analysis.



System: Multiprocessor Scheduling
Algorithm: RM + PCP
Schedulability Result: Schedulable

Figure 2.4. Schematic view of the Multiple-Node Schedulability Analysis.

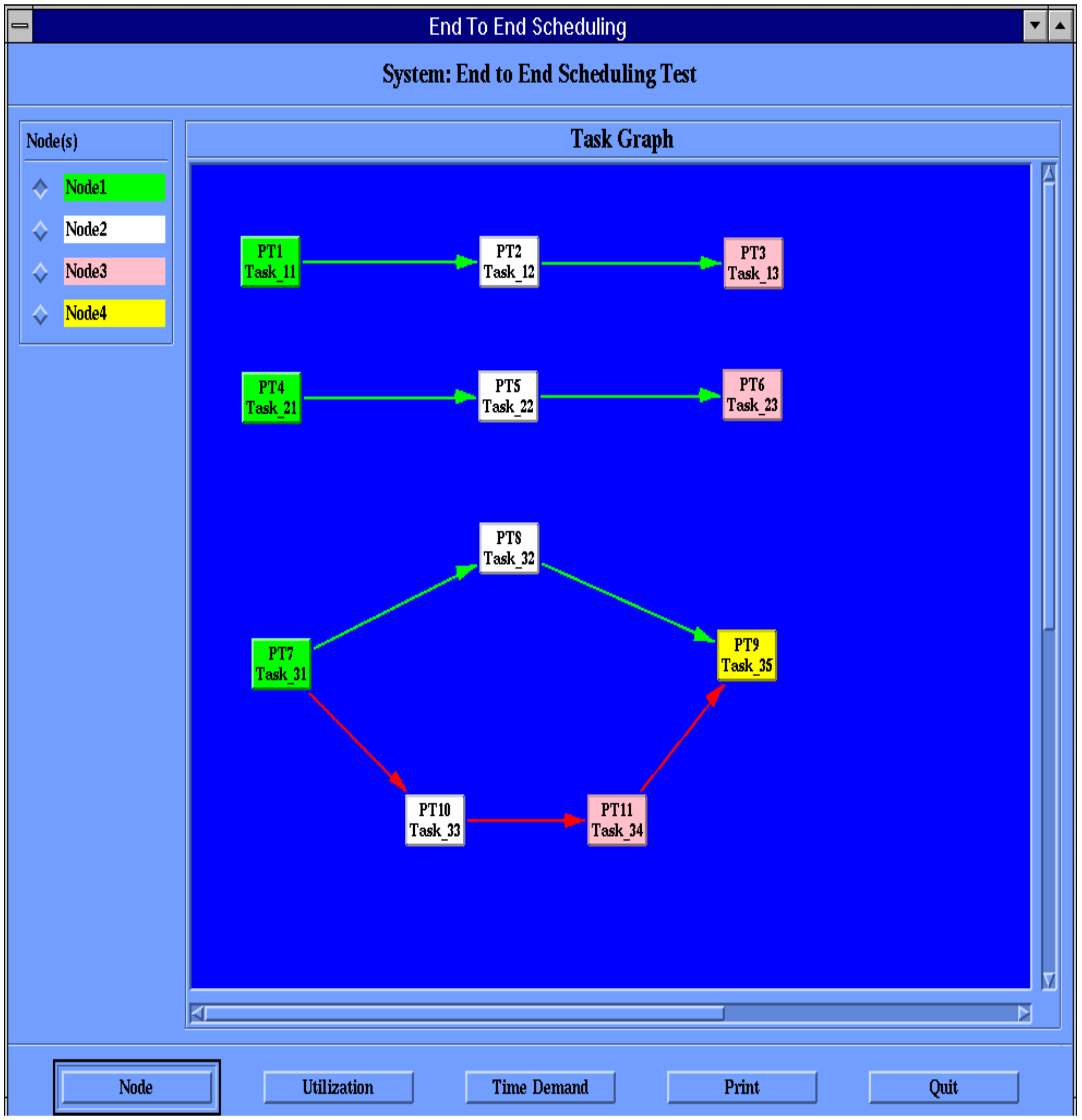


Figure 2.5. Schematic view of the End-to-End Schedulability Analysis.

Both Single-Node and Multiple-Node Analysis dialogs offer a node-oriented view of the system under consideration. They do not perform any path analysis in the systems with task dependencies.

End-to-End Analysis, shown in Figure 2.5, looks at the schedulability of a system with one or more paths of execution defined by a series of task dependencies. The End-to-End Analysis Window graphically represents all tasks and dependencies (similar to Task Graph Editor). Specifying any path, the user obtains a schedulability report on that particular path. The user may choose Single-Node Analysis for the detailed information on the particular node.

2.3.3 Schedulability Analysis: How It Is Done.

The main feature of all three analysis regimes in PERTS is the ability to guarantee the system schedulability. In this section we describe the theory underlying this analysis. There are two sufficient conditions for the schedulability of a real-time system. One of them is based on the concept of Processor Utilization Bound introduced by Liu and Layland [1] and another – based on the concept of Processor Time Demand introduced by Lehoczky *et. al.* [10].

The first criterion requires satisfaction of the following inequality:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1)$$

where tasks are indexed in the decreasing priority order (task T_1 has the highest priority on the considered node). C_j and T_j denote the worst-case execution time and period of the task T_j. B_i is the worst-case blocking time potentially suffered by any job in the task T_i due to resource contention or non-preemptive execution of lower-priority tasks. If a task T_i satisfies this condition, it is schedulable by the RM or DM and PCP or SBP. Schedulability of all tasks of the system means the system is schedulable.

When a fixed-priority scheduling algorithm is used with a resource access control protocol that effectively controls priority inversion, there is another more accurate schedulability condition [7, 10, 11]. This condition is stated in terms of the worst-case cumulative demand function $W_i(t)$ for processor time in the interval between the release time of a task T_i and the time t units after its release. The demand function $W_i(t)$ is given by

$$W_i(t) = \sum_{j=1}^{i-1} C_j \left\lceil t/T_j \right\rceil + C_i + \beta_i$$

The demand function has three parts: the processor time demand by all tasks with priorities equal or higher than T_i, the demand of T_i itself, and the worst-case blocking time suffered by each job in T_i. The job released at time t_0 completes at time t_0+t, if

$W_i(t)=t$. Consequently, whenever $W_i(t)\leq t$ for some t smaller than task T_i Relative Deadline, the task T_i's is schedulable.

The PERTS Schedulability Analyzer checks both criteria. Satisfaction of at least one of them by all tasks guarantees schedulability of the system. This approach should yield a correct schedulability result if both criteria are implemented correctly. Unfortunately, we have detected some theoretical and implementation errors in the present PERTS version, which yield the misleading results of the analysis. In Section 7.3.3 we summarize these errors, demonstrate counter-examples and provide the solutions. To reduce the effect of the implementation errors we recommend using only Lehoczky's condition for the schedulability analysis. We prove below that the satisfaction of the Liu-Layland's condition automatically leads to the satisfaction of the Lehoczky's criterion.

Theorem 1: If any task system satisfies the schedulability criterion based on Liu-Layland's condition (called in future L-L) it also satisfies the schedulability criterion based on Lehoczky's condition (Leh).

Proof:

Since a task system is said to be schedulable if all of its tasks are schedulable, the proof of the Statement for an arbitrary task proves it for a task system.

Let us consider an arbitrary task T_i, which does not use any shared resources. We also assume, that L-L schedulability criterion is satisfied for this task:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} \le i(2^{1/i} - 1)$$

Since the L-L condition is sufficient [1] and the Leh condition is necessary [10] (under the assumption that there are no shared resources used by the task) it follows automatically, that satisfaction of the L-L condition leads to the satisfaction of the Leh condition. Therefore there exists some value of time *t*, less than or equal to the Relative Deadline of the task T_i, such that

$$W_i(t) = \sum_{j=1}^{i-1} C_j \left\lceil t/T_j \right\rceil + C_i \le t$$

The theorem has been proven for the tasks without shared resources.

Let us assume now that the task T_i uses shared resources. It leads to some potential blocking B_i of the task T_i. Under these circumstances the Leh condition is not necessary anymore. In this case we can not use the same approach as in previous one. Introduction of the shared resources (and therefore blocking B_i) modifies the L-L and Leh conditions to be read

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1)$$

$$W_i(t) = \sum_{j=1}^{i-1} C_j \left\lceil t/T_j \right\rceil + C_i + B_i \le t$$

respectively. Let us consider now another task *T'*$_i$, which has identical to the task T_i parameters, but does not use shared resources and its execution time C'_i is equal to C_i+B_i. Let us also assume that task T'_i satisfies the L-L condition:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C'_i}{T_i} \le i(2^{1/i} - 1)$$

As it was proven in the first part of this proof, the satisfaction of the L-L condition for a task that does not use shared resources automatically leads to the satisfaction of the Leh criterion:

$$W_i(t) = \sum_{j=1}^{i-1} C_j \left\lceil {t}/{T_j} \right\rceil + C_i' \le t$$

Replacing C'_i by its equivalent C_i+B_i in both conditions, we obtain that satisfaction of L-L criterion for a task, using shared resources,

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1)$$

automatically leads to the satisfaction of the Leh criterion

$$W_i(t) = \sum_{j=1}^{i-1} C_j \left\lceil {t}/{T_j} \right\rceil + C_i + B_i \le t$$

It proves the Statement for the remaining case of tasks, using shared resources. The Theorem is proven. ■

III Real-Time CORBA Systems

3.1 CORBA

The Common Object Request Broker Architecture (CORBA) is an answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or what underlying system they use. CORBA provides a uniform way for any object to receive and respond to a request from any requester (client).

The *Object Request Broker* (ORB), CORBA's key component, is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The ORB facilitates the processing of client requests. A client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

To provide these capabilities, the CORBA specification defines an architecture of interfaces that may be implemented in different ways by different vendors. The architecture was specifically designed to separate the concerns of interfaces and implementations. The architecture, shown in Figure 3.1, has been described in detail [12].

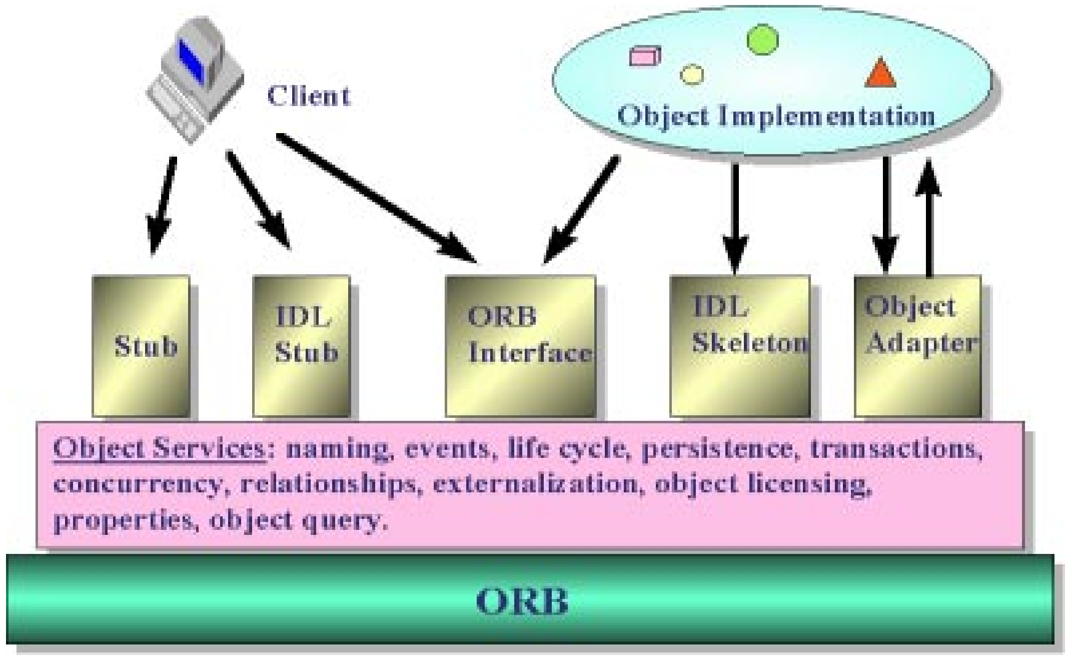

Client
Object Implementation
Stub
IDL Stub
ORB Interface
IDL Skeleton
Object Adapter
Object Services: naming, events, life cycle, persistence, transactions, concurrency, relationships, externalization, object licensing, properties, object query.
ORB

Figure 3.1. Schematic view of the CORBA.

3.2 Real-Time CORBA

Real-Time distributed applications such as automated factory control, avionic navigation and simulation have demonstrated the need to extend the CORBA standard to support real-time. The Real-Time Research group at the University of

Rhode Island has developed the first version of RT CORBA. The group was concentrated on CORBA/RT desired capabilities involving object services and features for handling real-time client/server interaction and addressed object services desired capabilities that are essential for expressing and enforcing timing constraints. These desired capabilities are expressing and enforcing timing constraints on CORBA method invocations, synchronized clocks, Global Time Service, and Real-Time Event Service, shown in Figure 3.2.

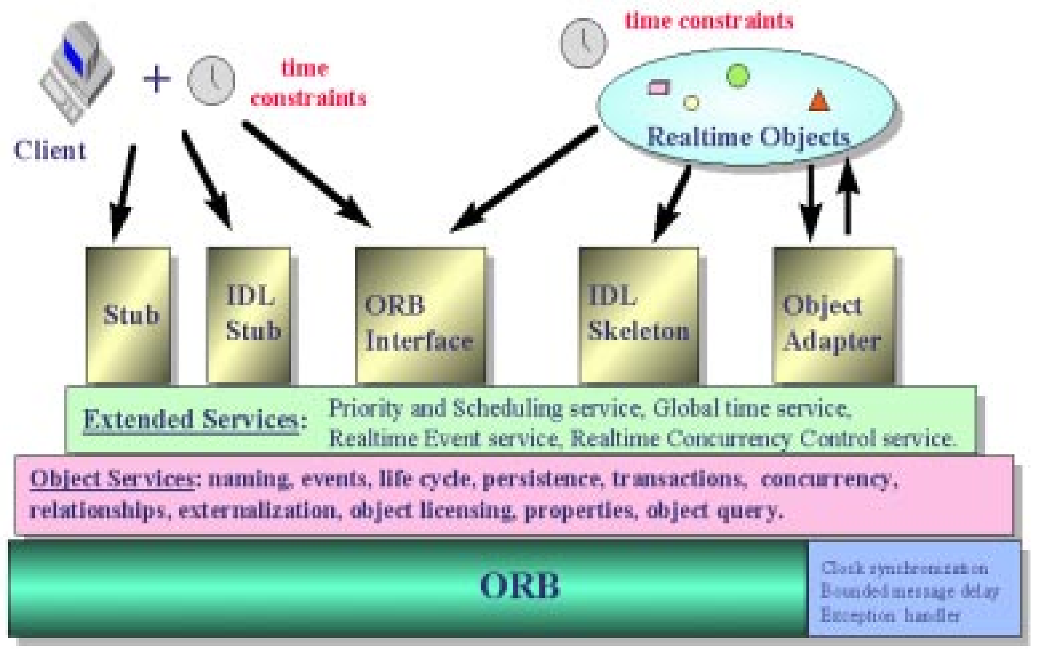

time constraints
time constraints
Client
Realtime Objects
Stub
IDL Stub
ORB Interface
IDL Skeleton
Object Adapter
Extended Services: Priority and Scheduling service, Global time service, Realtime Event service, Realtime Concurrency Control service.
Object Services: naming, events, life cycle, persistence, transactions, concurrency, relationships, externalization, object licensing, properties, object query.
ORB
Clock synchronization
Bounded message delay
Exception handler

Figure 3.2. Schematic view of the RT CORBA.

A RT CORBA client contains a set of requests to RT CORBA servers (method calls) intermixed with its local code. In addition to its final timing constraint (deadline) a client may contain a series of intermediate timing constraints (*Intermediate Deadlines*), associated with different method calls, calculations and data manipulations. The Intermediate Deadline is a crucial RT CORBA feature that PERTS 3.0 does not currently support. An Intermediate Deadline is specified by three time parameters: *Start Time* and *End Time* and *Deadline*. The Start Time and End Time describe the beginning and end of the portion of the client code to be completed by the Deadline.

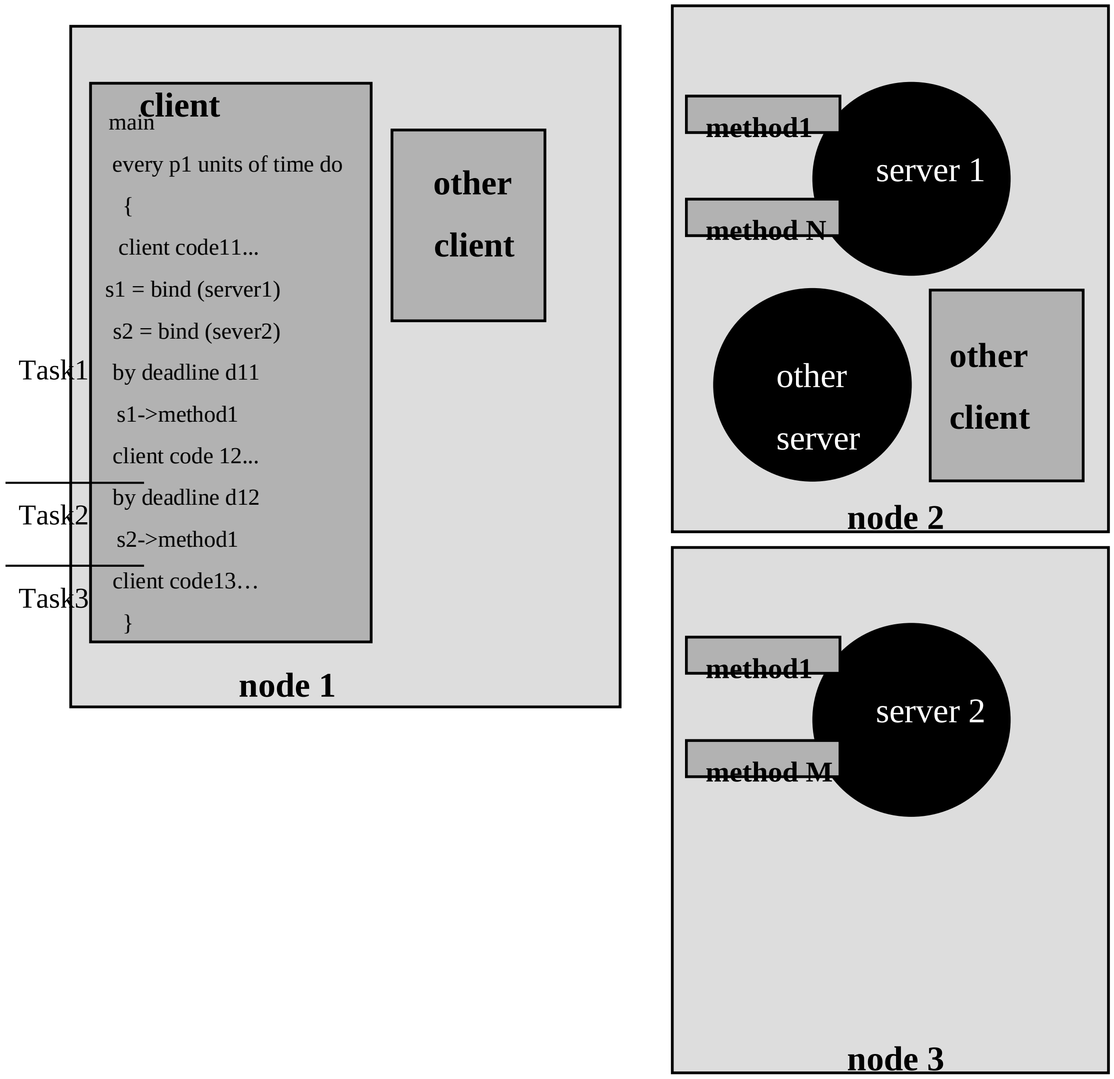


Figure 3.3. Example of the RT CORBA System.

We illustrate a typical RT CORBA client in Figure 3.3. This client (residing on node 1) has a period of P_1 units of time during which it makes two CORBA calls to remote CORBA servers (s1->method1 and s2->method1). Each CORBA call has its own pre-period deadline, shown by horizontal line (d11 for s1->method1 and d12 for s2->method1). Note that there is also some local client code before, after, and between CORBA method calls. The figure shows remote servers only, while in general some or all servers could reside on the client's node.

The distributed CORBA architecture causes network communication between clients and servers residing on different nodes. The time that clients spend sending requests to remote servers (*Network Delay*), may be significant enough to make the clients non-schedulable. This demonstrates the need to take the Network Delay into account in the schedulability analysis, as shown in the next chapter.

To enable the analytical schedulability analysis by PERTS, Real-Time Research group is currently modifying the existing Dynamic RT CORBA software (Orbix on Solaris) to handle Static RT CORBA. It involves a transition from currently supported by RT CORBA dynamic Earliest Deadline First (EDF) priority assignment (which does not allow an analytical schedulability analysis) to the Deadline Monotonic (DM). The latter enables preliminary priority assignment and use of the Distributed Priority Ceiling Protocol improving the worst case priority inversion bound relative to currently supported Basic Priority Inheritance (BPI).

IV Modeling RT CORBA with PERTS

4.1 Problem Domain

We are interested in end-to-end analysis of RT CORBA clients making distributed method calls to CORBA servers over a network. That is, client(s) and server(s) are potentially on different nodes in a distributed system. For this project, the clients will be periodic, with known execution times and we will allow multiple intermediate (pre-period) deadlines within the clients. The server methods will have known execution times, but will get their timing constraints and the basis of their priorities from the clients that invoke them.

The goal of PERTS analysis of RT CORBA is to determine whether all clients meet their timing constraints (final and all, if any, intermediate deadlines).

4.2 Mapping RT CORBA to PERTS

The RT CORBA system components are mapped to the PERTS primitives as follows:

4.2.1 CORBA Servers

All CORBA Servers, as well as CORBA ORB and Services, are represented by PERTS ***resources.*** For example, each server in Figure 3.3 would be modeled as a PERTS ***resource.***

4.2.2 CORBA Clients

The PERTS does not support the schedulability analysis of Intermediate Deadlines. Thus, the PERTS analysis for RT CORBA requires nontrivial client modeling.

Each client with N intermediate deadlines will be modeled as N+1 PERTS ***tasks***. For example, the detailed client in Figure 3.3 would be modeled as three ***tasks***:

- Task1: includes: the bind calls, "client code11", "s->method1", and "client code12.
- Task2: includes: "s->method2".
- Task 3: includes: "client code13".

We must emphasize that Task2 may start its execution only after Task1 completes, and Task3 – only after Task2 completes. Thus the analysis must find for each task a *Ready Time* (the earliest time instant at which a task may start execution), that guarantees serialization of the tasks and correctly describes their dependencies.

We have considered two different approaches to specify task dependencies within a client.

The first approach is based on modeling the task dependencies by *timing constraints*, having the release time of task *i* be the deadline of task *i-1*. This is a pessimistic way to model dependencies for two reasons. First, the deadline of the previous task may have much greater value than its Completion Time (the time it takes to finish execution in the worst case), which means that a task *i* will be idle for a long time after the previous task *i*-1 completed its execution. In this situation the task *i* more likely will miss its deadline. Second, all the tasks generated by the same client are harmonic and have different release times. These conditions guarantee that the "worst phasing condition" (when all tasks are ready for execution at the same time) will never happen. Unfortunately, neither schedulability theory nor PERTS consider this case, and as a result it leads to an overpessimistic evaluation of the system schedulability. We present corrections need to be made to the Lehoczky's condition to account for this harmonic tasks case in Section 7.3.1.

The second approach to specify the task dependencies within a client is by modeling them as real ***dependencies***, which are analyzable by PERTS, by having task *i* dependent on task *i*-1, and task *i*+1 dependent on task *i*. PERTS End-to-End analysis calculates the worst-case Completion Time of a task and assigns this value to the Ready Time of a consequent (in dependency) task. This approach guarantees the serializability and minimizes the idle time of the dependent tasks. For this reason, we choose to use PERTS ***dependencies*** to model client tasks.

However the dependency approach is also pessimistic. The PERTS analyzer assumes that a task may be blocked and preempted by the tasks that depend on it, which introduces a delay in the completion time of the task *i* and delay in the release of the tasks *i+1* and all subsequent tasks.

Client	Period	Ready Time	Relative Deadline	Execution Time	Node	Resource	Intermediate Deadline
Client1	1000	0	1000	500	1	Server1 [100-150] Server2 [200-250]	[0->200] by 300 [200->250]by 700

Table 4.1. Time parameters of the client from Figure 3.3.

We present here a model for the client introduced in the example of Figure 3.3 using the dependencies approach. For simplicity, but without loss of generality, we consider here only one client. The main parameters of the client are presented in the Table 4.1. It is a periodic client with the Period of 1000, Ready Time of 0, Relative Deadline of 1000 and Execution of 500 time units. It is residing on Node 1. It requires use of two resources, server1 and server2, during the periods of time from 100 to 150, and from 200 to 250 units of time, respectively. It also has two intermediate deadlines: the portion of the code from 0 to 200 units of time must be completed by the time of 300 units and the portion of the code from 200 to 250 units of time must be completed by the time of 700 units. For simplicity, we assume in this example that Phasing value is 0 and the client does not have any Non-Preemptable Sections or Optional Intervals, described in Chapter II.

To model this client in the PERTS environment we introduce three tasks Task1, Task2 and Task3, as shown in Table 4.2. Most of the task parameters, such as Period, Ready Time, Phasing, Node, are common for all tasks inherited from the client. The relative deadlines are specified by the intermediate deadlines and relative deadline of the original client. The Execution Time of the task *i* is calculated as the difference between the End Times of the Intermediate Deadlines *i* and *i*–1. The list of required resources is split between the tasks depending on the time intervals specified by the resource requirement and intermediate deadline. Since Server1 needs to be used during the [100-150] time interval, it fits into the time interval specified by the first intermediate deadline ([0-200]), and therefore is assigned to the Task1. The time interval of Server2 is [200-250], which belongs to the time interval of the second intermediate deadline ([200-250]), and therefore belongs to the Task2. Note that the time interval of this resource requirement has been modified to [0-50]. After separation of the first part of the code between 0 and 200 time units into separate task (Task1), the remaining code starts from the 0 time mark. This leads to modification of all time intervals, including resource requirements, Non-Preemptable Sections, Optional Intervals and remaining Intermediate Deadlines, as well as Execution Time, described above. The last column of Table 4.1 shows the dependence of the Task2 on Task1 and Task3 on Task2. Note that if the original client itself depends on some other task, then this dependency is inherited by the first generated task (Task1). If some other task depends on the original client, then this dependency is inherited by the last generated task (Task3).

Task	Period	Ready Time	Relative Deadline	Execution Time	Node	Resource	Depends on
Task1	1000	0	300	200	1	Server1 [100-150]	
Task2	1000	0	700	50	1	Server2 [0- 50]	Task1
Task3	1000	0	1000	250	1		Task2

Table 4.2. Time parameters of the three tasks generated form client from Figure 3.3.

While the original version of PERTS can not analyze clients with intermediate deadlines, it may analyze the system of dependent tasks presented in Table 4.2. Using End-to-End analysis, described in Chapter II, the PERTS engine performs schedulability analysis based on described dependencies. The engine calculates the worst-case completion times for all tasks and modifies the Ready Times and Relative Deadlines of the tasks, as shown in Table 4.3. Here, for simplicity and easy visualization of the process, we assume that there is no blocking and preemption involved in the example and therefore every task completes in "Execution Time" units after its Ready Time.

Task	Period	Ready Time	Relative Deadline	Execution Time	Node	Resource
Task1	1000	0	300	200	1	Server1 [100-150]
Task2	1000	200	500	50	1	Server2 [0- 50]
Task3	1000	250	750	250	1	

Table 4.3. Time parameters of the three tasks generated form client (from Figure 3.3) produced by the End-to-End PERTS analyzer.

4.3 Network Delay

As previously stated, it is necessary to account for network delay in the analysis of distributed real-time systems and PERTS does not directly support it. Note that we are not trying to analyze the network traffic control, but instead we let CORBA developers estimate the worst case Network Delay and input it to the task description. Our goal is to include the Network Delay into consideration in the schedulability analysis. The Network Delay is the worst case time that a task spends traveling from one node to another without holding original and destination CPUs. This specific feature (that original and destination CPUs are not hold by the task) excludes the possibility to use PERTS “acquisition/de-acquisition time” option. The acquisition/de-acquisition time is "charged" against the CPU utilization for the task that is acquiring/de-acquiring. This is too pessimistic since the task is *not* using its local CPU when it is using the network.

Introduction of the Network Delay into schedulability analysis inserts an additional term, *2 * N * delta*, into the time demand function $W_i(t)$ in the Lehoczky schedulability criterion:

$$W_i(t) = \sum_{j=1}^{i-1} C_j \left\lceil t / T_j \right\rceil + C_i + \beta_i + 2 * N * delta$$

Here *N* stands for the number of remote (remote means residing on different than task node) resource requests, generated by a task within one period, *delta* is the worst case time, that request spends traveling from one node to another one. Factor of 2 is attributed to the "round trip" of the resource request.

4.4 Priority Assignment

Task priority assignment will be done using Deadline Monotonic (DM) Mechanism, described in Chapter II.

To make the system analyzable we had to choose a static priority assignment mechanism usable on distributed systems and allowing use of local and global resources. There are two mechanisms, RM and DM, satisfying these conditions. We chose DM since RT CORBA tasks have pre-period deadlines.

4.5 Resource Access Protocol

To control the resource accesses we choose the Distributed Priority Ceiling Protocol (DPCP) [8], which was derived from PCP and extended for the distributed systems, as described in Section 7.2.1. Our choice is based on two main features of the resource access protocol, making system predictable: deadlock free and limited blocking time. These both requirements are satisfied by DPCP.

4.6 Schedulability Analysis

We have chosen the End-to-End Schedulability Analysis Mode, the only Mode that handles task dependencies. In performing a system schedulability analysis, the Analyzer generates a report that describes system parameters, including task priorities and resource priority ceilings.

V. Implementation

This chapter presents the implementation of the new and modified PERTS components. The implementation has been performed in C++ on a Sun Spark5 work station running Sun's Solaris 2.5.

5.1 Implementation Plan

To incorporate the new features, described in the previous chapter, we address three different issues:

1. Modification of the Graphic User Interface (GUI) for the Task Graph Editor to enable input of the Intermediate Deadlines and Network Delay.
2. Implementation of the Client->Task Translator, performing the split of a client into set of dependent tasks, based on the client's intermediate deadlines, described in Section 4.2.2.
3. Modification of the Schedulability Analyzer to account for the Network Delay, described in Section 4.3.

5.1.1 Modification of the Task Graph Editor GUI

To enable an input of the Network Delay value, we have added a box "Network QoS Parameter" to the General Task Data Edit Dialog, located at

"Parameters"/"Task Parameters"/"General Task Data" menu option. It allows specification of the Network QoS Parameter (or Network Delay) for every task along with previously presented in the dialog parameters. It is set to 0 (zero) by default. By clicking on "OK" button the Network QoS Parameter is saved (along with other task parameters). Option "Help" has been slightly modified to incorporate a description of the "Network QoS Parameter".

To enable an input of the Intermediate Deadline parameters, we have introduced a new Intermediate Deadline Edit Dialog. To call this Dialog we introduce a new option "Intermediate Deadlines" to the "Parameters"/"Task Parameters" menu bar. By clicking on this button user pops up the "Intermediate Deadline Edit Dialog" Window, consisting of four fields:

- *Start Time* - specifying the beginning of the portion of the task that needs to complete execution by some intermediate deadline (Start Time is not used in the schedulability analysis in the current project, since the task (client) is split into set of dependent tasks based on End Time. We have input Start Time field for possible future applications);
- *End Time* - specifying the end of the portion of the task that needs to complete execution by some intermediate deadline;
- *Deadline* - the intermediate deadline itself;
- *List* of all previously specified intermediate deadlines.

The Intermediate Deadline Dialog Window contains 6 buttons:

- *Insert* - inserts new set of parameters into the list of Intermediate Deadlines;
- *Delete* - deletes selected (one or more) Intermediate Deadlines from the list;
- *Modify* - modifies the parameters of a specified Intermediate Deadline;
- *OK* - saves current list of Intermediate Deadlines in the increasing End Time order;
- *Cancel* - closes the Intermediate Deadline Dialog Window;
- *Help* - pops up the window describing features of the Intermediate Deadline Dialog Window.

5.1.2 Client->Task Translator Implementation

As discussed in Section 4.2.2, every client, containing at least one Intermediate Deadline, is split into N + 1 tasks (where N is a number of Intermediate Deadlines). The tasks generated from the same client have in common most of the parameters inherited from the client. Here we list the modified parameters and describe the modifications themselves.

- *X coordinate* (a parameter describing task position in a graphical representation of the Task Graph in the Task Graph Editor and End-to-End Schedulability Analyzer) is incremented by 100 for every new task generated from the same client. Thus the task number *i* has *X* coordinate equal to the original client *X* coordinate plus *(i-1)*100*. If the *X* coordinate exceeds a value of 1200 it is set to 50 along with an increase of *Y* coordinate by 50. Under this modification, all the tasks generated from the same client are located on the same horizontal line (have the same *Y* coordinate) in increasing order left to right. If they do not fit on one line being visible in the end-to-end analysis, they are moved to the next horizontal line.

- *Y coordinate* is incremented by 50 units if *X* coordinate has exceeded value of 1200 for the reason described above.
- *Relative Deadline* of the task *i* is set to the Intermediate Deadline number *i* after checking that Intermediate Deadline is stricter (smaller) than the original client Relative Deadline. If not, then the task Relative Deadline is equal to the client's Relative Deadline.
- *Execution Time* of the task *i* is equal to the End Time of the Intermediate Deadline number *i* minus the End Time of the Intermediate Deadline number *i-1*. Reminder: the Intermediate Deadlines are arranged in the End Time increasing order.
- *Name* of the first task of the client keeps the client name. Names of all other tasks have an index attached to the original client name, such as NAME_2, NAME_3, ... NAME_(n+1), where n is the number of intermediate deadlines.
- *Optional Interval List*, *Resource Requirement List* and *Non-Preemptable Section List* of the task *i* include only those intervals (or resource requirements) that fit into the range between the End Time of the Intermediate Deadline number *i* and the End Time of the Intermediate Deadline number *i-1*. If the interval (corresponding to the Optional Interval or Resource Requirement) belongs to the

described range only partially (i.e. Start Time is in that range but End Time is not and vice versa) then the list contains only the corresponding part.

- *List of Intermediate Deadlines* is absent in the new tasks.
- *Identification* (id, PERTS internal unique task characteristic) is assigned after all clients have been split.

According to the client model described in Section 4.2.2, task dependencies are represented by PERTS ***dependencies***. Every Task Graph, as described in Section 2.1, contains a list of ***dependencies***. The Client->Task Translator modifies this list as follows.

If two tasks are generated from the same client and their numbers in sequence are *i* and *i+1*, then the task *i+1* is dependent on task *i* and this dependency is added to the list.

If a *source* of a dependency (a client/task, on which some other client/task depends) was a client, then the source of this dependency is modified and the new source is the last task generated from this client.

If a *destination* of a dependency (a client/task, which depends on some other client/task) was a client, then the destination of this dependency is modified and the new destination is the first task generated from this client.

The PERTS schedulability analyzer calls Client->Tasks Translator before it performs any analysis. By doing so, it guarantees that all intermediate deadlines will be translated into dependent tasks and thus will be taken into account in the analysis.

5.1.3 Modification of the Schedulability Analyzer to incorporate the Network Delay

The modification to allow PERTS to incorporate worst case Network Delay is the first time that we modify the PERTS schedulability analysis engine. It involves two ideologically different parts. The first part is based on the statement, proven in Section 2.3.3, that satisfaction of the schedulability criterion based on the Liu-Layland's condition guarantees satisfaction of the criterion based on Lehoczky's condition. Based on this, we conclude that PERTS engine should check the latter criterion only. We eliminate Liu-Layland's criterion from the schedulability analysis. In the second part, we modify the schedulability criterion based on the Lehoczky's condition to incorporate the total network delay, as described in Section 4.3.

5.2 Graphic User Interface (GUI) Modifications Implementation

As was pointed out in Section 5.1.1, the existing GUI had to be modified to accommodate the new parameters of tasks (clients), Network QoS Parameter and Intermediate Deadlines.

The implementation of the semantics of the set of Intermediate Deadline parameters was encapsulated in a base C++ class called *IntermediateDeadline*.

```
class IntermediateDeadline : public PObject {
public:
```

```
PTime        start, end;
PTime        iDeadline;
IntermediateDeadline(PTime from, PTime to, PTime by)
                                              {start = from; end = to; iDeadline = by}
 ~IntermediateDeadline(){};
      };
```

Three members of the class, *start*, *end* and *iDeadline*, specify the Start and End Times of the Intermediate Deadline, and the Intermediate Deadline, respectively. Since each task may have multiple Intermediate Deadlines we used a linked list containing pointers to the objects of class *IntermediateDeadline*. Instead of implementing a new class supporting this linked list and the basic operations, such as *Delete_Element*, *Insert_Element*, *First*, *Last*, *Next*, *Previous* and others, we have used the class *List* (previously implemented in PERTS), supporting all listed features. The class *List* contains a linked list of pointers to the objects of general class *PObject*. To enable use of the class *List*, our class *IntermediateDeadline* is inherited from general class *PObject*.

The complete description of the task parameters is encapsulated in the class *Task*. We have introduced two new members into this class:

```
List      *Intermediate_List,
PTime      networkDelay.
```

The first member, *Intermediate_List*, is a pointer to an object of class *List*, containing the list of the pointers to the objects of class *IntermediateDeadline*. The second member, *networkDelay*, contains a value of the Network QoS Parameter.

Along with two new members of the class *Task* we have introduced the method:

void DefineIntermediateList(PTime start, PTime end, PTime deadline).

It inserts a new set of Intermediate Deadline parameters into List of Intermediate Deadlines, sorting it in the increasing End Time order. If two or more sets of Intermediate Deadline parameters have the same End Time, then the list keeps only one of those sets with the earliest Intermediate Deadline.

To maintain all existing GUI operations after introduction of the new parameters, we have introduced some modifications/additions to methods of different classes, which we briefly describe here.

To save (Save/ Save As) the new Network Delay and list of Intermediate Deadlines parameters into a textual file containing Task Graph parameters, we have modified the format of this file, changing the save method (in gtaskgraph.cc)

static void write_task(ofstream &, GTask *, int).

To maintain the Open/Reopen operation, we have modified the PERTS Compiler to read the new fields from the textual file. We have modified (in tg_compile.cc) method

int TG_Compiler::TaskDataItem(Task * t),

and introduced a new method

int TG_Compiler::Intermediate_List(Task * t).

To incorporate the new parameters into the report (Generate Report) we have modified (in gtaskgraph.cc) static void generate_report_task(Widget, void *, void *).

To copy new (along with original) task parameters (Copy Task Parameters) we have modified (in gtask.cc) method

void Task::CopyTaskParameters(Task *src).

The appropriate description of the new parameters has been added to the help.h and help.cc files.

5.3 Client->Tasks Translator Implementation

To implement the Client->Task Translator described in Section 5.1.2, we have introduced the set of new methods described below. The first method

TaskGraph::Translate()

scans through the list of tasks of the Task Graph, searching for the clients with Intermediate Deadlines. As soon as an Intermediate Deadline is found, the Translate() method

- creates a new task;
- inserts a new task into the task list before the client under consideration;
- inserts a new dependency into dependency list and modifies the existing dependencies, as described in Section 5.1.2;
- adjusts the parameters of the new task using method:

Task::Modify_new_task(Task* cur_task, PTime cur_start, PTime cur_end, PTime cur_deadline);

- adjusts the parameters of the client using method:

Task::Modify_Task (PTime cur_end, int counter).

These two methods are similar in implementation. The first method copies all parameters from the original client and then deletes all events coming after the End Time of the Intermediate Deadline. The second method deletes all events before the End Time of the Intermediate Deadline under consideration. These methods perform the parameter adjustments using the following methods.

Task::Modify_optionalIntervalList(PTime cur_end, int i)

and

Task::Modify_NPSList(PTime cur_end, int i)

which delete all intervals after the End Time of the Intermediate Deadline for the new task and before it - for the client.

The method

Task::Modify_resRequirementList(PTime cur_end, int i)

deletes all Resource Requirements after the End Time of the Intermediate Deadline for the new task and before it for the client.

The method

Task::Modify_IntermediateList(PTime cur_end)

deletes the first Intermediate Deadline in the client's list of Intermediate Deadlines.

The method

Task::Modify_name(int counter)

increments the index attached to the client name (to indicate sequence of the tasks generated from the same client).

Taking into consideration the new amount of work to be performed by the client all these methods modify the appropriate client parameters.

After all clients have been split into tasks the method

```
TaskGraph::Reorder()
```

is called to reassign task id's to all tasks.

5.4 Schedulability Analyzer Modifications Implementation

To include the Network Delay into PERTS analysis, we have incorporated into class SA_Task (sa_task.h) a new member

```
_total_Network_Delay
```

containing a value of the Total Network Delay experienced by a task during one period, *2 * N * delta*, as described in Section 4.3. We also introduce new member methods

```
Ptime    SA_Task::network_Delay ()
Ptime    SA_Task::total_Network_Delay()
void    SA_Task::set_Total_Network_Delay ()
```

returning the Network Delay, Total Network Delay of the task and assigning value of the Total Network Delay, respectively.

We have modified the

```
PTime PCP_Node::worst_Blocking_Time(SA_Task *inTask)
```

method (in pcp_node.cc), which besides blocking time also calculates and sets the value of the Total Network Delay.

To incorporate the Total Network Delay into calculation of the Lehoczky's demand method we have modified method

```
BOOLEAN SA_Node::time_Demand_Test(SA_Task *inTask)
```

reporting system schedulability.

To eliminate Liu-Layland's condition from schedulability analysis, we have excluded the call for the Liu-Layland's test and test of the system schedulability before calling the Lehoczky's test (Note: the original code performed the Liu-Layland's analysis and called the Lehoczky's test only if previous result was "non-schedulable").

To include task total network delay into calculations of its completion time we have modified method

```
void SA_Node::build_Time_Demand_Line(SA_Task *inTask, int instance)
```

in sa_node.cc.

We have modified the format of the reports in all three regimes of the Schedulability Analyzer introducing Network QoS Parameter and Total Network Delay.

VI. Evaluation

After the implementation of the modifications to the GUI for the Task Graph Editor, the implementation of the Client->Task Translator, and the modification of the Schedulability Analyzer were completed, several tests were done to demonstrate the implementation correctness.

6.1 Task Graph Editor GUI Tests

Tests of the modified Task Graph Editor GUI have demonstrated presence and correctness of all desired features.

The General Task Data Edit Window contains a new box "Network QoS Parameter". The introduction of the new field did not affect others. A user can specify the Network QoS Parameter (along with previously presented in the dialog parameters). If the Network QoS Parameter is not specified, it contains a 0 (zero) value. By clicking on "OK" button the Network QoS Parameter is saved (along with other task parameters). It was tested by opening General Task Data Edit Window of a task, containing a non-zero value of the Network QoS Parameter assigned in advance. Option "Help" contains a description of the "Network QoS Parameter".

We have tested the "Intermediate Deadline Edit Dialog", which enables an edit of the Intermediate Deadline parameters. The "Intermediate Deadline Edit Dialog" properly inserts a new set of parameters into the list of Intermediate Deadlines, deletes selected (one or more) Intermediate Deadlines from the list, modifies the parameters of a specified Intermediate Deadline, saves the current list of Intermediate Deadlines in the increasing End Time order, closes the Intermediate Deadline Dialog Window and pops up the Help window, describing features of the Intermediate Deadline Dialog Window.

Using a task with known values of the Network QoS Parameter and Intermediate Deadlines parameters, we have performed the “Copy Task Parameters” operation to assure its correctness.

Generating the report we have confirmed presence of the new parameters and the correct fit into report format.

By saving a Task Graph containing clients with some Intermediate Deadlines and Network QoS Parameter, we have tested the modified “Save” function. We have checked the textual file generated by “Save” function for presence of the new parameters. As a continuation of this test and as a test for “Open” operation, we have opened previously saved file with Intermediate Deadlines and Network QoS Parameter. Using the appropriate dialogs in the Task Graph Editor Window we have checked previously saved parameters and confirmed “Save” and “Open” operations.

6.2 Client->Task Translator Tests

To perform the Client->Tasks Translator tests we have built various Task Graphs, containing clients with and without intermediate deadlines. Using the Translator we have performed the translation of the original Task Graphs (with clients, containing Intermediate Deadlines). The ability of Schedulability Analyzer to save modified Task Graphs allowed us to save the translated Task Graph as a textual file of standard format. Comparing the translated Task Graph with theoretical expectations confirmed Translator correctness.

Intermediate Deadlines were chosen to test all possible situations, including such non-trivial as “Intermediate Deadline is less strict than the original client Relative Deadline” or “Intermediate Deadline End Time is later than client entire Execution Time”. In the tests we have adjusted parameters of Optional Intervals (or Resource Requirements) so that the Intervals belong completely or partially to a new task generated from a client. One of the tests, in which we have tried to combine all mentioned aspects, is presented in Appendix A.

The task parameters, specifying geometrical location of the tasks on screen in End-to-End Analysis and in Task Graph Editor, have been tested visually. We have introduced large number of intermediate deadlines to “saturate” the screen with tasks (generated from the same client) not fitting on one line. The Translator has performed a correct assignment of these parameters by moving exceeding tasks to the next line. We also have checked visually the presence of the dependencies between generated tasks and the original clients.

6.3 Schedulability Analyzer Tests

The testing of accounting for the Network Delay in the Schedulability Analyzer consisted of two different parts. In the first part we confirmed that schedulability report depends on Lehoczky's (not Liu-Layland's) criterion, and that Single-Node Analysis takes into account the Network Delay.

To do so, we have built a Task Graph containing a task, which barely meets its deadline (introduction of one more unit of time delay makes the task non-schedulable). At that stage, the Network QoS Parameter of the task was set to zero. After running the Schedulability Analyzer and confirming that the system is schedulable, we have set the Network Delay parameter of the described task to one. All other parameters were kept unchanged. Running the Schedulability Analyzer we have confirmed that the system is not schedulable. These tests have proven that the Network Delay is taken into account.

The original Schedulability Analyzer performed tests of the Liu-Layland's condition and, only if it has not been satisfied, it then checked Lehoczky's criterion. Since we have modified code for the Lehoczky's criterion only, the report of the system non-schedulability in our test proves that schedulability report is based on the Lehoczky's criterion only.

The second part of the test aimed to confirm correctness of the End-to-End Schedulability Analysis that we modified to incorporate Network Delay. The purpose of this part was to demonstrate that the Network Delay is taken into account in the calculation of the Completion Time of a task from the path of dependent tasks. This value is crucial for assignment of the Ready Time for the successor in that path. We built a Task Graph that contains a path of dependent tasks with Network QoS Parameters set to zero. Running the End-to-End Schedulability Analyzer, we obtained and saved Ready Times of all tasks in the Task Graph. Then we set some non-zero values to the Network QoS Parameters of all tasks in the original Task Graph. Running the End-to-End Schedulability Analyzer, we confirmed that Ready Times of each task in the path had been shifted (relative to the value from the previous run) by the sum of all Total Network Delays of the predecessors in the path. This proves the correctness of End-to-End Schedulability Analysis incorporating Network Delay.

In the tests we also have checked the presence of the Network QoS Parameter and Total Network Delay values in the generated report.

VII. Limitations and Future Work

This project has been a first and necessary step towards creating analysis theory and tools for distributed real-time systems such as RT CORBA–based applications. However, further steps are necessary before PERTS can fully analyze RT CORBA systems. In this chapter we describe the further necessary steps.

7.1 Effects of Limited Priorities

RT CORBA implemented on commercial real-time operating systems (RTOS) may face the problem of the RTOS providing fewer priorities than the RT CORBA system requires, particularly under DPCP which has its own range of (very high) priorities for tasks executing *global critical sections* (see Section 7.2.1). For instance, a Solaris RTOS provides only 60 local RT priorities, while the RT CORBA system may need more than 60 priorities assigned. We suggest to modify the PERTS Resource Graph Editor interface to allow specification of the number of priorities on each node.

To further define the problem, let a node have N tasks (representing parts of CORBA clients) C global critical sections (executing on CORBA servers) under the DPCP protocol, and P local priorities. If $N+C \leq P$, then there is no problem - an enforceable priority assignment can be done and PERTS can analyze it. However, if

$N+C > P$, then either the resulting priority assignment is not enforceable, or several entities will need to be at the same priority. PERTS can not currently handle this situation.

We present here one possibility to assign available priorities and analyze a schedulability in the described situation. We split P available priorities into two partitions, that are proportional to N and C. The first partition, P_{task}, is serving tasks, and the second partition, P_{gcs}, is serving global critical sections on the particular node. Then we assign $P_{task}–1$ highest priorities to the $P_{task}–1$ highest priority tasks and $P_{gcs}–1$ highest priorities to the $P_{gcs}–1$ highest priority gcs's. Then we assign the lowest task priority to the remaining $N\text{-}(P_{task}–1)$ tasks and the lowest gcs priority to the remaining $C\text{-}(P_{gcs}–1)$ gcs's. This algorithm may not guarantee the best schedulability results (other combinations of split of available priorities and their distributions among the tasks and gcs's are possible). We have chosen this algorithm to simplify the illustration of the problem.

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. We assume FIFO scheduling of tasks with the same local priority and make the worst case assumption that each task or global critical section falls at the end of the FIFO queue for its priority. The demand function should be modified as follows

$$W_i(t) = \sum_{\substack{j=Alltasksof\\higherpriority}} C_j \lceil t / T_j \rceil + \sum_{\substack{k=Alltasksof\\thesamepriority}} (C_K * M_k) + C_i + B_i \leq t$$

Here C_l represents an Execution Time of the task T_l, B_i - blocking time of task T_i. 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 global critical sections executed by task T_i. The origin of this factor is in the fact that the task T_i may be waiting for an end of the same priority task execution. It may be waiting once, when the task T_i is initialized, and every time when it releases its CPU for an execution of the remote global critical section, 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 frequency of the same priority task

$$\left\lceil {}^{t}\!/\!{}_{T_k} \right\rceil$$

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

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

Before a global critical section could be blocked for a duration of the longest lower priority global critical section, b_g. Now along with this blocking it may be blocked by the duration of all global critical sections, *CS*, of the same priority.

The described modifications should be incorporated into the PERTS Schedulability Analyzer engine.

7.2 On possibility of Using DASPCP

We speculate that URI's Distributed Affected Set Priority Ceiling Protocol (DASPCP), which has been shown to improve concurrency in object oriented systems [13], can be incorporated into PERTS analysis. The DASPCP is a relatively new resource access protocol developed at URI particularly for RT object-oriented software [13]. It incorporates two protocols: DPCP [8] and ASPCP [14]. In Section 7.2.1 we prove its deadlock free property and tight Priority Inversion bound. The main idea behind DASPCP is to consider particular methods of a CORBA server (not the entire server) as a PERTS ***resources*** and assign each method its own priority ceiling based on other methods of the server with which it conflicts. In Section 7.2.1 we describe the DPCP and DASPCP and show that latter increases concurrency.

Introduction of the DASPCP slightly modifies the mapping of the RT CORBA to PERTS, described in Chapter IV. Namely, PERTS ***resources*** do not represent whole CORBA servers, instead they represent the methods of the servers. The necessity to specify a set of conflicting methods [13, 14] (***resources***) for each method would need to be added to the PERTS Resource Graph Editor interface and the revision of the calculation of the priority ceiling in PERTS Schedulability Analyzer.

7.2.1 DPCP versus DASPCP

In this section we describe the Distributed Priority Ceiling Protocol (DPCP) and Distributed Affected Set Priority Ceiling Protocol (DASPCP). We compare the concurrency under these protocols and prove that DASPCP is deadlock free and has a limited blocking time.

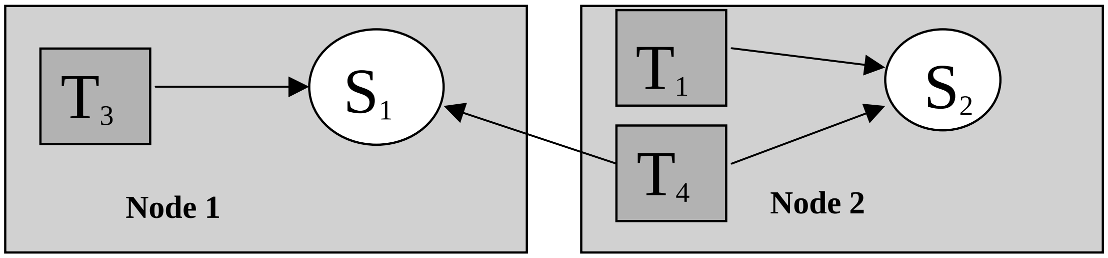
The DPCP [8] handles a synchronization of task method calls, executing on distributed systems. Before we start the description of the protocol we must introduce the following definitions:

- A semaphore that is accessed by tasks allocated to different processors (a single processor) is referred to as a *global (local) semaphore*.
- A critical section guarded by a global (local) semaphore is referred to as a *global (local) critical section*, *gcs (lcs).*

First, all tasks must be bound to processors. A task *T* executes its non-critical-section code and lcs's on its host processor, while its gcs's may be bound and executed on a processor(s) different than the *T*'s host processor. All gcs's controlled by the same semaphore S_G, and the semaphore S_G itself, are bound to the same synchronization processor. A gcs, generated by task *T*, is assigned a priority equal to the sum of the base priority ceiling P_G (a fixed priority, higher than the priority assigned to the highest priority task in the system) and *P*, the priority of task *T*. Each processor runs the priority ceiling protocol on the gcs's (considering each thread of execution for executing a gcs as a "task"), the set of application tasks (if any), and the set of global and local semaphores bound to the processor. DPCP prohibits a mixed nesting of lcs's and gcs's.

The following example is not an exhaustive demonstration of possible situations (of blocking, preemption etc.), that may occur under DPCP. Our goal is a simplest possible example, demonstrating benefits of DASPCP relative to DPCP. For more detailed example of application of DPCP we refer reader to the original Rajkumar's work [8].

Example 7.1. Consider a distributed system with 2 nodes. The application consists of 3 tasks and 2 databases (O_{track1} and O_{track2}), guarded by 2 semaphores (S_1 and S_2). Task T_3 is bound to the *Node 1*, while tasks T_1 and T_4 are bound to the *Node 2*.



Tasks T_1, T_3 and T_4 execute the following sequence of steps.

T_1: ...O_track2.read_speed...

T_3: ...O_track1.write_speed...

T_4: ...O_track1.read_altitude...O_track2.read_depth

Note: in our system the priority of task T_i, $p(T_i)$, is assumed to be lower than that of T_{i+1}.

The semaphores S_1 and S_2 are bound to the *Nodes 1* and *2* respectively. The priority ceilings of each semaphore, and the normal execution priority of each critical section thread are listed in Tables 7.1 and 7.2, respectively.

Priority Ceiling Of Semaphores	
Semaphore	Priority Ceiling
S_1 (Global)	$p(T_4) + P_G$
S_2 (Local)	$p(T_4)$

Table 7.1. The priority ceilings of Semaphores in Example 7.1 under DPCP.

Normal Execution Priorities of Critical Sections		
Task	Critical Section Guarded by	Execution Priority
T_1	S_2	$p(T_1)$
T_3	S_1	$p(T_3) + P_G$
T_4	S_1 S_2	$p(T_4) + P_G$ $p(T_4)$

Table 7.2. The Normal Execution Priority of Critical Sections in Example 7.1 under DPCP.

The following example demonstrates the sequence of events in the system under DPCP, presented graphically in Figure 7.1:

- At time t_0, task T_1 arrives on *Node 2* and begins its execution. Similarly, task T_3 begins execution on *Node 1*.
- At time t_1, task T_1 locks the local semaphore S_2 on *Node 2* and begins execution of lcs at its normal execution priority of $p(T_1)$. Task T_3 locks the global semaphore S_1 on *Node 1* and begins execution of gcs at its normal execution priority of $p(T_3) + P_G$.
- At time t_2, task T_4 arrives on *Node 2* and preempts T_1. Task T_3 continues its execution of gcs on *Node 2*.
- At time t_3, task T_4 requests a lock on global semaphore S_1. However, the semaphore is currently locked by a lower priority gcs, $p(T_3) + P_G$. Hence T_4 is blocked and T_3 continues its gcs execution at the inherited priority of $p(T_4) + P_G$. Task T_1 resumes its execution of lcs at *Node 2*.
- At time t_4, task T_3 completes the execution of its gcs and releases the lock on global semaphore S_1 and resumes its own priority. Task T_4 locks the global semaphore S_1 on *Node 1* and begins execution of gcs at its normal execution priority of $p(T_4) + P_G$. Task T_3 is preempted by higher priority T_4's gcs. Task T_1 continues the execution of its lcs at *Node 2*.
- At time t_5, task T_4 completes the execution of its gcs and releases lock on global semaphore S_1. T_3 resumes its execution on *Node 1*. T_4 attempts to get a lock on semaphore S_2. However, the semaphore is currently locked by a lower priority

task T_1. Hence T_4 is blocked and T_1 continues its execution with inherited priority of $p(T_4)$.

- At time t_6, task T_1 completes the execution of its lcs and releases the lock on semaphore S_2 and resumes its own assigned priority. Task T_4 locks the local semaphore S_2 on *Node 2* and begins its execution.
- On completion of execution of T_4 at t_9, task T_1 resumes its execution. T_1 and T_3 complete their executions at some later times.

To increase the concurrency of the task method calls in a distributed system we propose to incorporate DPCP with ASPCP [14] into DASPCP. The DASPCP copies all characteristics of the DPCP except the resource access control protocol at a processor level. While under DPCP each processor runs PCP on the gcs's, the set of application tasks, and the set of global and local semaphores bound to the processor, the DASPCP uses ASPCP.

The following example illustrates the application of DASPCP and demonstrates an increased concurrency compared to application of DPCP. Here we consider the system of tasks identical to one described in Example 7.1. Also we have the same databases, but instead of associating a semaphore with each database we provide one semaphore per each method of a database [13, 14].

The priority ceilings of each semaphore, and the normal execution priority of each critical section thread are listed in Tables 7.3 and 7.4 respectively.

Priority Ceiling Of Semaphores	
Semaphore	Priority Ceiling
$S_{1\ write\_speed}$ (Local)	$p(T_3)$
$S_{1\ read\_altitude}$ (Global)	*0*
$S_{2\ read\_speed}$ (Local)	*0*
$S_{2\ read\_depth}$ (Local)	*0*

Table 7.3. The priority ceilings of Semaphores in Example 7.1 under DASPCP.

Normal Execution Priorities of Critical Sections		
Task	Critical Section Guarded by	Execution Priority
T_1	$S_{2\ read\_speed}$	$p(T_1)$
T_3	$S_{1\ write\_speed}$	$p(T_3)$
T_4	$S_{1\ read\_altitude}$ $S_{2\ read\_depth}$	$p(T_4) + P_G$ $p(T_4)$

Table 7.4. The Normal Execution Priority of Critical Sections in Example 7.1 under DASPCP.

Following example demonstrates the sequence of events in our system under DASPCP, illustrated in Figure 7.1:

- At time t_0, task T_1 arrives on *Node 2* and begins its execution. Similarly, task T_3 begins execution on *Node 1*.
- At time t_1, task T_1 locks the local semaphore $S_{2\ read\_speed}$ on *Node 2* and begins execution of lcs at its normal execution priority of $p(T_1)$. Task T_3 locks the local semaphore S_{1write_speed} on *Node 1* and begins execution of lcs at its normal execution priority of $p(T_3)$.
- At time t_2, task T_4 arrives on *Node 2* and preempts T_1. Task T_3 continues its execution of lcs.

- At time t_3, task T_4 requests a lock on global semaphore $S_{1\ read\_altitude}$. Since its gcs's priority, $p(T_4) + P_G$, is higher than the priority ceiling of $S_{1\ write\_speed}$, $p(T_3)$, it gets lock on $S_{1\ read\_altitude}$ and preempts T_3's lcs. Task T_1 continues the execution of its lcs at *Node 2*.
- At time t_4, task T_4 completes the execution of its gcs and releases the lock on global semaphore $S_{1\ read\_altitude}$. Task T_3 resumes the execution of its lcs at $S_{1\ write\_speed}$. Task T_4 requests a lock on local semaphore $S_{2\ read\_depth}$. Since its priority, $p(T_4)$, is higher than the priority ceiling of $S_{2\ read\_speed}$, 0, it gets lock on $S_{2\ read\_depth}$ and preempts T_1.
- At time t_5, task T_3 completes the execution of its lcs. No changes on *Node 2.*
- At time t_7, task T_4 completes its execution, as well as the execution of its lcs on S_{2read_depth} and releases the lock. T_1 resumes its execution of lcs on $S_{2\ read\_speed}$ on *Node 2*. T_1 and T_3 complete their executions at some later times.

The main advantage of the DASPCP compared to the DPCP may be seen in Figure 7.1 and two considered sequences of events: under DASPCP there were no blocking, while running it under DPCP, T_4 was blocked twice, ones at global and ones at local resource.

To conclude the discussion of the DASPCP we state and proof its main properties, using Rajkumar's approach [8].

1. Under DASPCP deadlocks are avoided.

Proof: A task can deadlock with other tasks only, since, by assumption, it cannot deadlock with itself. Since, by assumption, nesting of gcs's and lcs's is prohibited, access to gcs's and lcs's cannot occur within the same critical section. Since each global and local semaphore is accessed only by a single processor, deadlocks can't occur across processor boundaries. The only possibility, we have not considered yet, is a deadlock within a processor. The ASPCP used on each processor excludes this, last, possibility of deadlock [13,14]. Therefore, under DASPCP deadlocks are avoided.

2. Maximum blocking time is finite under DASPCP.

Proof: There are 3 possible types of blocking. The limitation of blocking time in each type guarantees the finite total blocking time.

- Satement1: a task *T* can be blocked for the duration of at most n^G+1 local critical sections of lower priority tasks bound to the same processor as *T*. Here n^G is a number of gcs's executed by *T* at remote processors during on period.

  Proof: Task *T* can be considered to be suspending itself n^G times during one period, when it attempts to execute gcs at remote processors. Every time, when task *T* tries to resume its execution after suspension, it may be blocked on local resource by a lower priority task. It may happen once more, when task *T* arrives on a processor. Under the ASPCP the blocking time is limited by a longest critical section of a low priority task. Statement1 follows from here.

- Statement2: for every outermost gcs that *T* enters at remote processor, the task *T* can be blocked for the duration of one longest global critical section of a lower priority tasks, executing their gcs's at the same remote processor.

Proof: This statements follows from the consideration of the gcs's as a tasks at the remote processor accessing resource on the same processor, and the fact that under ASPCP the blocking time is limited by a longest critical section of a low priority task.

- Statement3: a task *T* can be preempted by any task T_i residing at the remote node and accessing *T*'s host node, as well as by higher priority tasks T_i executing their gcs's at the same remote nodes as used by *T*'s gcs's., for a finite amount of time.

  Proof: The execution times of gcs's of tasks T_i are finite quantities. Number of tasks is also finite. Periods of tasks T_i and *T* are also finite, and therefore there may not be an infinite repetition of a task T_i during one period of *T*. Statement3 is proven.

Since all three types of blocking are finite under DASPCP the total one is also finite.

3. Introduction of DASPCP never can decrease concurrency of the system in comparison with DPCP.

Proof: Replacement of PCP by ASPCP at a processor level may never increase the priority ceiling of any semaphore (it may introduce additional semaphores controlling particular methods, but their priority ceilings may not be higher than a priority ceiling of the original semaphore controlling the entire database). Therefore under DASPCP the blocking time may never increase and concurrency may never decrease.

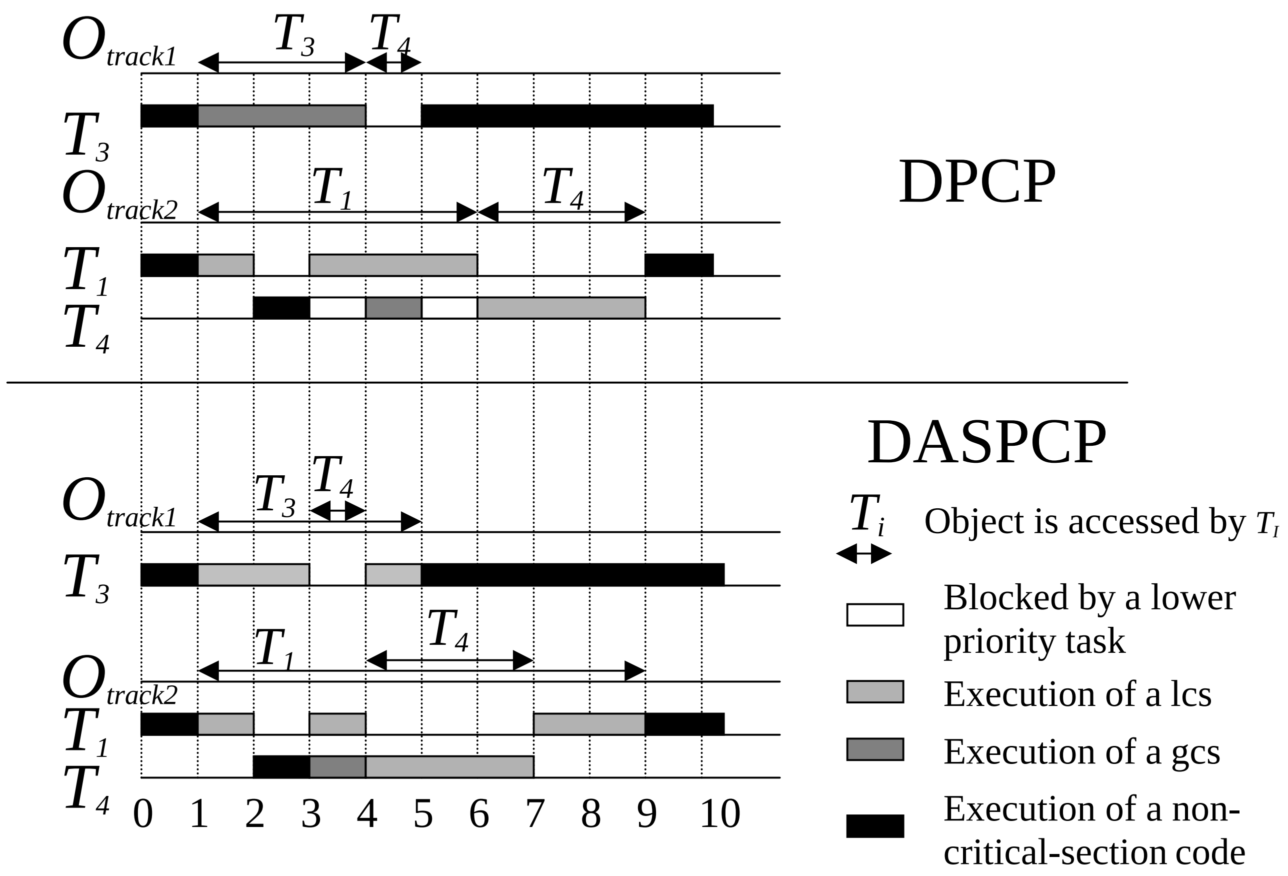


Figure 7.1. Time diagram for the task system, described in Example 7.1 under DPCP (top) and DASPCP (bottom).

7.3 Possible Improvements of the Schedulability Analyzer

We consider here three different aspects that will improve the schedulability analysis.

The first issue addresses the problem of schedulability analysis of the Harmonic Tasks. In Section 7.3.1 we describe this problem and suggest the modification of the Lehoczky's schedulability criterion. In Section 7.3.2 we discuss a possible way to make the End-to-End schedulability analysis less pessimistic.

Finally, we describe a need to eliminate the detected mistakes in the schedulability theory and PERTS implementation for systems under DPCP. These problems are summarized in Section 7.3.3 along with suggested ways to correct them.

7.3.1 Necessary Schedulability Criterion for the task systems with harmonic tasks

An existence of the harmonic tasks in a task system requires modifications to the original Lehoczky's necessary and sufficient schedulability criterion. In this section we present the original Lehoczky's condition. We demonstrate a counter-example, demonstrating that the original criterion is not necessary. Finally, we present necessary modifications to the criterion to guarantee its necessity.

The original Lehoczky's criterion states following [10]:

If for every *i-th* task in a system of *n* tasks that do not use shared resources and do not contain Non-Preemptable Sections there exists a value of *t* such, that

$$0 < t \le d_i$$

and

$$W_i(t) \le t$$

then the system is schedulable.

Here

$$W_i(t) = \sum_{j=1}^{i} C_j \left\lceil t/T_j \right\rceil$$

d_i is a relative deadline of the task *i*, C_j and T_j are the Execution Time and Period of the task *j*. The tasks are numbered in the decreasing priority order (the first task has the highest priority). This criterion indeed is necessary and sufficient in case of worst-case phasing, when all tasks are ready to start their executions at the same time.

We have found that the worst-case phasing may never occur in the set of harmonic tasks with different Release Times (in terms of the task timing parameters, introduced in Chapter II, task Release Time is a sum of its Phase and Ready Time). Here we present a counter-example demonstrating that the Lehoczky's criterion is not necessary in the described situation.

Example 7.2. Let us consider two tasks, Task1 and Task2, residing on a single node system without shared resources. We present the tasks timing parameters in Table 7.5.

	Task1	Task2
Relative Deadline	1	3
Release Time	3	0
Execution Time	1	3
Period	5	10

Table 7.5. Timing parameters of the tasks from the Example 7.2.

Using RM priority assignment Task1 has a higher priority. Following the Lehoczky's schedulability criterion we obtain that

$$W_1(1) = C_1 = 1 = d_1$$

and

$$W_2(t) = \left\lceil t/T_1 \right\rceil C_1 + C_2 \geq C_1 + C_2 = 1 + 3 \geq 3 = d_2$$

Under this criterion the Task1 is schedulable since at *t=1* function *$W_1(t)=1$* which is equal to the deadline *$d_1=1$*. Task2 is not schedulable since the function *$W_i(t)$* is monotonically increasing and its smallest value of *1+3=4* is greater than its deadline *$d_2=3$*.

Analysis of the time diagram, shown in Figure 7.2, demonstrates that this system is schedulable.

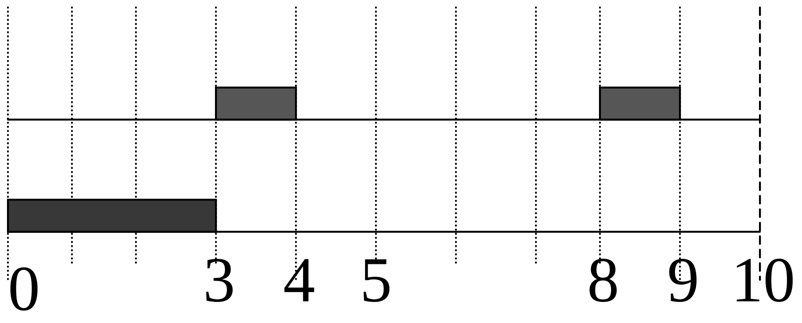


Figure 7.2. Time diagram of the task system described in Example 7.2. The top line corresponds to the Task1, the bottom one - to the Task2.

At time $t=0$ Task2 is the only one ready to start execution. It gets the CPU, runs until $t=3$, completes its execution and meets the deadline $d_2=3$. At the time $t=3$ Task1 gets the CPU, runs until $t=4$, completes its execution and meets its timing constraint $d_1=1$. It also runs between $t=7$ and $t=8$ and meets its deadline. Figure 7.2 represents one cycle (period) of the Task2. Since the two tasks are harmonic and their Release Times are separated by the same time period in every cycle, the time diagram will be repeated as time goes on.

The original Lehoczky's condition would yield correct result if the tasks have the same Release Time (indeed, in this case the Task2 would miss its deadline). The crucial point in the considered example is that two tasks never interfere with each other, which may happen only in case of harmonic tasks with different Release Times.

Concluding that Lehoczky's condition is sufficient but not necessary, we present here our modifications to correct this drawback. We modify the calculation of the time demand function $W_i(t)$. Note that the original description of the function $W_i(t)$ includes the execution times of all higher than the task i priority tasks. In our approach we separate these tasks into two different sets: the set of tasks that are harmonic with task i and the set of tasks that are non-harmonic with i. To exclude the execution time of the harmonic tasks that never "interfere" with task i, we introduce a new function L_k that characterizes the time period between the Release Time of the task i and harmonic task k. This function is formally defined as

$L_k = Release\_Time_i - Release\_Time_k - N * T_k$

where N is a maximum whole number guaranteeing positive L_k value.

Lehoczky's demand function after we exclude the execution of the non-interfering tasks has the following form:

$$W_i(t) = \sum_j C_j \left\lceil t/T_j \right\rceil + \sum_k C_k \left\lceil \frac{t - T_k + L_k}{T_k} \right\rceil + C_i$$

where k and j represent the set of higher priority harmonic and non-harmonic tasks, respectively.

This improved criterion is necessary and sufficient in two extreme cases: the case of worst-case phasing and the case of "non-interfering" harmonic tasks. However, it is not valid in case of "partial harmonic task interference". This case occurs when a task *i* is released after the harmonic task *k* has completed some, but not all, of its execution. Under these circumstances, the task *i* is not blocked by entire execution time of task *k* (C_k), but only by the part of task *k* that has not been completed by the Release Time of task *i*. To incorporate this case into Lehoczky's schedulability criterion, we introduce the function M_k that represents the time period necessary to complete the execution of task *k* after task *i* has been released. Function M_k is formally defined as:

$M_k = 0$ *if* $W_k(t) \leq t$, *for* $0 \leq t \leq L_k$

$M_k = W_k(t) - L_k$ *otherwise.*

The first condition corresponds to the case when harmonic task *k* completes its execution before task *i* is released and therefore $M_k = 0$, while the second one calculates the time period necessary to complete the execution of task *k* after task *i* has been released.

The final form of the Lehoczky's demand function, after we exclude the execution of the non-interfering tasks and incorporate the execution of the "partially interfering tasks", has the following form:

$$W_i(t) = \sum_j C_j \left\lceil t/T_j \right\rceil + \sum_k \left[C_k \left\lceil \frac{t - T_k + L_k}{T_k} \right\rceil + M_k \right] + C_i$$

This final form of the time demand function corresponds to the necessary and sufficient schedulability condition. It is necessary only for the task systems that do not use shared resources.

To analyze the task systems that use shared resources, one need to add the blocking time of task *i* to the time demand function. Since schedulability theory does not calculate the exact blocking time, but only its upper bound, no schedulability criterion for the systems with shared resources may be necessary, but sufficient only.

7.3.2. Modification of the End-to-End Schedulability Analysis

Unfortunately, the schedulability analysis assumption of worst-case phasing drastically affects the End-to-End analysis, since in a path of the dependent harmonic tasks all their Ready Times are different. The schedulability criterion for harmonic

tasks developed in Section 7.3.1 does not support analysis of systems with task dependencies. The reason for this is that in the End-to-End analysis, a task Ready Time is modified, while our schedulability criterion strongly depends on its value. This issue is important for the analysis of the RT CORBA. Since every client with Intermediate Deadlines is modeled as a path of dependent tasks, these tasks never interfere (preempt or block each other). A new schedulability criterion needs to be developed to exclude the described problem with the analysis of the dependent tasks. We propose that, until this criterion is developed, the PERTS End-to-End analysis use the following modification to the Lehoczky's criterion to make it less pessimistic:
When calculating the demand function $W_i(t)$ of a task i, do not include Execution Times and blocking due to the tasks from the same path (regardless of their priority) if they have hard deadlines, the same Periods, the same Phases and for each such task the sum of its Ready Time and Relative Deadline is not greater than its Period. These conditions guarantee that next cycle of the path execution never starts until the previous is completed. Furthermore, a successor task never starts its execution before its predecessor completes. Thus, tasks of the same path that satisfy these conditions never interfere. The described modification will make the analysis of the RT CORBA systems less pessimistic because tasks that are generated from the CORBA clients always satisfy the described conditions.

The proposed schedulability criterion modification is not the final solution for the analysis of the dependent tasks, but it does allow making less pessimistic analysis of RT CORBA systems.

7.3.3 Concerns with PERTS Analysis of DPCP

We have identified some concerns with the current PERTS analysis of systems under DPCP. We consider two categories of concerns: dangerous (when PERTS reports task sets schedulable while they are not schedulable) and pessimistic (when PERTS reports tasks sets non-schedulable when they are actually schedulable). For each concern, we show an illustrative example complete with full PERTS parameters (that we have run through PERTS), an explanation as to why the improper behavior occurred, and in most cases suggested solutions. All concerns are numbered (D1-D3 for dangerous and P1-P6 for pessimistic concerns).

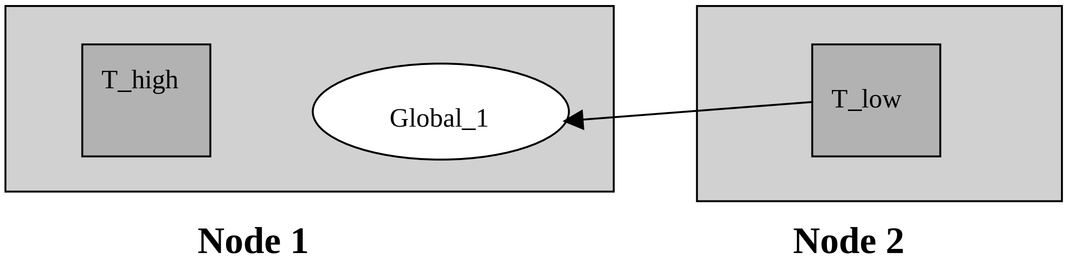
Notations:

In all examples we are considering distributed systems with RM algorithm and DPCP.

Our examples include two tasks T_high and T_low with higher and lower priorities, respectively. Global_*i* and Local_*i* stand for the global and local resources (as specified in Section 7.2.1) residing at Node *i*.

B.1 Dangerous Concerns

D1) GCS's of lower priority tasks at the "processor of interest" are not detected.



Example:

Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	10	0	0	10	7	1	
Low	100	0	0	100	50	2	Global_1 [0->50]

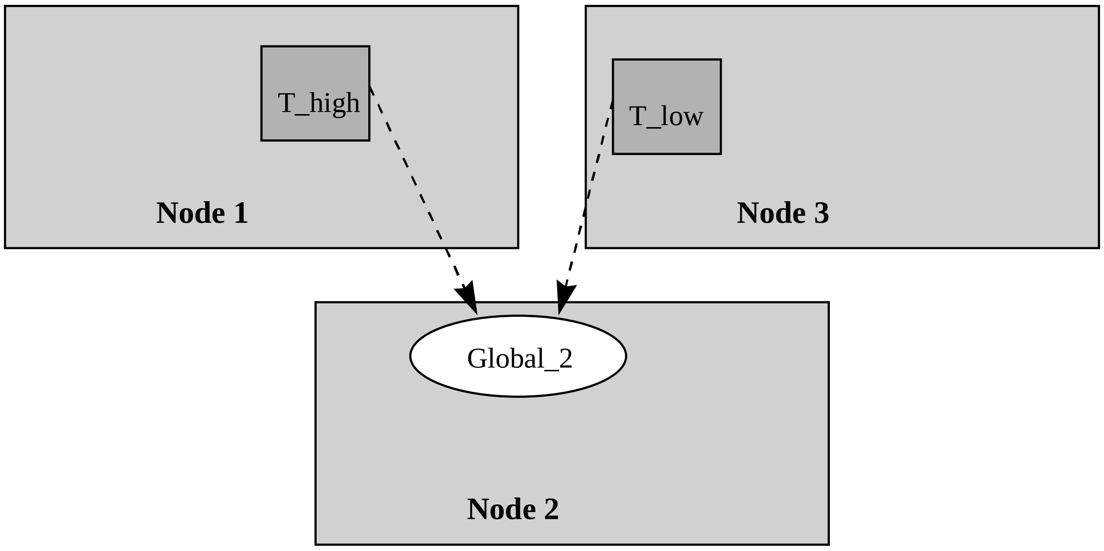
This system of tasks is reported by PERTS to be schedulable, while one can see that it is not: T_low executes its GCS at the Node_1 and preempts task T_high for 50 units of time. This causes T_high to miss its deadline five times before it starts its execution. Thus, the system is not schedulable.

Solution: Let us take a look at the schedulability criterion

$$W_i(t) = \sum_{j=1}^{i-1} C_j \left\lceil {}^{t}\!/_{T_j} \right\rceil + C_i + \beta_i$$

The first term in the schedulability criterion includes executions of higher priority tasks residing on the same node and all GCS's of higher priority tasks that reside at remote processors and access the host processor for GCS execution. The solution is to include executions of all GCS's of all tasks residing at remote processors and accessing the host processor for GCS execution, not only executions of GCS's of higher priority tasks.

D2) Global blocking: GCS's of lower priority tasks at remote processors are not detected.



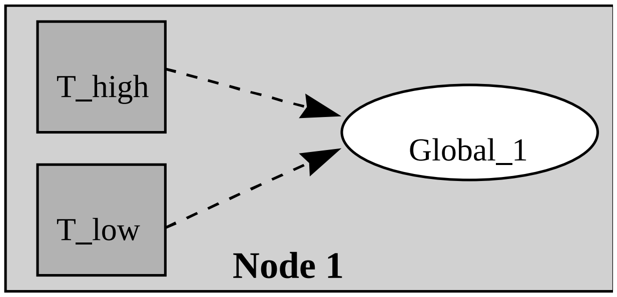
Example:

Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	10	0	0	10	10	1	Global_2 [1->2]
Low	100	0	0	100	50	3	Global_2 [0->2]

This system of tasks is reported by PERTS to be schedulable, while one can see that it is not: Task T_low executes its GCS at the Node_2 and blocks (Globally) task T_high for 1 unit of time. This causes task T_high to miss its deadline at time 10 since it has to execute 10 units before time 10. Thus, the system is not schedulable.

Solution: Redondo's [11] and Rajkumar's [8] schedulability theory is correct for this case, so the implementation in PERTS must be wrong.

D3) Local blocking: GCS's of lower priority tasks at host processor are not detected properly.



Example:

Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	10	1	0	10	7	1	
Low	100	0	0	100	25	1	Global_1 [0->5]

This system of tasks is reported to be schedulable by PERTS, while one can see that it is not: Task T_low executes its GCS at Node_1 and blocks task T_high for 4 units of time when it is ready to run ([1->5]). T_high starts to run only after task T_low releases Global_1 at time 5. Task T-high finishes its execution at time 12, missing its deadline at time 11. Thus, the system is not schedulable.

Solution: This type of error is due to a mistake in the schedulability theory behind it. Namely, the Local Blocking is said to be N+1 times the longest execution times of local critical section, b_l , which may block the task; where N is the number of GCS's executed by the task. We claim, that b_l should be replaced by max{ b_l , b_{gl} }, where b_{gl} is the longest execution time of a GCS executed at the host processor by lower priority tasks residing at that host processor.

B.2 Overly Pessimistic Cases

P1) Harmonic tasks with different Release Times

Example:

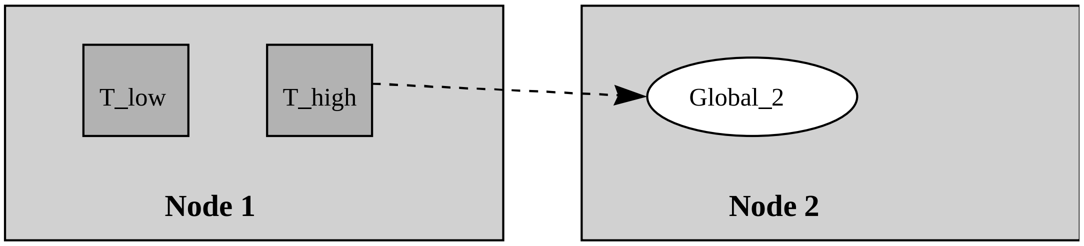
Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	5	3	0	1	1	1	
Low	10	0	0	3	3	1	

This system of tasks is reported to be non-schedulable by PERTS, while one can see that it is schedulable.

Task T_low runs at the Node_1 first because task T_high is not available yet due to task T_high's phasing. The task T_low finishes its job at time 3. The task T_high becomes ready at the same time and runs its execution for 1 unit of time, and also meets its deadline. Task T_high also executes between time 8 and time 9. The situation is identical during all future periods of time: [10->20], [20->30].... Thus, the system is schedulable.

Solution: This type of error is due Lehoczky's schedulability condition being designed to guarantee the schedulability of the worst phasing case. However, in the systems with harmonic tasks with staggered phasing, the worst phasing case occurs either every time or never. We have modified the original version of the Lehoczky's criterion to analyze the systems with harmonic tasks and presented it in Section 7.2.1.

P2) Including execution time of high priority tasks while they are executing GCSs at other processors.



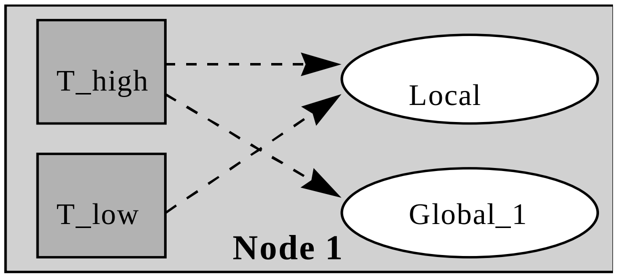
Example:

Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	10	0	0	10	8	1	Global_2 [1->8]
Low	11	0	0	11	4	1	

This system of tasks is reported by PERTS to be non-schedulable, while one can see that it is schedulable. Consider the worst case phasing: Task T_high starts its execution and after 1 unit of time starts the execution of its GCS at Node_2 and relinquishes the CPU on Node_1. At this tine, task T_low executes on the Node_1 and finishes its execution at time 5 (before its deadline). At time 8, task T_high finishes its execution (also before the deadline). Since we have considered the worst case phasing case, the system is always schedulable.

Solution: The calculation of the processor time demand function for a task i should not include the total execution times of the higher priority tasks, but only their execution times at the host processor - excluding the time spent by higher priority tasks while executing GCSs at other nodes.

P3) Local blocking: wrong number of GCS's used



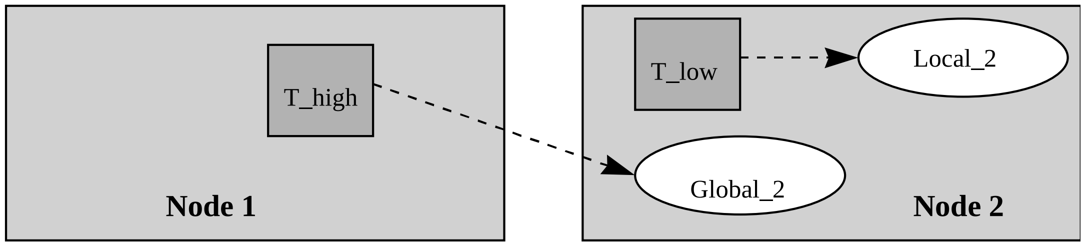
Example:

Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	10	0	0	10	9	1	Global_1 [1->8] Local_1 [0->1]
Low	1000	0	0	1000	9	1	Local_1[0->1]

This system of tasks is reported by PERTS to be non-schedulable, while one can see that it is schedulable.

Solution: The problem here is that task T_high accesses the same Local resource as task T_low and may be blocked by task T_low on that Local resource. This blocking is for the longest duration of the Local Critical Section (LCS) of task T_low. The schedulability theory assumes that the blocking may happen n+1 times, where n is the number of GCSs executed by task T_high. We claim that n should be the number of GCS's executed by task T_high at other than the host processor.

P4) Global blocking: reports a false blocking when GCS is bound to a processor with Local resource in use.



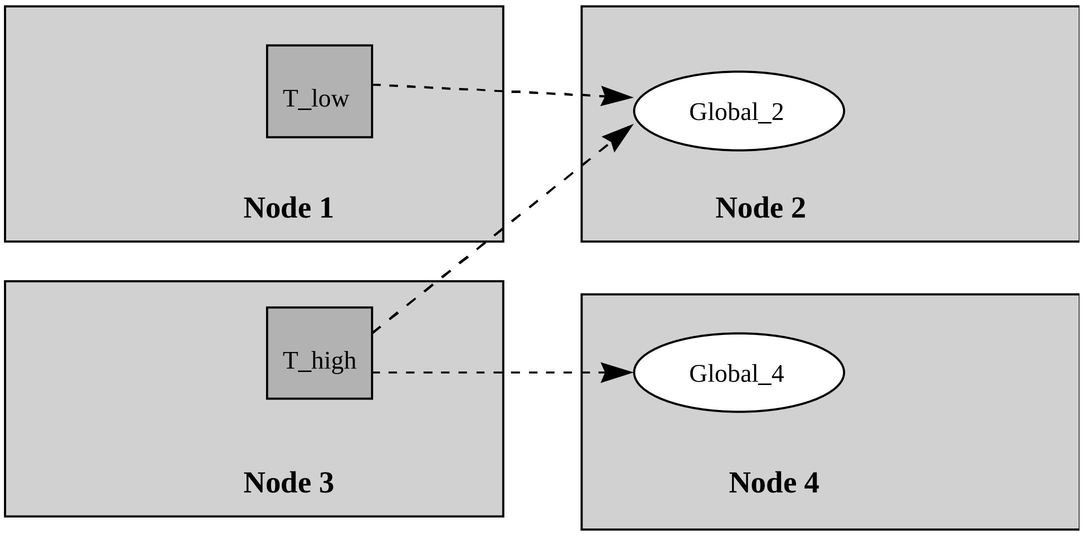
Example:

Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	10	0	0	10	9	1	Global_2 [1->2]
Low	100	0	0	100	11	2	Local_2[0->11]

This system of tasks is reported by PERTS to be non-schedulable, while one can see that it is schedulable.

Solution: This appears to be an implementation mistake in PERTS. The problem here is that PERTS wrongly reports the blocking of the GCS that is initiated by task T_high by the LCS of the task Low. This blocking by the LCS can never happen because under the DPCP all GCSs execute at the higher priorities than any LCS. This means that the LCS of task T_low will be preempted by the GCS of task T_high.

P5) Remote blocking: reports false blocking by all GCS's of higher priority tasks that are accessing the processor used by lower priority GCSs even if not all of these higher priority tasks are accessing the processor used by lower priority GCS's.



Example:

Task	Period	Phasing	Ready Time	Relative Deadline	Execution Time	Node	Resources
High	10	0	0	10	10	3	Global_2 [0->1] Global_4 [1->2]
Low	100	0	0	100	90	1	Global_2 [0->5]

This system of tasks is reported to be non-schedulable by PERTS, while one can see that it is schedulable.

Solution: The problem is the PERTS implementation. PERTS assumes that, since task T_low accesses the same node as task T_high, T_low may be blocked for the duration of all global critical sections of task T_high, which is wrong. PERTS should check conflicts only at nodes that are accessed by the task T_low's GCSs. In the example above, T_low does not access Node_4, and therefore T_low should not be blocked by task T_high's GCS on Node_4.

P6) The priority ceilings of global resources are assigned to zero.

Solution: This appears to be an error in the implementation of PERTS. Redondo's and Rajkumar's DPCP definition clearly state that the priority ceiling of global resources is the highest priority of the task accessing this resource plus the base priority ceiling, defined in the Section 7.2.1. This error unnecessarily increases blocking time in PERTS analysis.

VIII. Conclusion

In this thesis we have presented our contributions to real-time scheduling theory for distributed systems. We concentrated on techniques to perform schedulability analysis of Real-Time CORBA systems. On the basis of PERTS 3.0, we have developed an automated schedulability analysis tool for RT CORBA systems. We have discussed various aspects of the schedulability theory, including: modification of the Lehoczky's schedulability criterion for the systems containing harmonic tasks and for the RT systems built on operating systems with limited available priorities; comparison of the Liu-Layland's and Lehoczky's criteria for the systems with shared resources; description and comparison of two resource access protocols, DPCP and DASPCP.

This thesis has presented the ability of the PERTS to describe real-time systems and analyze their schedulability. We have modeled RT CORBA systems using PERTS primitives, such as ***resources***, ***tasks*** and their ***dependencies***. To support the schedulability analysis of the RT CORBA systems we have modified PERTS.

We have introduced new task parameters: Intermediate Deadline and Network QoS Parameter. It involved modification of the Task Graph Editor GUI to enable user to specify these new task parameters.

Since the original PERTS version could not support the schedulability analysis of a system of clients with intermediate deadlines, we have designed and implemented the Client->Tasks Translator. The Translator is called prior to the schedulability analysis to translate the RT CORBA clients into set of dependent tasks, based on clients intermediate deadlines, according to our model.

In order to take into account the Network Delay (a time that a remote service request spends travelling through the network) we have modified the Lehoczky's schedulability criterion. Based on this new criterion, we have modified PERTS schedulability analysis

We have performed the exhaustive testing of the new and modified PERTS components and demonstrated their correctness and proper behavior.

A significant part of our project addressed the problems of the schedulability analysis theory.

We have considered a potential problem when a real-time operating system provides less priorities than the task system requires (schedulability theory always assumes unlimited available priorities). Under these circumstances, a system that is predicted to be schedulable in practice might appear to be non-schedulable. In our study we have suggested one possible mapping of tasks to limited priorities. Under this approach, the priorities are not unique, so we have modified the Lehoczky's schedulability criterion to account for this situation in analysis.

We have detected a set of concerns in schedulability analysis under DPCP. We considered two categories of concerns: dangerous (when PERTS reports a task system schedulable while it is not schedulable) and pessimistic (when PERTS reports a tasks system non-schedulable when it is schedulable). Along with the examples demonstrating the misleading results of the schedulability analysis we suggest the solutions to these problems.

We have demonstrated that the Lehoczky's schedulability criterion is not necessary (but sufficient only) in the analysis of a system with harmonic tasks. We have eliminated an overly pessimistic assumption of "worth phasing case" made by Lehoczky *et. al*. [10]. If the harmonic tasks have different Ready Times then the "worth phasing case" will never occur. We have modified the Lehoczky's criterion to make it necessary in the case of harmonic tasks.

We have pinpointed that the End-to-End analysis of the dependent tasks is overly pessimistic. We have described a modification that could be used until a general criterion is developed for the schedulability analysis of the dependent tasks. In the RT CORBA systems clients are represented by the set of dependent tasks. All of the tasks generated from the same CORBA client never interfere (preempt or block) with each other. Eliminating these tasks in the calculation of the processor demand function in the Lehoczky's schedulability criterion, we make it less pessimistic.

We have compared Liu-Layland's and Lehoczky's schedulability criteria and have proven that the satisfaction of the first guarantees satisfaction of the latter one in all real-time systems. Based on it we have eliminated the first one from the schedulability analysis for PCP-DM and recommend the same changes for all combinations of the priority assignment mechanisms and resource access protocols supported by the Lehoczky's criterion.

We have described DPCP and DASPCP and have proven that the latter one is deadlock free and has a limited blocking time. We have compared concurrency in the real-time systems under DPCP and DASPCP. While it indicated the advantage of the DASPCP, we suggest to incorporate DASPCP into PERTS and CORBA only after previously described drawbacks are fixed.

This project is a first and necessary step towards creating analysis theory and tools for distributed real-time systems such as RT CORBA–based applications. On the basis of the PERTS 3.0 we have developed an automated schedulability analysis tool for RT CORBA systems. However, further steps are necessary before PERTS can fully analyze RT CORBA systems. Based on our modifications to the analysis theory for the distributed real-time systems, we have proposed the continuation of the project. The aspects to be addressed in the nearest future include: modification of the PERTS engine to account for the effect of limited available priorities in RTOS; improvement of the schedulability analysis of the system of harmonic and dependent tasks; elimination of the mistakes in the schedulability theory and PERTS implementation in analysis of the systems under DPCP; incorporation of the schedulability analysis of systems under DASPCP into PERTS and replacement of the DPCP by DASPCP in RT CORBA systems.

References

[1] Liu, C.L. and J,W,Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, Journal of the Association for Computing Machinery, Vol. 20, No.1, pp. 46-61, January 1973.

Mok, A.K., “Fundamental Design Problems of Distributed Systems for Hard Real-Time Environment”, Ph.D. Thesis, MIT, 1983.

Sprunt, B., Sha, L. and Lehoczky J.P., “Aperiodic Task Scheduling for Hard Real-Time Systems”, Journal of Real-Time Systems, pp.27-60, 1989.

Chen, M.I. “Schedulability Analysis of Resource Access Control Protocols in Real-Time Systems”, Ph.D. thesis, UIUC, 1991.

[2] Expressing and Enforcing Timing Constraints in a Dynamic Real-time CORBA System. Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh and Russell Johnston.

[3] Real-Time CORBA. Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh and Russell Johnston.

[4] http://www.tripac.com

[5] Leung, J. and Whitehead, J., “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks”, Performance Evaluation, 2, pp.237-250, 1982.

[6] Rajkumar, R., Sha, L. and Lehoczky, J.P., “Real-Time Synchronization of Multiprocessors”, Proceedings of the 9th Real-Time System Symposium, pp.259-269, December 1988.

[7] Sha, L., Rajkumar, R. and Lehoczky, J.P., “Priority-Inheritance Protocols: An Approach to Real-Time Synchronization”, IEEE Transactions on Computers, 39 (9), pp.1175-1185, September 1990.

[8] Rajkumar, R., “Synchronization in Real-Time Systems: A Priority Inheritance Approach”, Kluwer Academic Publishers, 1991.

[9] Baker, T.P., “A Stack-Based Allocation Policy for Real-Time Processes”, Proceedings of IEE 11th Real-Time Systems Symposium, pp. 191-200, December 1990.

[10] Lehoczky, J.P., Sha, L. and Ding, Y., “The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior”, Proceedings of the 10th Real-Time Systems Symposium, December 1989.

[11] Redondo, J.L., “Schedulability Analyzer Tool”, Technical Report No. UIUCDCS-R-93-1791, Department of Computer Science, University of Illinois, February 1993.

[12] http://www.infosys.tuwien.ac.at./Research/Corba/OMG/arch2.htm#446864
http://www.omg.org

[13] "Concurrency Control in Real-Time Object-Oriented Systems: The Affected Set Priority Ceiling Protocols" by Squadrito, DiPippo, Cooper, Esibov and Wolfe. To appear in proceedings of the First IEEE Symposium on Real-Time Object-Oriented Computing, Kyoto, Japan, April 1998.

[14] Michael A. Squadrito, “Extending the Priority Ceiling Protocol Using Read/Write Affected Sets”, M.S.Thesis, URI, 1996.