

ON-LINE SCHEDULING FOR REAL-TIME
CORBA ENVIRONMENTS

By

DIANA AURA DUMITRU

Master of Science

Western University of Timisoara

Timisoara, Romania

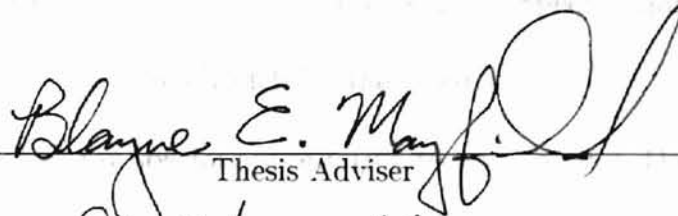
1996

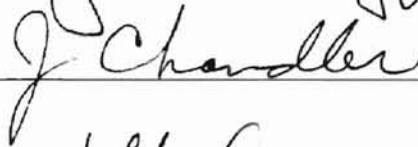
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2000

Oklahoma State University Library

ON-LINE SCHEDULING FOR REAL-TIME
CORBA ENVIRONMENTS

Thesis Approved:


Thesis Adviser






Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my research advisor, Dr. B. E. Mayfield for his outstanding supervision, guidance and moral support. My sincere appreciation extends to my other committee members, Dr. J. P. Chandler and Dr. H. K. Dai for their excellent guidance and assistance. Special thanks go to Dr. K. M. George and to Dr. M. Samadzadeh for their help.

I would like to give my special appreciation to my husband, Daniel, for his kind encouragement, love and understanding throughout this process.

Finally, I wish to thank to the Department of Computer Science and especially to Dr. B. E. Mayfield and Dr. J. P. Chandler for their friendly support and supervision during these two years of study.

Chapter		Page
	The Introduction	33
	Users and Client Factories	36
	TABLE OF CONTENTS	
	Knowledge	38
	Knowledge Representation	40
Chapter		Page
1	INTRODUCTION	1
2	THE CORBA STANDARD	5
	2.1 The Object Management Group	5
	2.2 The Architecture of a CORBA Application	7
	2.3 CORBA Benefits	11
3	ON-LINE SCHEDULING OF REAL-TIME TASKS	13
	3.1 Introduction to On-line Scheduling	13
	3.2 Real-time Scheduling	15
	3.3 Performance Measures of On-line Schedules	16
4	RELATED WORKS	19
5	ON-LINE SCHEDULING FOR CORBA ENVIRONMENTS	21
	5.1 A General System Model	21
	5.2 A Proposed System Model (Problem Statement)	23
	5.2.1 The On-line Scheduling Algorithm	26
	5.2.2 The Off-line Scheduling Algorithm	28
6	THE TEST SYSTEM	32
	6.1 The On-line System Architecture	32

Chapter	Page
6.1.1 The Controller	33
6.1.2 Clients and Client Factories	36
6.1.3 The Scheduler	38
6.2 The Off-line System Architecture	40
6.2.1 The Loader	41
6.2.2 The Off-line Scheduler	41
6.2.3 The Execution Simulator	42
6.3 Performance Tests	42
6.3.1 Experimental Configuration	43
6.3.2 Running the Test	43
6.3.3 Test Input	44
6.3.4 Test Results	46
7 CONCLUSIONS	52
BIBLIOGRAPHY	55

LIST OF TABLES

6.1	Operating environment used for tests.	43
-----	---	----

LIST OF FIGURES

2.1	The typical structure of a CORBA application	9
5.1	The simplest scenario involving ORBs.	21
6.1	On-line Test System Architecture	33
6.2	The Number of Late Jobs as a function of the System Load for On-line (solid) and Off-line (dotted) algorithms.	47
6.3	The variation of the Total Scheduling Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.	48
6.4	The variation of the Waiting Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.	49
6.5	The variation of the Lateness Penalty Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.	50
6.6	The variation of the Processing Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.	51

CHAPTER 1

INTRODUCTION

As the computer technology develops, distributed computing systems gain more and more importance because of their advantages over large centralized mainframes: flexibility, increased information transparency, interoperability, etc. Consequently the availability of low-cost microcomputers and the progress of the communication technology force the trend towards distributed systems.

An important problem arising from the use of distributed systems is to enable the interaction between machines built by different vendors which work under various operating systems. As a result great effort is put nowadays into setting some standard general frameworks which allow the integration of all common machines and operating systems. Examples of such standards are the OSF Distributed Computing Environment (DCE), the ISO Open Distributed Processing (ODP) Reference Model and the Object Management Group's (OMG) CORBA Specifications.

In the present work we use the OMG's Common Object Request Broker Architecture (CORBA) standard as an environment for distributed systems. The CORBA standard defines an architecture for distributed object-oriented systems which allows the integration of very heterogeneous components, using different programming

models and programming languages. Furthermore, the transparency of the platform and of the machine type on which they are implemented is ensured. Simply stated, CORBA allows applications to communicate with each other regardless of their location or implementation language. These valuable features make CORBA a good choice for industrial applications in areas such as advertising, marketing, aerospace/defense, banking/finance, chemical/petrochemical, electronic commerce, health care/insurance, manufacturing, publishing/multimedia, retail, telecommunications, transportation/travel, etc.

On the other hand, real-time applications are increasingly emerging in some of the above-mentioned areas such as telecommunications, automated manufacturing, aerospace, medical patient monitoring, multimedia and others. Therefore a natural demand for providing support for real-time applications arises. For such applications the CORBA environment must provide services that meet the functional requirements within the real-world timing constraints. Given a CORBA software that has to perform more than one task at a time (e.g. handle two concurrent requests) a way of allocating the resources needed for these tasks must be determined such that the time constraints are respected. These resources can be internal to the CORBA environment: processing resources, storage resources, communication resources or external ones such as server applications. For the moment the CORBA specifications do not contain approaches to ensure a good performance for real-time end applications. Therefore the design, implementation and performance measurements of different real-time strategies for CORBA environments have become an important research topic for the past few years.

The aim of this thesis is to propose, implement and evaluate the performance of such a real-time service strategy. The on-line algorithm that will be implemented was first proposed by Hong and Leung [10] in 1992 and falls within the interest of Operating System Theory. We will transfer and adapt it to our CORBA environment in order to determine whether it is suitable or not for distributed applications. A few changes were made to the initial algorithm in order to improve it. While initially its only goal was to build a feasible schedule, now it is also attempting to minimize as much as possible the total scheduling cost (i.e the sum of waiting, penalty and processing costs). In on-line computation theory the performance of an algorithm is usually determined by comparing it with the performance of an equivalent optimal off-line algorithm. Unfortunately, a comparison of our preemptive on-line algorithm with an equivalent *preemptive* off-line one is not feasible. Despite the fact that the scheduling results of such an off-line algorithm would probably be more efficient than those of our on-line one, the time required to determine the optimal schedule is very large even for a small number of tasks. Therefore preemptive off-line algorithms are not suitable for real-time applications.

Our purpose is to implement both the preemptive on-line scheduling and a non-preemptive off-line scheduling commonly used for this type of problems and to compare their performances.

The thesis is organized as follows. Chapter 2 gives a short overview of the OMG's CORBA standard. A brief introduction to scheduling theory is presented in Chapter 3. Here we focus on the basic concepts of the on-line approach, introduce the notion of real-time scheduling and explain the methods used in the performance measures

of scheduling algorithms. In Chapter 4 a brief review of the literature on the subject of real-time CORBA technology is presented. A system model to be studied is proposed in Chapter 5 and two different scheduling strategies (static and dynamic) are presented for further evaluation. Chapter 6 describes the test system developed for the evaluation of the strategies and the results obtained. In addition to the architecture of the system, the functionality of the system components and their implementations are explained in detail. Chapter 7 contains our conclusions and suggestions for further work.

CHAPTER 2

THE CORBA STANDARD

During the beginning of the 1990's two technologies began to be actively used: distributed computing and object-oriented programming. In order to benefit from the advantages of these technologies it was logical to use them together while developing applications. As a consequence, the Object Management Group (OMG) was formed with the purpose of setting some standards for distributed object computing. The Common Object Request Broker Architecture (CORBA) is the result of OMG's effort in this area [12].

2.1 The Object Management Group

The Object Management Group was founded in 1989. On start-up, there were less than a dozen corporate members in the consortium. They were anticipating that the computing superstructure was about to change and their vision was to organize the various system vendors and software developers into a consortium which would set some standards for developing portable distributed applications for heterogeneous systems. OMG has received a tremendous amount of industry backing since then

and is now the world's largest software consortium, with more than 800 members, including system vendors, application developers, academic institutions and users.

The primary goal of the OMG is to solve problems arising from the development of large software systems. Because software grows more and more complex, traditional approaches to producing software are no longer sufficient to develop programs in an efficient way. This increased complexity of modern applications is also caused by the fast emergence of very heterogeneous systems (i.e. systems produced by different vendors and running different operating systems).

To solve these problems, the OMG proposes a framework for software development which is based on two computing paradigms. First, the *object-oriented programming* paradigm is adopted because it proved to be a suitable method for construction of large software systems. The second paradigm integrated is inspired by the trend towards *distributed computing*, which has a great influence on the development of future information systems. Thus the OMG's efforts concentrate on the definition of a framework for the development of *distributed object-oriented software systems*. Object-orientation provides a method for improving reusability, portability and interoperability of software products. By defining individual components in terms of object-oriented interfaces, which do not tackle implementation aspects, the platform independence of the interfaces is guaranteed and an interaction between components running in a heterogeneous environment is therefore enabled. Another important aspect for the OMG is the conformance of the released standards with existing standards in the software industry so that an integration into existing systems is possible and the interaction with systems conforming to other standards is enabled.

The ORB 2.2 The Architecture of a CORBA Application

The main components of a CORBA application are the client code, the server code (which contains the implementation of the server object), the Object Request Broker (ORB), the Object Adapter, the object's interface declarations, the skeleton and the stub codes.

The Client and Server Applications. Both the client and the server (object implementation) codes can be written by the CORBA user in any object-oriented language supported by the CORBA environment. The client sends a request/message which needs to be carried out by the object instantiated within the server application. The two applications can reside on different hosts having different operating systems, with the only constraint that a CORBA software (but not necessarily the same one) be installed on both machines.

The Object Request Broker (ORB). The ORB is the core part of any CORBA application and it is responsible for all interactions between the client and the server codes. It processes any client invocation, converts it into a message format (marshaling) and sends it to an appropriate server object in a transparent manner. On the server side, the ORB receives the messages from the client, processes the messages (demarshaling) and passes the results back to the client application. Thus, the ORB takes care of the most of the house-keeping activities such as creating and closing sockets, network operations and request demultiplexing.

The Object Adapter (OA). When a client makes invocations on a server object, the OA helps the ORB to activate the appropriate object and to deliver requests to it. The OMG has defined a Basic Object Adapter (BOA) which has to be supported by all CORBA-compliant ORBs. The BOA is being superseded by the Portable Object Adapter (POA), which is more tightly specified and it is aimed to increase the portability of object implementations.

The Object's Interface. The interface of the object that performs the services is defined by the user in the OMG Interface Definition Language (IDL). This ensures that the interface is not dependent on the particular language in which the object is implemented. The interface definition specifies the operations the object is prepared to perform, the input and output parameters they require and any exceptions that may be raised along the way. All this information is needed by the client in order to build its requests. An advantage brought by the interface is that clients see only the object's interface and not the implementation details.

The Stub and the Skeleton. The stub and the skeleton codes are produced by compiling the object's interface code with the IDL compiler. They represent the client-side/server-side mappings of the interface and they intermediate the interaction between the client/object code and the ORB. While the stub contains function declarations used to issue invocations for the object implementation, the skeleton contains function calls which pass the incoming requests to the object implementation.

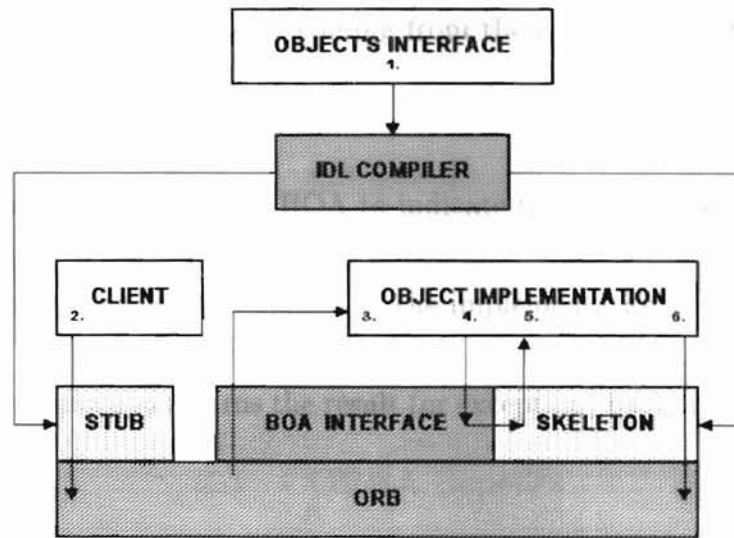


Figure 2.1: The typical structure of a CORBA application

The typical interaction between these elements is presented in Figure 2.1. Different shades of gray are used to distinguish between components with different origins. The **Object's Interface**, the **Client** and the **Object Implementation** (in white) represent the code written by the CORBA user while the **IDL Compiler**, the **BOA Interface** and the **ORB** (in dark gray) constitute the CORBA environment itself. Finally, the **Skeleton** and the **Stub** (in light gray) are generated by the CORBA software in a particular programming language; they are used as starting point of the code created by the user. Let us describe briefly how the various application components interact with each other.

1. The object's interface produces by compilation the stub and the skeleton code.
2. After receiving a reference to the server object, the client calls methods on this object through the stub.

3. The ORB hands the requests coming from the client to the BOA, which activates the implementation.
4. The implementation invokes BOA to indicate that it is active and available.
5. The BOA passes method requests to the implementation via the skeleton.
6. The implementation returns the result (or exception) back to the client through the ORB.

For realistic applications, the basic interaction facilities provided by the ORB are not sufficient. To support further functionality, *Object Services* and *Common Facilities* are provided. The Object Services perform system-oriented low-level services for application development such as the *Naming Service* which allows clients to obtain references for objects of certain types, or the *Object Life Cycle Services* which are used to create, delete, move or copy objects. In contrast to these basic services, the Common Facilities provide more application-oriented services at a higher level, such as compound document management, electronic mail, or help systems. The overall goal of these services is to reduce the complexity of software development by supporting services which are often needed in many application programs.

Another important point which has not been entirely covered yet is how clients specify to which object implementation they wish to send a request. For this purpose each object implementation is assigned by the ORB an unique *object reference*, which is an opaque representation used to denote an object in the CORBA-based environment. There are several techniques that can be adopted by the client in order to obtain a reference to the desired server but the one used in the present work

is through the Naming Service. The objects need to register their references with this service in order to be later looked up and called by clients. A client itself has no information about the location, the language, or any other details of the object implementation and therefore it becomes the responsibility of the ORB to locate and deliver the requests to the appropriate object.

2.3 CORBA Benefits

CORBA provides the abstractions and services one needs in order to develop complex distributed object-oriented applications. Some of its most important features can be summarized as follows:

platform independence; CORBA objects can be used on any platform for which there is a CORBA ORB implementation (this includes virtually all modern operating systems as well as some not-so-modern ones).

language independence; CORBA clients and servers can be implemented in any object-oriented programming language.

portability and easy integration of legacy systems; CORBA's separation of the object definition from its implementation is perfect for encapsulating existing applications.

multiple request-respond capabilities; CORBA provides support for multiple request-respond models.

high-level language bindings; CORBA provides language-neutral data types and instructions which make object interactions possible across languages and operating systems without worrying about low-level details.

local/remote transparency; an ORB can run in a stand-alone mode or can be interconnected to other ORB's using CORBA 2.0's Internet Inter-ORB Protocol (IIOP) services. Since the communication between hosts in a CORBA application is done at the ORB level, the location transparency of the server object with respect to the client is ensured.

built-in security of transactions; the ORB includes context information in its messages to handle the security of transactions across machine and ORB boundaries.

static and dynamic method invocation; a CORBA ORB allows one to either statically define the method invocations at compile time or to dynamically discover them at run-time.

polymorphic messaging; since the ORB invokes a function on a target object the same function call will have different effects, depending on the object that receives it.

CHAPTER 3

ON-LINE SCHEDULING OF REAL-TIME TASKS

We start with the *ancient scheduling problem* in which a sequence of n tasks/jobs have to be scheduled on m machines or other available resources. The tasks are assumed to arrive at different moments in time, are usually characterized by their running time (also called processing time) and have to be assigned for that time period to one or more resources. The goal of any schedule is to distribute the jobs to the various resources as efficiently as possible, such that some objective function (performance measure) is optimized.

3.1 Introduction to On-line Scheduling

There exist two distinct scheduling paradigms for the above problem, namely the off-line (static) approach and the on-line (dynamic) one. Off-line scheduling consists in building a scheduling plan at the beginning of a scheduling period and results in a complete, fixed schedule for that period. One of the major disadvantages of this approach is that it must make assumptions about the exact parameters (such as, for instance, the execution time) of each task to be scheduled. Whenever the actual

parameters differ from these assumptions the schedule will become invalid. This problem can be solved by the *on-line scheduling algorithms* in which the scheduling is performed at runtime only with a partial knowledge of the input. We know only past events, without having any information about the future. These algorithms fit much better into a realistic scenario than the off-line algorithms, where the scheduler handles the requests knowing ahead of time the parameter values of all the tasks in the system.

The on-line scheduling algorithms have been explicitly studied for more than 30 years. The first article to propose an efficient on-line schedule was published in 1966 by Graham [9] and it analyzed a simple greedy algorithm, nowadays commonly called *List Scheduling*. Two other early results about on-line scheduling algorithms are developed by Sahni and Cho [17] in 1979 and by Davis and Jaffe [4] in 1981. Since then, many other articles were published, concerning different variants of on-line scheduling. The on-line scheduling algorithms can be classified according to the information given on-line at the task arrival time [5]:

- **Scheduling jobs one by one**

In this paradigm each job is scheduled at arrival time, before the next job is encountered. As soon as a job is presented, the scheduler also learns its characteristics, including the running time.

- **Unknown running time**

Unlike the previous paradigm, this algorithm assumes that the running time of the tasks remains unknown at arrival time and, even more, it will not be de-

terminated until the task is completed. Therefore the only information available about a task at any time is its state, which can be one of the following three: *available*, *running* or *terminated*. Another difference between this schedule and the previous one is that now all available jobs are at the disposal of the scheduler in order to be either scheduled or further delayed.

- **Clairvoyant algorithms (Tasks arrive over time)**

This paradigm is similar to the previous one with the only difference that the running times are known at job arrival. Therefore the single on-line feature is the lack of knowledge of jobs arriving in the future.

- **Interval scheduling**

In this case, when the scheduler "learns" about a new arriving job, it is also constrained to schedule it within a certain time interval. Contrary to the previous three paradigms, this one does not allow a job to be postponed indefinitely.

The on-line schedule which will be studied in this work is an interval schedule.

3.2 Real-time Scheduling

In the real-time scheduling there are rigid time constraints on the operation of the resources or the flow of data in the system. If these constraints are not respected the system will fail. Each task is usually characterized by its *release time* which is the earliest time when it can be scheduled and in the on-line setting it also represents the time when the scheduler learns about the existence of that particular task. While

in real-time, off-line scheduling each task is also described by an extra parameter, namely its *individual deadline*, it has been proven that in the case of on-line scheduling there can exist only a single deadline, common to all tasks in the system [10]. Otherwise no feasible on-line scheduling algorithm can exist, where by feasible we mean that each task has to be terminated prior to the expiration of its deadline.

3.3 Performance Measures of On-line Schedules

The traditional approach to studying the quality of an on-line scheduling algorithm falls within the framework of *distributional complexity* (or *average-case complexity*). In this case an assumption is made regarding the distribution of the events (i.e. the distribution of task arrivals over time) and the expected total cost or expected cost per event is evaluated. During the past 10 years the interest in this subject has been redirected towards a new approach called *competitive analysis* [2]. In this latter theory the performance of an algorithm is determined by the value of the *competitive ratio* σ w.r.t. some objective function f and can be expressed as:

$$\sigma = \frac{f_{on-line\ alg}(workload)}{f_{off-line\ alg}(workload)} \quad (3.1)$$

Therefore competitive analysis falls within the framework of the *worst-case complexity*. An on-line algorithm is said to be σ - *competitive* if for each input sequence the objective value is at most σ times larger than that of an equivalent optimal off-line algorithm. The competitive ratio may depend on the number of resources or other system parameters and it is usually a number greater than 1. However, since our off-line and on-line schedules are not equivalent (that is, they are not both

preemptive), the latter can have a worse performance and the competitive ratio can become less than 1 for some inputs.

There is a large number of objective functions depending on the various goals that we try to accomplish by using the scheduling algorithms. Some of the most important measures are enumerated in [20]:

- *Processing Time* or *Makespan* which is the time interval during which all tasks have been carried out for a given input sequence.
- *Waiting time* which for a particular job is defined as the total sum of the periods spent waiting in the ready queue.
- *Tardiness* which for a particular job is defined as the amount by which the completion time of the job exceeds its deadline, thus a positive tardiness indicates that the job will not meet its deadline.
- *Turnaround time* which for a particular job is defined as the interval from the time of its submission to the time of its completion.
- *Response time* which for a particular job is defined as the time from the submission of a request until the first response is produced.
- *Resource utilization* which for a particular resource is defined as the time the resource is busy.

The objective functions are defined as the sums of these values over all jobs or the average values over all jobs, since these differ from the first ones only by a factor proportional to the number of jobs.

The on-line scheduling algorithm which will be studied in this thesis attempts to determine a feasible schedule. If such a feasible schedule cannot be found for the input task system, the algorithm builds a schedule in which a minimum number of jobs miss their deadlines and for which the sum of the processing time, waiting time and tardiness is minimized.

CHAPTER 4

RELATED WORKS

During the past few years CORBA applications became more and more popular in the industry and the demand for real-time support increased accordingly. This is why a new research topic started to develop rapidly: real-time CORBA technology. There are a lot of articles published lately on this subject but one of the most important contributions was made by the Real-Time Special Interest Group (RTSIG). The group was created in 1995 and works within the Object Management Group, their goal being to augment existing CORBA technology for the requirements of real-time systems. The group has put together a joint proposal for Real-Time CORBA Specifications which describes the features that should be supported by a real-time ORB and provides pointers on how to implement these features. The proposal has passed the voting stage and the joined revised submission [13] has been recently completed. However, the roles of certain modules such as the Real-Time Scheduling Service (RTSS) are loosely defined. It is important to realize that real-time applications are diverse and require different levels of service quality. The RTSIG has proposed a priority-based approach to provide real-time support for the objects. However, studies have shown that a priority-based approach is not always suitable

for handling multimedia applications [8, 21] and therefore there is a need to provide alternative real-time support for these types of objects. Another drawback of Real-Time CORBA Version 1.0 proposal is that it does not address dynamic scheduling. The group is currently working on a second project whose objective is to extend the scheduling mechanism and the interfaces defined in the Real-Time CORBA 1.0 to support applications requiring dynamic resource allocation.

Another important result is the TAO project developed by the Distributed Object Computing Group at Washington University [18]. TAO is a real-time CORBA implementation which uses a *nearly on-line* scheduling mechanism. This means that it schedules the tasks like an on-line scheduler except that it needs some additional information about the jobs coming in the future.

A number of other real-time CORBA implementations exist on the market today with varying levels of support for real-time applications. Some examples are ORBIX from IONA Technologies [11] and DIMMA - the microkernel ORB from ANSA [1]. However, a lot of work still needs to be done to provide end-to-end support for real-time objects.

CHAPTER 5
ON-LINE SCHEDULING FOR CORBA ENVIRONMENTS

5.1 A General System Model

As discussed in the second chapter, CORBA applications are basically client/server systems in which one or more client applications request services from a set of server applications. The services are subsequently carried out by different resources available to the servers. Each client performs a sequence of operations, where each operation may invoke a request from a specific server. The simplest scenario involving ORBs, namely a single client interacting with a single server is shown in Figure 5.1. The solid arrows represent the request made by the client and routed by the ORB



Figure 5.1: The simplest scenario involving ORBs.

towards the server. The dashed arrows represent the message that might be sent by the server in response to the request (however, this is not necessary to happen).

In real-time applications we are usually interested in achieving an end-to-end high service quality, which means that we are trying to minimize the period of time from the issue of the request until its completion. Given the complexity of the underlying mechanism that enables the server-client interaction (i.e. the ORB itself) this problem becomes a very complicated one. The overhead that occurs in such applications is influenced by a large number of factors, most of them being pointed out by Smith in some of his papers [6, 7, 19]. These include:

- the speed of the IDL compiler
- non-optimized data copying and memory management
- inefficient receiver-side demultiplexing
- excessive control information carried in request messages
- inefficient design of network adapters
- inefficient protocol implementations and improper integration with the I/O subsystem
- inefficient implementations of the ORB transport protocols
- lack of proper integration with the operating system.

We can conclude that there are various ways of reducing the overhead but in the case of multiple servers and clients it is obvious that the most important one remains the scheduling algorithm itself. The implementation and performance testing of such a scheduling algorithm will make the object of our study.

5.2 A Proposed System Model (Problem Statement)

The proposed scenario for our study is the following: the system will consist of a set of client applications, a set of identical resources (servers) and a scheduler that has to route the requests made by the clients to the various servers. In fact, this is the most common general structure of nontrivial applications in a CORBA environment. All the service requests will be treated by the scheduler as unrelated tasks. We assume that the clients make requests in a random manner (i.e. from the scheduler viewpoint the tasks have random arrival times) and that the tasks have soft deadlines.

We will also consider that the system can handle *urgent tasks*, that is tasks that must be assigned to a resource as soon as they are created. Even more, we will allow these urgent tasks to have arbitrary deadlines as long as the non-urgent ones still have one common deadline. Introducing urgent tasks in our system allows us to also handle cases in which the resources can become unexpectedly unavailable. This situation can be simulated by using the urgent tasks, with the only restriction that the duration of the down time has to be made known. Thus whenever a resource becomes unavailable we can consider that it is executing an urgent task that has just arrived.

The following notations can be used to describe the system in study:

$\{R_j, j = 1..m\}$ the set of available resources in the system, $m \geq 1$

$\{C_i, i = 1..l\}$ the set of clients in the system

$\{T_k, k = 1..n\}$ the set of tasks/requests in the system generated by the clients.

Each task T_k is characterized by the following set of data $(r_k, d_k, e_k, w_k, p_k)$ where:

- r_k stands for the *release time* or *arrival time*, which represents the time when the scheduler learns about the new service request T_k .
- d_k stands for the *deadline*, which represents the time when the task T_k is due.
- e_k stands for the *execution time*, which represents the time necessary for completion of task T_k .
- w_k stands for the *waiting cost per waiting time unit* associated with task T_k .
- p_k stands for the *lateness penalty cost per delay time unit* associated with task T_k .

Since all the resources are identical, the only parameter describing them, the *processing cost per resource*, will have the same value for all resources. We will consider that a task system T is feasible if there exists a scheduling algorithm such that each task T_k is executed within its *executable interval* $EI_k = [r_k, d_k]$. Also, using these notations, the deadline of any urgent task T_k can be expressed as $d_k = r_k + e_k$. For each task T_k the waiting and lateness penalty costs can be computed as follows:

$$W_k = w_k \times \text{job waiting time}$$

$$P_k = p_k \times \text{job tardiness}$$

and therefore the global waiting and lateness penalty costs will be the sum over all jobs in the system:

$$W = \sum_{k=1}^n W_k$$

$$P = \sum_{k=1}^n P_k$$

The processing cost of the given task system can be expressed as:

$$PR = m \times \text{processing time} \times pr \quad (5.1)$$

where pr is the processing cost per time unit and per resource. We can compute now the total scheduling cost for scheduling the given task system as:

$$C = W + P + PR = \sum_{k=1}^n (W_k + P_k) + PR \quad (5.2)$$

Because of the unpredictable nature of request arrivals, it is extremely difficult to design a dynamic real-time system which guarantees that the deadline of all tasks will be met. In practice, for heavily loaded systems, we are concerned with assuring the completion of as many tasks as possible in the case of tasks with equal priorities, or the completion of the most important tasks in the case of priority based schedules. Consequently we need to build an algorithm which works as follows. Every time a new task is created, its release time, execution time, deadline and waiting and lateness penalty costs are made known to the system. Next, the scheduler has to evaluate whether the newly arrived task along with the other unfinished tasks, currently executing, can be completed so that all the deadlines are met. If all the deadlines can be met, the system will execute the tasks according to the schedule constructed by the on-line scheduler. Otherwise, the system will try to execute as many tasks as possible or the most important tasks prior to the expiration of their deadlines, allowing the remaining tasks to exceed their deadlines. Even more, the total scheduling cost of the system will be minimized. Such an on-line algorithm is

proposed in the next section and will be implemented in our CORBA environment.

5.2.1 The On-line Scheduling Algorithm

The on-line scheduling algorithm proposed in this section uses as starting point the Hong and Leung's algorithm [10] which was designed for a system having features similar to those discussed previously. As mentioned before, the goal of the original version was to construct only feasible schedules and it has been modified in order to minimize the total scheduling cost as well. The algorithm employs a subprocedure called Slack-Time Algorithm to reschedule tasks whenever new tasks arrive. To describe this subroutine we have to define one extra parameter characteristic to all the tasks in the system called *slack time*. For an active task T_k the slack time is defined to be $s_k = d_k - t - e_k$ where t is the current time value and e_k is the remaining execution time for T_k . Thus the slack time of an urgent task is 0 at its release time. The policy of the Slack Time Algorithm is to schedule the tasks with the shortest slack time first, one task per resource. If there are several tasks having the same slack time and if the number of resources left over is less than the number of these tasks, then the remaining resources will be shared among those tasks. Each of the resource sharing tasks will be allocated a resource time proportional to the sum of its waiting and lateness penalty costs. On the other hand, since a task cannot be executed concurrently on two different resources, it cannot be allocated a time larger than the time for which the resources are shared. Let's denote the set of the active tasks in the system at the current time t by $A(t)$. The Slack-Time Algorithm

has the structure presented below.

The Slack-Time Algorithm

1. FORALL tasks $T_k \in A(t)$ DO
 COMPUTE the current slack time s_k ;
OD
2. IF \exists tasks $T_k \in A(t)$ such that $s_k < 0$
 OR the number of tasks T_k with $s_k = 0$ is larger than m
 THEN declare the system infeasible;
 FI
3. WHILE the number of available resources > 0
 AND the number of tasks in waiting queue > 0 DO
 ASSIGN the tasks starting with those having the lowest slack times
 to the resources starting with the highest indexed ones;
 IF several tasks T_k have the same slack times s_k
 AND there are not enough resources to be assigned to the tasks
 THEN the tasks share the remaining resources proportionally to their
 waiting and lateness penalty costs;
 FI
 OD
4. IF a task T_k is executing at a faster rate than other tasks with
 smaller slack times OR a task is finished
 THEN {
 release all resources;
 place the unfinished jobs back in the waiting queue;
 goto Step 1;}
 FI
5. IF the constructed schedule has missed a deadline
 THEN declare the task system infeasible
 FI

The Slack-Time procedure can be implemented to run in $O(n^2)$ time. There is one important feature of this subroutine that needs to be highlighted: it is a resource-sharing schedule or a preemptive one. In this case we can use both terms interchangeably since, as shown in [3], a resource-sharing algorithm can always be transformed into a preemptive one and the tasks will preserve their previous completion times. Furthermore, the tasks which share some of the resources will obtain in

the preemptive case the same amount of processing time as in the resource-sharing algorithm at each integer-valued time. Therefore tasks can maintain in both cases the same slack time at each integer-valued time. Since the release times of the tasks are integers, this property can be maintained at every instant when a new task is released.

Using the Slack Time procedure we can now introduce the actual scheduling algorithm to schedule the requests coming from the client applications.

The Scheduling Algorithm

```
Whenever new tasks arrive DO
{
  t ← the current time
  A(t) ← the set of active tasks in the system at time t
  Call the Slack-Time Algorithm to schedule the tasks in A(t)
}
```

Hong and Leung specify in their paper that the running time of the scheduling algorithm is $O(r n^2)$ where r is the number of release times and n the number of tasks. Since more than one task can be released at one time by different clients, the number of release times in the system can be at most equal to the number of tasks in the system. Therefore the worst-case running time for the scheduling algorithm becomes $O(n^3)$.

5.2.2 The Off-line Scheduling Algorithm

We will determine the performance of the previous on-line scheduling algorithm by comparing it with the performance of an optimal off-line algorithm proposed by Rajaraman [16]. Given the complexity of the problem to be solved, i.e. the existence

of more than one resources, of different task arrival times and of various deadlines as well as all the optimization criteria considered, it becomes very difficult to construct an efficient static scheduling algorithm. This is because in the static case the only way of finding an optimal job sequence is to enumerate all the possibilities taking into account the task arrival time and resource availability constraints. For some particular cases (i.e. scheduling on a single resource) or by making simplifications regarding some parameters (e.g arrival times, deadlines) or even by totally eliminating them, the scheduling problem can be reduced to a maximal-flow network problem. This approach can be more efficient than the enumeration method but still requires $O(n^3)$ running time.

There exist several attempts to improve the enumeration algorithm with similar results. Among them is Rajaraman's [16] iterative solution which eliminates at each stage the task sequences that promise to perform worse than other existent ones. This reduces to some extent the run-time, depending on the input task system considered. Still, the algorithm has to take into consideration a relatively large number of task permutations. Therefore job preemption cannot be taken in consideration as it would increase significantly the already large number of task permutations to be analyzed. This is why one of the constraints imposed upon this off-line algorithm is that once a job starts executing on a resource, it proceeds until completion (that is, job splitting is not allowed).

The static algorithm which will be implemented starts by constructing C_m^n partial sequences of length m , such that each sequence has an unique set of m jobs out of the total of n jobs in the system. Although the sequences are aimed to indicate the

order in which the jobs will later obtain resources, at this first stage the order of the jobs is not important since all the m resources are considered to be available at the beginning. Hence each of the jobs in this initial sequence is allocated to a resource at arrival time, with no delay and the sequence order is ignored. From these sequences new sequences of length $m + 1$ are generated by taking each sequence of length m and adding, in turn, all the $(n - m)$ possible jobs to it. Thus $C_m^n (n - m)$ sequences of length $m + 1$ are constructed. From these new sequences we form groups such that each group has the same $m + 1$ jobs but in a different order. In each group we consider sequences in pair and eliminate one of them if the criterion below is met by both sequences:

- Let i and j be two permutations of the same job sequence.
- Eliminate sequence i if $a_i > a_j$ and $C_i \geq C_j$.
- Eliminate sequence j if $a_i \leq a_j$ and $C_i \leq C_j$.

where $a_{i(j)}$ is the earliest resource available time and $C_{i(j)}$ is the total scheduling cost for the sequence $i(j)$. In other words, if for a given set of tasks there exists a permutation such that its total scheduling cost is minimal and, in addition, it releases a resource earlier than another existing permutation, then it becomes optimal and will perform better even when new tasks are added to the system. This allows us to discard the inefficient permutations. If none of the previous conditions are met, both sequences will be retained. This procedure is repeated for all possible pairs in a group and for all groups. Notice that paired comparison is done only within groups.

With the remaining sequences of length $m+1$ we generate new sequences of length $m+2$. Thereafter we form the groups once again and go through the elimination procedure. Finally, when we arrive at sequences of length n there will be only one group which can be searched for the optimal sequence. We will look for a sequence that has the minimum number of tasks missing their deadline and the minimum scheduling cost.

The main advantage of the algorithm is that it essentially eliminates a certain number of partial sequences at each stage. The larger the number of partial sequences eliminated and the earlier the elimination takes place, the better the performance of the algorithm will be. If, for example, at any stage when the sequences are of length i we eliminate one partial sequence, then $(n-i)!$ sequences are discarded from future consideration. The algorithm requires $(n-m)$ stages for finding the optimal solution. Since the run-time can vary considerably from one input to another, we can only say that the worst case run-time for this algorithm is the same as the one for the enumeration algorithm, but the best case is slightly improved. Still, this algorithm can not be used for large sets of jobs to be scheduled.

CHAPTER 6

THE TEST SYSTEM

This chapter describes the test system developed for the evaluation of the on-line scheduling algorithm for CORBA environments.

6.1 The On-line System Architecture

The main components of the on-line test system are: the Controller, the Client Factories, the Clients and the Scheduler. The general structure of the system and the way these different parts interact with each other are presented in Figure 6.1. Furthermore, the test system contains some additional components such as a test input parameter file and a test output parameter file, which are aimed to set up the tests and collect the results of the evaluation.

The general procedure for performance tests is as follows: the Controller reads the test parameters from an input file. Some of the most important parameters set at this point are the period of time for which Clients will be generated and the names of the machines on which the Controller can randomly create them. A Client Factory application will be running on each of these hosts. The task of the Client

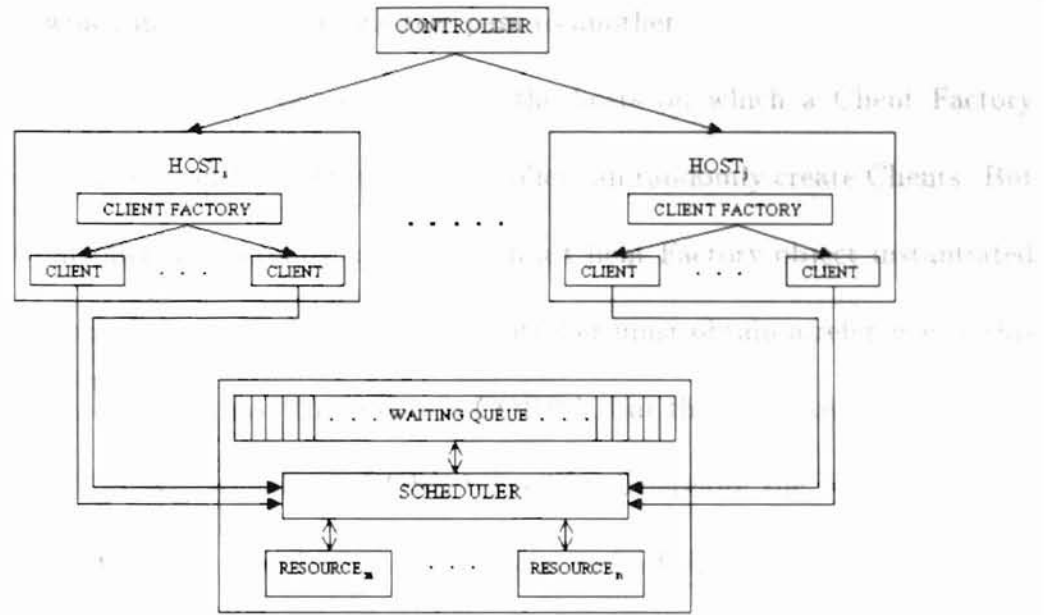


Figure 6.1: On-line Test System Architecture

Factories is to generate local Client objects each time they receive a request from the Controller to do so. Once a new Client is created, the Scheduler is notified immediately of its existence and decides, depending on the other Clients existent in the system, whether to allocate a resource to it or to place it in the waiting queue. The Scheduler is the central part of the whole application. Having access to the waiting queue and knowing at each moment the status of all resources, it schedules the Clients dynamically on the available resources such that the output parameters are optimized.

6.1.1 The Controller

The Controller is the component responsible for the test system initialization. Every time a new test is started, the Controller reads from an input file the test

parameters which may vary from one test pass to another.

The first parameters to be specified are the hosts on which a Client Factory application is running and on which the Controller can randomly create Clients. But since the Controller will actually interact with a Client Factory object instantiated within the Client Factory application, the Controller must obtain a reference to this object. There are many ways to do this in CORBA, but the one chosen in this case is the Naming Service, which is a CORBA Facility. Therefore the Naming Service application has to run as a background process on each of the hosts on which Clients will be created as well as on the host on which the Scheduler application resides, as we will see later. Before the Controller can be started, each Client Factory object must register its reference with the local Naming Service. This service acts as a "white pages" service and can be used by any application which needs to find the reference of a registered object. Whenever the Controller sends a message to a Client Factory, the only pieces of information at its disposal are the name of the host on which it resides and a preset communication port to the host (for simplicity we have defined the same port number for all Client hosts). Using this information, the Controller obtains from the ORB a reference to the Naming Service running on the given host which, in turn, provides a reference to the desired Client Factory.

Another parameter read by the Controller at startup is the host name on which the Scheduler resides. This information is passed to all Client Factories before any Clients are created and helps them locate the Scheduler and make the scheduling request.

Finally, the last parameters specified in the input file are:

- the period of time for which Clients are created;
- the maximum amount of time which can elapse between two consecutive Client creation requests made to the Client Factories by the Controller;
- the maximum execution time requested by a Client (i.e. the maximum amount of time a Client needs to use the resources);
- the number of resources available in the system;

When the Controller makes a request for a Client creation these parameters, except for the second one, are passed along. The Client Factory produces in response a Client with random parameter values, some of them being limited by the above maximum values. Besides influencing the client attributes, these parameters will affect a more global system property: the system load. By modifying their values we can perform tests for more relaxed or more heavily loaded systems.

A few more aspects need to be discussed at this point. Since this is a scheduling simulation, the system clocks of the different machines involved in the testing are used extensively. But it is very possible that these clocks do not show the exact same time value at the same moment in time. Therefore, the current time of the host on which the Controller runs is passed to all the other hosts for clock synchronization. Another issue is that the seed for the random number generator is set to be equal to the system time for each test pass in order to obtain a different set of random numbers for each test.

After all the system parameters are initialized, the Controller is ready to start the Client creation. At this point it enters a loop which has the following structure:

the Controller chooses randomly a host to which it will send the creation request. It obtains from the Naming Service the reference for the corresponding Client Factory and sends the message. A new creation request is made after a random period of time. The Controller exits the loop at the end of the testing period and terminates, but not before sending a termination message to all Client Factories.

6.1.2 Clients and Client Factories

The CORBA services [12] Life Cycle Service specifications provide a few guidelines on how CORBA objects can be created, copied, moved and destroyed. The Object Life Cycle is one of the most challenging topics in distributed systems and the way the OMG specifications suggest to solve the object creation issue is by using the *factory* design pattern. A factory is a CORBA object that offers one or more operations to create other objects of a particular type. In other words, factory operations in a distributed system play the role of the constructor in C++.

Let us now describe briefly how the factory pattern works in our case. As specified in the previous section, the Client Factories should be registered on every machine on which Clients will be started. To create a new Client object, the Controller obtains from the Naming Service the reference to the desired Client Factory and invokes an operation on it. The operation's implementation, which can be located on the same machine as the Controller application or on a remote host, creates a local Client object. It is obvious now that the major difference between a factory and a regular C++ constructor is that the factory operation can create a CORBA

object in a possibly remote address space, as opposed to a C++ constructor which always creates a C++ object in the local address space. It can also be noticed that one needs to instantiate a factory object in order to invoke the factory operations on it, whereas C++ constructors can be invoked C++ without having a preexisting object.

For a realistic simulation it is necessary that Clients originate independently on different hosts and request resources immediately after creation. The first requirement is met by implementing the Client creation request, *CreateClient ()*, made by the Controller, as a CORBA *oneway* operation. This means that the Controller is not blocked after issuing its request and can therefore proceed immediately to make new ones. Otherwise, it would have to wait until its request is completed. Unfortunately there is a limitation to oneway operations: they can be invoked only when no results are expected back by the caller and thus the caller does not have to wait for the operation to complete. The second requirement, which is the prompt registration of the Client with the Scheduler right after its creation is realized through multi-threaded programming. This aspect will be covered in more detail in the next following section. To conclude, by starting the Clients with oneway operations and by using a threaded policy for scheduling, it is guaranteed that Clients originate independently of each other and that they are executed in parallel. Hence, a sophisticated simulation of real systems is made possible.

Let us summarize the operations performed by a Client Factory. At the beginning it instantiates a Client Factory object and registers it with the Naming Service to be accessed by the Controller. After the Controller provides the name of the host on

which the Scheduler resides, the Factory obtains a reference to the Scheduler object from the Naming Service in a manner similar to that in which the Controller obtains the references to the Client Factories. At this point, the Factory is ready to create Clients whenever it receives a request, by generating random Client parameter values such as: start time, execution time, deadline, waiting cost and lateness penalty cost. Each creation is followed by a request for resource scheduling made on the behalf of the Client and for this reason the Client parameters are passed to the Scheduler. At the end of the test period the Client Factory receives a termination signal from the Controller, transmits it to the Scheduler and exits.

6.1.3 The Scheduler

The Scheduler is the key part of the simulation. Its task is to schedule the Clients created dynamically by the Client Factories based on the on-line algorithm presented in Chapter 5. Since our Client objects are tasks which require resource allocation, we will use from now on the terms client, job and task interchangeably.

At startup, the application instantiates a Scheduler object and registers its reference with the Naming Service. This enables the Clients to make the scheduling requests. Whenever the Scheduler receives such a call, it checks the status of the resources, analyzes the jobs waiting in the waiting queue and decides in accordance to its scheduling algorithm whether to assign the newly arrived job to a resource or to place it in the waiting queue. After the Controller and the Client Factories have terminated, the Scheduler continues to service all the jobs in the waiting queue and

exits in the end.

Let us consider the situation in which two different Clients originate at the same moment on two different machines. In this case they will send their scheduling requests concurrently. But as we know, the conventional programming methods do not allow two operations to be performed at the same time on the same object, in our case the Scheduler. Hence the Scheduler will start by completing one of the requests while blocking the other one. Only after the first request is completed, the second one can be considered. In the case of a heavily loaded system this can result in a queue of Scheduler calls waiting for a busy Scheduler. In order for our simulation to be accurate, it is imperative that the Scheduler learns about a newly generated Client immediately after its creation. We will accomplish this requirement with concurrent programming techniques.

CORBA allows different multi-threaded concurrency models to be established for the client and server activities of an application. The one that fits our situation and proves to be useful is the *thread-per-client server concurrency model*. In this activation policy, for every request sent by the clients to the server, i.e. the Scheduler, a new process is started to handle this request. This will allow the Scheduler to receive and attempt to resolve multiple requests in parallel.

A very common problem in multi-threaded programming occurs when processes have concurrent access to shared data. It is known that such situations can lead to data inconsistency. In our simulation the threads have to share objects such as the waiting list or the resources. The solution to this problem is to ensure that parallel processes have mutually exclusive access in time over the shared data. The CORBA

software used in the present simulation facilitates this task since it is provided with a high-level thread abstraction library called JThreads/C++ [14] similar to the one existent in the Java language. This library supports monitors that can assure the mutual exclusion with two easy to use classes: *JTCMonitor* and *JTCSynchronized*.

There exist two types of threads in the system. The first one is a thread started by us manually in the main procedure of the Scheduler implementation. This thread controls the scheduling process and requests a rescheduling whenever this is necessary according to the rescheduling criteria presented in Chapter 5. In addition, there are threads created by the ORB for each Client request. Such a process registers a new Client with the Scheduler and attempts to allocate a resource to it.

One of the limitations of the thread-per-client model is that the ORB allows only one active thread for each client in the user code. Since in our case each Client makes a single request, this limitation becomes of no importance. Another drawback could be the overhead created by new thread production, that such a concurrency model could induce. The performance measures presented in the CORBA software's specifications [15] show that the model is fairly efficient, but still induces some overhead especially for large numbers of jobs.

6.2 The Off-line System Architecture

The off-line scheduling simulator has three distinctive logical components: the Loader, the Off-line Scheduler and the Execution Simulator, each of them performing a very specific task.

6.2.1 The Loader

The Loader is the piece of code responsible for system setup at the beginning of each test. It loads from an input file the parameter values for the resources, namely the number of resources and the processing cost per resource and per time unit, as well as information regarding the jobs that need to be scheduled and forwards them to the Scheduler. The following attributes are attached to each job: start time, execution time, deadline, waiting cost and lateness penalty cost. Another responsibility of the Loader is to send these jobs to the Simulator at simulation time.

6.2.2 The Off-line Scheduler

As opposed to the on-line scheduling, in this case the scheduler and the simulator components are completely separated due to the static feature imposed.

Given a set of jobs and a fixed number of resources, the off-line scheduler has the role of finding the optimal resource distribution among the jobs, in accordance to the algorithm presented in Chapter 5. As described there, the scheduler builds various job sequences corresponding to the order in which the jobs would receive a resource and eliminate the sequences that prove to be inefficient. This process is repeated until the optimal job sequence is found. Because sequences are continuously created and eliminated, the necessity of using a flexible data structure to keep track of all sequences arises. We opted for the *vector* container from the Standard Template Library. Having a vector of sequences, an entry can be easily added to or removed from the vector with the help of stack operations such as *push_back()* and *pop_back()*

defined for vectors. These functions perform all the backstage work such as dynamic memory allocation and deallocation.

A very important issue in choosing an optimal schedule is that as the number of jobs competing for the resources increases, the number of job permutations to be considered increases dramatically. This leads to a heavy utilization of system resources and therefore, dynamic memory deallocation of all the variables which become useless while running the program was of great concern to us.

6.2.3 The Execution Simulator

After an optimal job sequence is established, the execution of the jobs on the resources is simulated. The jobs reach the resources in the order specified by the optimal sequence. When a job receives a resource, it uses it for a period of time equal to its execution time. Whenever a resource is released, it is reallocated to the next job in the remaining sequence.

6.3 Performance Tests

This section describes the performance tests carried out with the two test systems presented previously in this chapter and the two scheduling strategies presented in Chapter 5.

6.3.1 Experimental Configuration

For all performance tests the fundamental system configuration was kept identical to allow an easy comparison of the results. Two Sun Workstations having the parameters presented in Table 6.1 were used. The tests were set such that for the on-line scheduling the Controller was running on the first host and the Scheduler on the second one. Both machines were used for Client creation. For the off-line algorithm the jobs were sent from the first host to be scheduled on the second one.

Parameter	Host #1	Host #2
Node Name	z.cs.okstate.edu	a.cs.okstate.edu
Machine Type	Sun SparcStation 5	Sun Microsystems Enterprise 3000
Number of Processors	1	2
CPU Info	Sun Microsystems Sparc	UltraSparc 250 Mhz
Main Memory	32 MB	512 MB
Operating System	Solaris 7	Solaris 7
Name of ORB	ORBacus 3.2.1 provided with JThreads/C++ 1.0.8 support	ORBacus 3.2.1 provided with JThreads/C++ 1.0.8 support
Language Mapping	C++	C++
Compiler Info	gcc 2.95.2	gcc 2.95.2
Measurement Method	gettimeofday	gettimeofday

Table 6.1: Operating environment used for tests.

6.3.2 Running the Test

The tests for the two schedulers are performed by running them consecutively on the same set of jobs to be processed. We begin with the dynamic algorithm by starting its component applications in the following order: first we start the Naming Service application as a background process on each machine where Clients are going to be created and on the host on which the Scheduler application will run. Next we start

the Client Factories and the Scheduler. All these applications perform some startup operations such as variable initializations and then enter their event loop waiting for incoming requests. Only when the Controller is started, jobs are randomly generated and the actual scheduling process begins. The on-line test is followed by the off-line one which first starts both the Scheduler and the Execution Simulator and next the Loader.

6.3.3 Test Input

The goal of the tests is to analyze the behaviour of the two algorithms for various system load values, where the load is defined as the average amount of work per resource. The load within the test system is determined by three factors: it increases with the number and size of the jobs to be scheduled and decreases with the number of resources. Hence, we can express the load using the notations from Section 5.2 as:

$$= const \times \frac{n \times (\sum e_i/n)}{m} = const \times \frac{\sum e_i}{m} \quad (6.1)$$

where *const* is a constant and $1 \leq i \leq n$. The system load is tuned indirectly for the on-line algorithm by giving different values to the input parameters read by the Controller from the input file. The number of jobs in the system was modified for different test passes by varying the Controller maximum waiting time between two consecutive Client creations in the range 9 - 19 seconds. The execution time requested by a job took values in the range 1 - 50 seconds and systems having 1 - 5 resources were considered. Each of the two scheduling strategies was tested for a total of approximately 20 hours.

To compare the performance of the two scheduling strategies we need to run them on the same sets of jobs. For each test, the on-line algorithm generates these jobs randomly and the final, complete set is saved in a temporary file which will serve as input for the off-line algorithm.

Let us discuss in more detail this aspect. As we know, the off-line scheduling method is a static one and therefore all jobs in the system need to be known ahead of time, prior to their arrival. This is of course impossible in practice, but there are a few techniques which attempt to make predictions about the job system. Usually these techniques rely on the past history, i.e. on information about jobs created previously in the system. In this case the assumption is made that the next coming job set will have parameters similar to the previous ones and a schedule is set for these predicted jobs. When the real jobs are created they are allocated to the resources according to the schedule generated for the predicted ones, which means that the worse the predictions, the worse the off-line algorithm will perform. Good results can be obtained for periodic task systems where job parameters follow specific distributions within the periods. But for random jobs, as in our case, it becomes impossible to make reasonable predictions and the performance of the off-line scheduler is poor. Another problem arising for the particular case studied by us is that each job is characterized by five different parameters, and thus the predictions are even more complicated. Therefore, we chose to not make any predictions about the incoming jobs. After saving the set of tasks on which the on-line algorithm was tested, we forward them to the off-line algorithm and the static scheduling is performed. Since these jobs are the "real" jobs created by the system, the situation is equivalent to

the ideal case in which the off-line algorithm would predict the future jobs with no errors. In other words, we perform the off-line tests for ideal situations, whereas the on-line ones are performed for realistic situations. Hence the performance of the dynamic method is compared to the best case performance of the static one.

6.3.4 Test Results

To compare the performance of the two algorithms several performance measurement objects such as: number of late jobs, total scheduling cost, waiting cost, lateness penalty cost and processing cost were analyzed. More exactly, we trace their evolution as changes in the system load occur.

Because our work focuses on the real-time systems, the measurement parameter of most importance becomes the number of jobs that do not meet their deadline during a test pass. The variation of this parameter with the system load is presented in Figure 6.2.

It can be noticed that for reasonable loads the on-line algorithm manages to minimize the number of jobs that fail to meet their deadline. This can be explained considering how the algorithm works: whenever a resource becomes available it is allocated to the job with the smallest slack time value, i.e. to the job which is in greatest danger to miss its deadline. The resources are shared in time by such problematic jobs, each job receiving a time slice proportional to the inverse of their slack times. This policy proves to save as many jobs as possible if the number of jobs in danger is reasonable. However, as their number increases, all the problematic

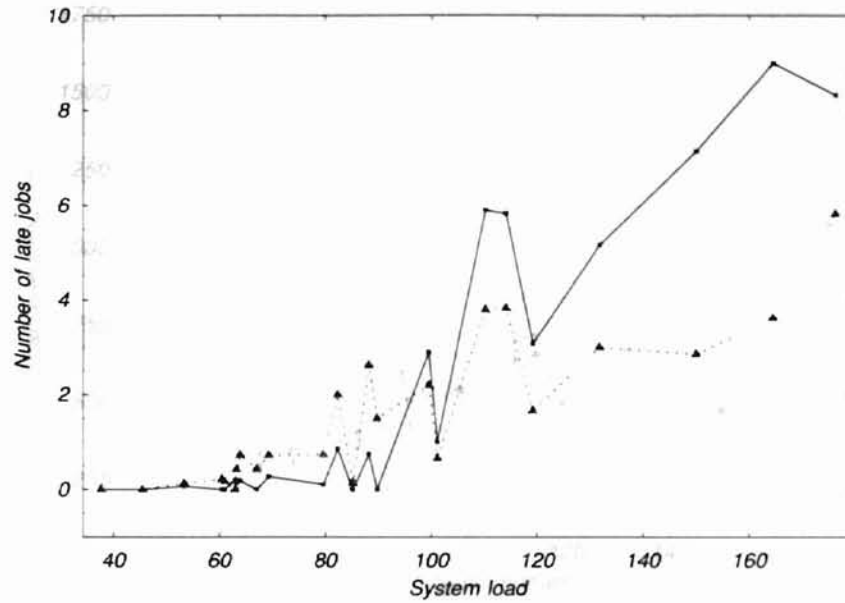


Figure 6.2: The Number of Late Jobs as a function of the System Load for On-line (solid) and Off-line (dotted) algorithms.

jobs are allocated resources for a time period which usually is not large enough to ensure that delays are avoided. For heavily loaded systems the off-line policy becomes more efficient because it sacrifices from the beginning some of the problematic jobs in order to save as many as possible of the remaining ones. The off-line algorithm decides which jobs are of less importance and have small or no chances to be executed on time and leaves them to be last scheduled. Hence the number of the remaining problematic jobs decreases and their chances to meet their deadline improve. This explains the better performance of the off-line algorithm for busy systems.

The second important parameter used as a performance measurement object is the total scheduling cost defined by the Equation 5.2. Its dependency on the system load is drawn in Figure 6.3. In order to understand and interpret this result we

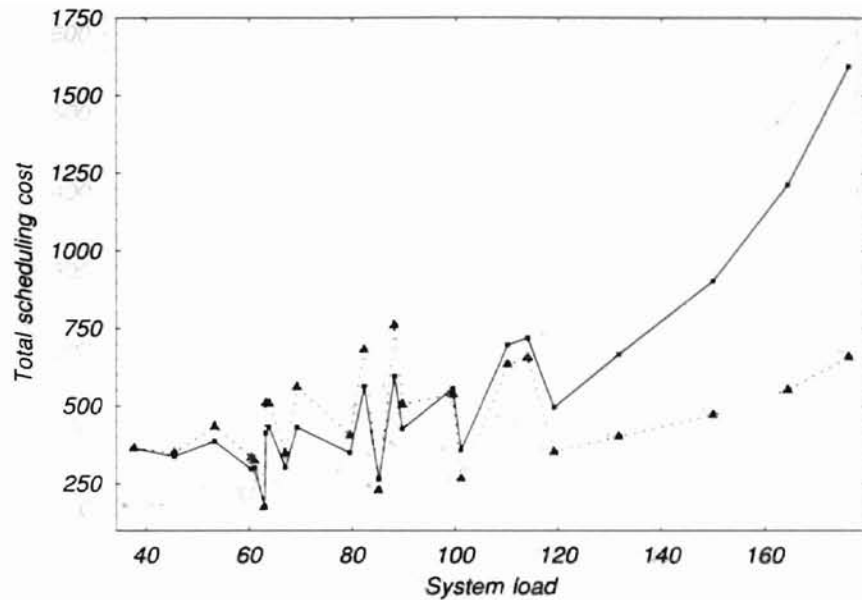


Figure 6.3: The variation of the Total Scheduling Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.

should first analyze the behaviour of its components, namely the waiting cost, the lateness penalty cost and processing cost. These results are shown in Figures 6.4, 6.5 and 6.6 respectively. They all follow a general pattern in which the on-line algorithm performs slightly better for reasonable system loads and worsens for large load values. The behavior of the waiting and lateness costs can be explained through the same mechanism used for the number of late jobs. As previously described, the off-line algorithm chooses some less important jobs having small waiting and lateness penalty costs and large sizes and postpones them until all the other jobs are completed, ignoring the fact that they will miss their deadline. The advantage of this policy in the case of busy systems is that the expensive jobs are executed as early as possible, hence the waiting and lateness costs are optimized. On the other

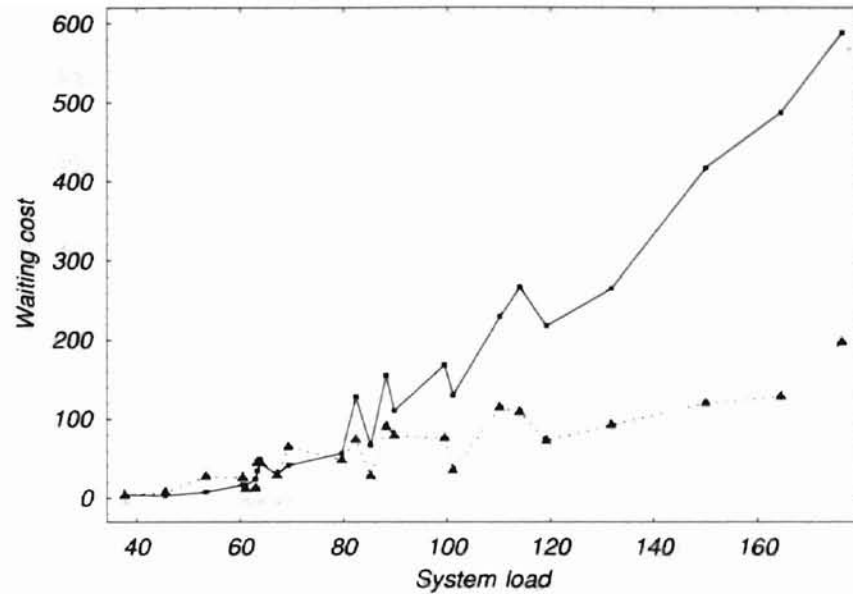


Figure 6.4: The variation of the Waiting Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.

hand, the on-line scheduler attempts to service all the problematic jobs concurrently through resource sharing. This leads to long waiting times and late completion times that explain the results obtained. The costs play a much less important role in dynamic scheduling. They are considered only in cases in which there are two or more problematic jobs having identical slack time values

Slightly different than the other plots is that of the processing cost versus system load presented in Figure 6.6.

In this case, the on-line algorithm has an overall better performance. The explanation for this behavior is the following. As mentioned before, the off-line algorithm has all the job system information prior to runtime and it constructs a schedule that attempts to minimize the measurement parameters. In the case of relaxed systems,

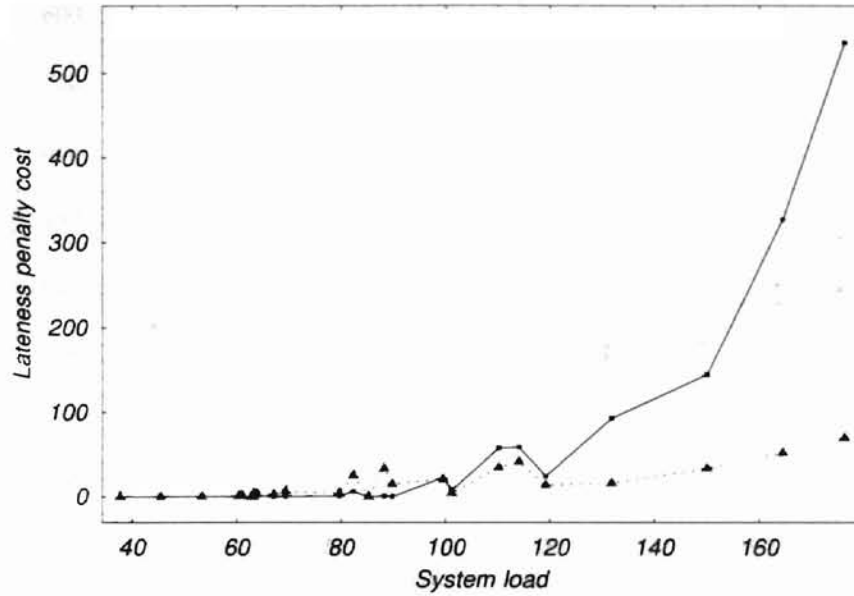


Figure 6.5: The variation of the Lateness Penalty Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.

this may result in schedules with gaps, since the algorithm is nonpreemptive. A gap in the schedule is defined as the period of time for which a resource is idle even though there are jobs waiting to be scheduled. Gaps can occur in cases when the scheduler knows that a more expensive job will soon be created and that a better overall performance will be obtained if the idle resource is reserved for it. Hence the performance is sometimes improved at the cost of a worse completion time and processing cost. The on-line algorithm does not have information about future jobs, is preemptive and dynamic and therefore gaps do not occur in its schedule. For heavily loaded systems, gap-free schedules are created, even in the case of the off-line algorithm. For this case, the processing time of the on-line schedule becomes slightly worse due to communication overhead between the applications involved.

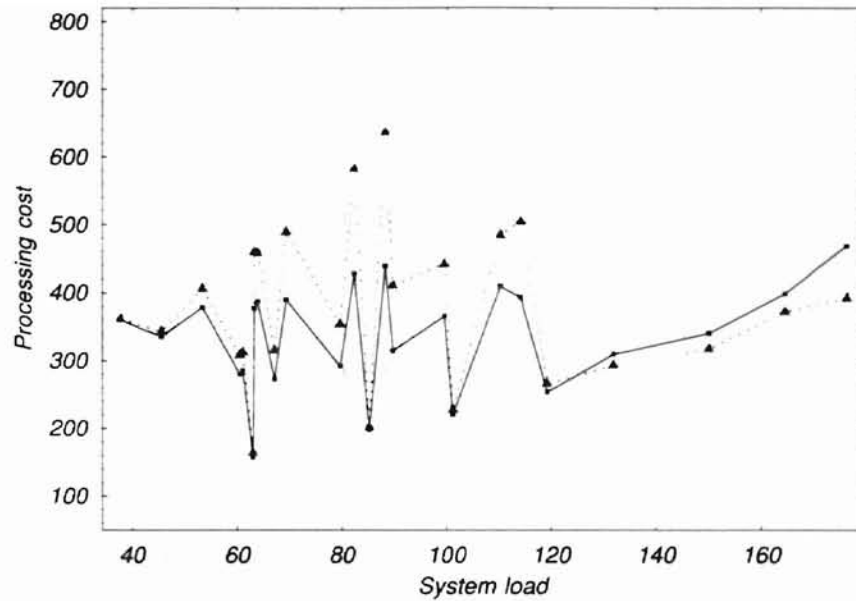


Figure 6.6: The variation of the Processing Cost with the System Load for On-line (solid) and Off-line (dotted) algorithms.

Finally we can come back and analyze Figure 6.3. For small loads, the difference between the results produced by the two algorithms for the waiting and lateness costs are negligible compared to the differences in the processing cost. Hence the obvious dominance of the on-line algorithm over the off-line one, visible in Figure 6.3, is inherited from the processing cost. For large load values its contribution becomes irrelevant and therefore the features from Figures 6.4 and 6.5 become dominant.

the dynamic scheduling method. The cost is insignificant, being compensated by the reduction of lateness. As the load increases, the performance of

the dynamic scheduling method is better because the on-line

CHAPTER 7

CONCLUSIONS

In this work we propose an on-line scheduling strategy which could be implemented at CORBA level for distributed real-time systems. Although a good end-to-end performance in CORBA environments is influenced by many factors as: messaging mechanism, network adapters design, protocol implementation, integration with the operating system, etc., the most important factor remains the scheduling policy. A distributed client-server type test system involving two hosts has been implemented in order to analyze the behaviour of the dynamic algorithm. To obtain a more realistic simulation, multi-threaded programming was applied for job registration with the scheduler and for job scheduling. The performance of the algorithm was evaluated by comparing it to the best case performance of a traditional off-line strategy. For both algorithms the tests were performed with different loads. Output parameters such as the number of late jobs and the total scheduling cost defined as the sum of waiting, lateness and processing costs were used as primary measurement objects.

We found that the dynamic scheduling method produces good results for both the number of late jobs and the total scheduling cost for reasonable load values.

The communication overhead for this case is insignificant, being compensated by a good job management. Unfortunately, as the load increases, the performance of the algorithm decreases dramatically. This phenomenon occurs because the on-line scheduling policy is to share the resources between the jobs which have the smallest chances to finish on time, rather than to pick some of the most important problematic jobs and to concentrate its effort on their completion, as the off-line algorithm does. Another reason for this worse performance is that as the number of clients increases, the communication overhead between the applications involved increases as well. We need not forget though that the results are compared with the results produced by the ideal case off-line scheduling, hence we could say that the performance of the on-line algorithm is satisfactory.

For real-time systems, an important feature offered by the static schedule is that it provides information about the task system schedulability prior to run-time. Another advantage of the off-line scheduling is the low run-time computation overhead. However, the off-line algorithm has its drawbacks. First, the idea of learning the task parameters at arrival time (as used in the on-line scheduling) seems more natural and more feasible than the requirement of knowing all this information ahead of time in the case of the off-line algorithm. The latter becomes even more unnatural when the jobs have totally random arrival times, which is exactly our case. Another drawback of off-line algorithms is that the computation of the optimal schedule prior to the run-time is very time consuming for large numbers of tasks in the off-line case. We can conclude that on-line scheduling approaches offer potential solutions to these complex problem but at some significant cost.

As further work, a mixed on-line/off-line algorithm could be considered for the system in study. Such an algorithm could lead to a higher resource utilization than purely static scheduling strategies with acceptable run-time costs.

BIBLIOGRAPHY

1. Kelly, M. (1997) *Robustness for an Advanced Real Time*

ii

iii

iv

202

BIBLIOGRAPHY

Journal of Real-Time Scheduling for Multimedia / Bell System

- [1] APM Ltd. DIMMA, "A Novel Design for an Advanced Real-Time Object Request Broker (ORB)".
<http://www.ansa.co.uk/ANSATech/software/dimma.tar.gz>.
- [2] Borodin, A. and El-Yaniv, R., "Online Computation and Competitive Analysis", Cambridge University Press, 1998.
- [3] Coffman, E.G. Jr. and Denning, P.J., "Operating Systems Theory", Englewood Cliffs, NJ Prentice-Hall 1973.
- [4] Davis, E. and Jaffe, J.M., "Algorithms for scheduling tasks on unrelated processors", J. ACM, 28(4):721-736, 1981.
- [5] Fiat, A. and Woeginger, G.J., "Online Algorithms", Lecture Notes in Computer Science 1442, Springer-Verlag 1998.
- [6] Gokhale, A. and Schmidt, D., "Measuring the performance of Communication Middleware on High-Speed Networks", Special Interest Group on Data Communication, August 1996.
- [7] Gokhale, A. and Schmidt, D., "Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks", ICDCS, 1997.

- [8] Goyal, P., Guo, X. and Vin, H.M., "Hierarchical CPU Scheduler for Multimedia Operating Systems", Second Usenix Symposium on Operating System Design and Implementation, 1996.
- [9] Graham, R.L., "Bounds for certain multiprocessor anomalies", Bell System Technical Journal, 45:1563-1581, Nov. 1966.
- [10] Hong, K.S. and Leung, J.Y.-T., "On-line scheduling of real-time tasks", IEEE Transactions and Computers, 41(10):1326-1331, 1992.
- [11] IONA Technologies, <http://www.iona.com/products/orbix>.
- [12] Object Management Group, "CORBA services: Common Object Services Specification". <ftp://ftp.omg.org/pub/docs/formal/98-07-05.pdf>, November 1997.
- [13] Object Management Group, "Realtime CORBA Version 1.0 - Proposal", <http://www.omg.org/cgi-bin/doc?orbos/99-02-12>, 1999.
- [14] Object Oriented Concepts Inc., "JThreads/C++. Java-like Threads for C++", <ftp://ftp.ooc.com/pub/JTC/1.0/JTC-1.0.10.pdf.gz>, 2000.
- [15] Object Oriented Concepts Inc., "ORBacus for C++ and Java", <ftp://ftp.ooc.com/pub/OB/3.3/OB-3.3.1.pdf.gz>, 2000.
- [16] Rajaraman, M.K., "A parallel sequencing algorithm for minimizing total cost", Naval Research Logistics Quarterly 24(3):775-783, Sept 1977.
- [17] Sahni, S. and Cho, Y., "Nearly on line scheduling of a uniform system with release times". SIAM J. Comput., 8(2):275-285, 1979.

- [18] Schmidt, D., Bector, R., Levine, D., Mungee, S. and Parulkar, G., "TAO: A Middleware Framework for Real-Time ORB Endsystems", IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, 1997.
- [19] Schmidt, D., Gokhale, A., Harrison, T. and Parulkar, G., "A High Performance Endsystem Architecture for Real-Time CORBA", IEEE Communications Magazine, 14(2), 1997.
- [20] Silberschatz, A. and Galvin, P.B., "Operating System Concepts", Addison-Wesley, 1997.
- [21] Wolf, L., Burke, W. and Vogt, C., "Evaluation of CPU Scheduling Mechanism for Multimedia Systems", Software Practice and Experience, 26:375-398, 1996.

VITA 2

Diana Aura Dumitru

Candidate for the Degree of

Master of Science

Thesis: ON-LINE SCHEDULING FOR REAL-TIME CORBA ENVIRONMENTS

Major Field: Computer Science

Biographical:

Education: Graduated from C. D. Loga High School, Timisoara, Romania in 1991; received a Master of Science Degree in Physics from Western University of Timisoara, Timisoara, Romania in 1996. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in July, 2000.

Experience: Employed by the Oklahoma State University, Computing and Information Services as a Computer Lab Assistant, 1998-1999.